

The effects of modeling and inspection methods upon problem solving in a computer programming course

by JAMES BRUGGEMAN

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Education
Montana State University
© Copyright by JAMES BRUGGEMAN (1985)

Abstract:
Reading, composing, and debugging computer programs involve problem solving in a conceptually complex domain. Some educators have viewed computer programming both as a worthy subject of study by grade school children and as a promising means of teaching general, problem-solving skills. One goal of this study was to test the supposition that problem-solving skills, learned through computer programming, may prove useful in related domains, specifically, word problems of a nonroutine nature. This was accomplished by conducting an experiment organized according to a five group, pretest-posttest design with one control group and four treatment groups. Another goal was to assess the effects of four instructional methods upon solving computer programming problems. Effects of these instructional methods were tested by means of a four group, posttest design. Subjects for both experiments were 113 fifth and sixth graders enrolled in a computer programming course.

Participation in the computer programming course had no effect upon participants' ability to solve nonroutine, mathematical word problems, but significant associations were found between the Iowa Problem Solving Test pretest and the programming posttest. Neither the method of instruction (models or inspection methods) nor the relative complexity of programs used in instruction produced significant among-group differences in solving programming problems, but training with less complex programs improved the subjects' debugging skills. Program complexity hindered the solution of programming problems, but none of the instructional methods gave subjects a relative advantage in solving these more complex programs.

Practice, gender, and grade level exerted the greatest influence on problem-solving performance.

Results of this study add credibility to the argument which favors the teaching of domain-specific, rather than general, problem-solving skills. The absence of differences among the instructional methods is ascribed to several factors, including the random effects of practice and gender variables. Instructional variables may have produced differences during the initial stages of instruction before other factors came into play. Specific recommendations are made for the use of models and inspection methods in computer programming courses. An argument is also made for better measures problem-solving performance.

THE EFFECTS OF MODELING AND INSPECTION METHODS UPON PROBLEM SOLVING IN A COMPUTER PROGRAMMING COURSE

by

James George Bruggeman

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Doctor of Education

MONTANA STATE UNIVERSITY
Bozeman, Montana

March 1985

D378
B833
Cop. 2

APPROVAL

of a thesis submitted by

James George Bruggeman

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the College of Graduate Studies.

3/26/85
Date

Chairperson, Graduate Committee

Approved for the Major Department

3/26/85
Date

Head, Major Department

Approved for the College of Graduate Studies

3-27-85
Date

Graduate Dean

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a doctoral degree at Montana State University, I agree that the Library shall make it available to borrowers under rules of the Library. I further agree that copying of this thesis is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for extensive copying or reproduction of this thesis should be referred to University Microfilms International, 300 Zeeb Road, Ann Arbor, Michigan 48106, to whom I have granted "the exclusive right to reproduce and distribute copies of the dissertation in and from microfilm and the right to reproduce and distribute by abstract in any format."

Signature James J. Ruggeman

Date March 18, 1985

"What is laid down, ordered, factual, is never enough to embrace the whole truth: Life always spills over the rim of every cup."

Boris Pasternak, 1890-1960

TABLE OF CONTENTS

	Page
1. INTRODUCTION	1
Problem-Solving Research and Instruction	1
Computer Programming as Problem Solving	5
Learning through Interactive Programming	8
Statement of the Problem	11
Need for the Study	12
LIP Claims	13
Instructional Methods	13
Conceptually Complex Computer Languages	14
Modeling and Instruction	15
Perception and Instruction	16
Presentation of the Study	17
Summary	18
2. REVIEW OF THE LITERATURE	20
Understanding and Problem Solving	21
Procedural Specification	24
Novice-Expert Distinctions	30
Barriers to Understanding Programming Languages	40
Propositional Structures	41
Text Comprehension Strategies	44
Schemas in Natural and Programming Languages	46
Program Complexity	58
Instructional Implications	61
Inspection Methods	62
Modeling	66
Summary	72
3. PROCEDURES	75
Population Description and Sampling Procedures	76
Research Site	76

TABLE OF CONTENTS--Continued

Recruitment of Subjects 76
Assignment of Subjects and Sample Size 78
Description of the Treatments 80
Treatment Groups 80
Experimental Validity 81
General Problem-Solving Methods 84
Modeling Methods 85
Inspection Methods 89
Methods of Collecting Data 95
Testing Procedures 95
The Iowa Problem Solving Test 96
The Programming Test 98
Think-Aloud Protocols 108
Statement of Hypotheses 110
Statistical Analyses 115
Summary 115

4. RESULTS 119

Experiment 1, The Effect of Programming Instruction upon Mathematical Problem Solving 119
Hypothesis 1 120
Hypothesis 2 121
Hypothesis 3 121
Post Hoc Analysis of IPST Data 123
Experiment 2, Hypotheses Related to Programming Instruction and Problem Solving 124
Solution and Error Distributions 125
Hypothesis 4 127
Hypothesis 5 128
Hypothesis 6 130
Hypothesis 7 130
Hypothesis 8 131
Hypothesis 9 131
Post Hoc Analysis 132
Pooled Group Analysis of Solutions 132
Analysis without Outliers 132
Error Analysis 133
The Program Task Type Variable 135
The Problem Relatedness Variable 139
The Program Complexity Variable 151

TABLE OF CONTENTS--Continued

The Effect of Practice on Problem-Solving Performance 165
Gender and Problem-Solving Performance ... 170
Grade Level and Problem-Solving Performance 174
Summary 177

5. CONCLUSIONS 181

Summary of Results 181
Discussion of Results 184
The Transfer Question 184
Method of Instruction 190
Post Hoc Analysis of Error Patterns 194
Post Hoc Analysis of Complexity 195
Post Hoc Analysis of Program Task Type Data 199
Post Hoc Analysis of Gender Effects 201
Recommendations for Future Research 202
Educational Implications 205
Final Summary 208

6. BIBLIOGRAPHY 214

7. APPENDICES 230

Appendix A - Glossary of Terms 231
Appendix B - Additional Tables 237

LIST OF TABLES

Table		Page
1	Pretest: Iowa Problem-Solving Test (Form 561)	79
2	ANOVA: IPST Pretest by Experimental Groups	79
3	Alternate Forms Reliablity of the Iowa Problem Solving Test by Grade and Subtest	97
4	Split Half Reliability, Computer Programming Test	108
5	Group Performance: Iowa Problem Solving Test	120
6	ANOVA: IPST Posttest by Treatment Groups	122
7	Pearson R's: IPST Pretest with Programming Test Totals	123
8	Solution Means: Total Test	126
9	Error Means: Total Programming Test	127
10	ANOVAs of Total Test: Solutions and Errors by Treatment Groups	129
11	Comparison of Total Syntax Errors using the Duncan Studentized Range Test	134
12	ANOVAs: Programming Task Type by Group, The Interpretation of Problems	136
13	ANOVAs: Programming Task Type by Group: The Generation of Problems	137

LIST OF TABLES--Continued

14	Comparison of Syntax Errors in Generation Problems using the Duncan Studentized Range Test	137
15	Comparison of Data Errors in Generation Problems using the Duncan Studentized Range Test	138
16	Solution Means along the Problem Relatedness Axis	141
17	ANOVAs of Problem Relatedness Categories: Solutions by Treatment Groups	142
18	ANOVAs of Problem Relatedness Categories: Solution and Error Patterns by Treatment Groups	145
19	Comparison of Solutions of Equivalent Problems using the Duncan Studentized Range Test	146
20	Comparison of Syntax Errors in Equivalent Problems using the Duncan Studentized Range Test	146
21	ANOVAS: Pooled Group Performances in Equivalent Problems	147
22	Comparison of Syntax Errors in Identical Problems using the Duncan Studentized Range Test	149
23	Solution Means along the Complexity Axis	153
24	ANOVAs and Multiple Comparisons of Mean Solutions within the Complexity Categories of the Simple Modeling Group	154
25	ANOVAs and Multiple Comparisons of Mean Solutions within the Complexity Categories of the Complex Modeling Group	155

LIST OF TABLES--Continued

26	ANOVAs and Multiple Comparisons of Mean Solutions within the Complexity Categories of the Simple Inspection Methods Group	156
27	ANOVAs and Multiple Comparisons of Mean Solutions within the Complexity Categories of the Complex Inspection Methods Group	157
28	ANOVAs: Program Complexity by Treatment Group, Program Elements: Simple Statements	160
29	Comparison of Syntax Error Patterns in Simple Statements using Studentized Range Tests	161
30	ANOVAS: Program Complexity by Treatment Group, Program Connections: Linear Block, Forward Branching and Loop Structures	162
31	Comparison of Data Error Patterns in Linear Blocks using Studentized Range Tests	163
32	MANOVA: Total Solutions by Instructional Method and Practice Level	167
33	MANOVA: Total Syntax Errors by Instructional Method and Practice Level	167
34	MANOVA: Total Control Flow Errors by Instructional Method and Practice Level	168
35	MANOVA: Total Data Errors by Instructional Method and Practice Level	168
36	MANOVA: Total Solutions by Complexity of Training Materials and Practice Level	169
37	MANOVA: Solutions by Instructional Method and Gender	171
38	Comparison of Practice Level, Grade, and Gender	172

LIST OF TABLES--Continued

39	Comparison of Practice Level and Treatment Group	173
40	t Tests of Differences between Solution Means within the Problem Relatedness Categories of the Simple Modeling Group	238
41	t Tests of Differences between Solution Means within the Problem Relatedness Categories of the Simple Inspection Methods Group	238
42	t Tests of Differences between Solution Means within the Problem Relatedness Categories of the Complex Modeling Group	239
43	t Tests of Differences between Solution Means within the Problem Relatedness Categories of the Complex Inspection Methods Group	239
44	MANOVA: Syntax Errors by Instructional Method and Gender	240
45	MANOVA: Control Flow Errors by Instructional Method and Gender	240
46	MANOVA: Data Errors by Instructional Method and Gender	241
47	MANOVA: Solutions to Program Interpretation Problems by Instructional Method and Gender	241
48	MANOVA: Solutions of Related Problems by Instructional Method and Gender	242
49	MANOVA: Solutions within Problem Complexity Categories by Instructional Method and Gender	243
50	MANOVA: Solution of Problems, Complexity Categories by Materials Complexity and Gender	244
51	MANOVA: Solutions of Related Problems by Instructional Materials and Gender	245

LIST OF TABLES--Continued

52 MANOVA: Total Syntax Errors by Instructional Methods and Grade 246

53 MANOVA: Syntax Errors within the Relatedness Categories by Instructional Methods and Grade 247

54 MANOVA: Syntax Errors within the Relatedness Categories by Instructional Materials and Grade 248

LIST OF FIGURES

Figure		Page
1	A Computer Program Expresed in BASIC and Propositional Notation	51
2	Chunkwise Decomposition of a BASIC Program	52
3	Temporal-Sequential Program Schema (Expressed in Propositional Notation)	57
4	Chunking Program Schema	57
5	General Problem-Solving Methods Taught in the Problem Sets	85
6	Two Parallel BASIC Programs used in Instruction	89
7	Chunks used in the Inspection Methods Treatments (The BASIC Toolbox)	91
8	Comparison of the Instructional Treatments of the Simple and Complex Inspection Methods Groups	94
9	Matrix of Computer Test Variables	100
10	Problem Relatedness Variables	103
11	Relationship of the Main Test Problems to the Syntax, Context, and Content of the Target Problems	104
12	Think-Aloud Protocols: Computer Program Interpretation and Composition Tasks	109

Abstract

Reading, composing, and debugging computer programs involve problem solving in a conceptually complex domain. Some educators have viewed computer programming both as a worthy subject of study by grade school children and as a promising means of teaching general, problem-solving skills. One goal of this study was to test the supposition that problem-solving skills, learned through computer programming, may prove useful in related domains, specifically, word problems of a nonroutine nature. This was accomplished by conducting an experiment organized according to a five group, pretest-posttest design with one control group and four treatment groups. Another goal was to assess the effects of four instructional methods upon solving computer programming problems. Effects of these instructional methods were tested by means of a four group, posttest design. Subjects for both experiments were 113 fifth and sixth graders enrolled in a computer programming course.

Participation in the computer programming course had no effect upon participants' ability to solve nonroutine, mathematical word problems, but significant associations were found between the Iowa Problem Solving Test pretest and the programming posttest. Neither the method of instruction (models or inspection methods) nor the relative complexity of programs used in instruction produced significant among-group differences in solving programming problems, but training with less complex programs improved the subjects' debugging skills. Program complexity hindered the solution of programming problems, but none of the instructional methods gave subjects a relative advantage in solving these more complex programs. Practice, gender, and grade level exerted the greatest influence on problem-solving performance.

Results of this study add credibility to the argument which favors the teaching of domain-specific, rather than general, problem-solving skills. The absence of differences among the instructional methods is ascribed to several factors, including the random effects of practice and gender variables. Instructional variables may have produced differences during the initial stages of instruction before other factors came into play. Specific recommendations are made for the use of models and inspection methods in computer programming courses. An argument is also made for better measures problem-solving performance.

Chapter I.

INTRODUCTION

Problem-solving Research and Instruction

Educators have been slow to adapt their teaching methods to the findings of recent research on human problem solving. Teaching problem solving skills has become a subject of increasing interest among teachers and administrators (Lochhead, 1981; Moses, 1982). Likewise, cognitive psychologists have developed their discipline to the point of constructing a general theory of human problem-solving. However, a productive intermingling of the insights of the cognitive psychologist with the concerns of the practical educator has not occurred.

Cognitive theories of human problem solving have had little influence on instructional design. This is due partially to the reigning influence of behaviorism upon instructional methods and to an emerging research paradigm within cognitive psychology itself. Behavioral psychologists have attempted to identify and test the effects of teaching and learning variables, resulting in such fruitful teaching methods as programmed learning and direct instruction. In contrast, cognitive psychologists have been more concerned with describing cognition and the

mental structures resulting from cognition (Mayer, 1981; Davis, 1983). The cognitive psychologists have tended to adopt single-subject, rather than group, research designs, and their research methods have been ethnographic, deemphasizing instructional intervention in favor of naturalistic observation. Although this preoccupation with cognitive structures to the detriment of instructional function has been recognized by some cognitive psychologists, their research paradigm remains less interventionist than that held by the behaviorists and, therefore, less useful to educators (Catania, 1978; Voss, 1978).

Educators and psychologists also have viewed problem solving differently. Educators often have confused problem solving with those "higher," creative endeavors in literature, art, mathematics and scientific research that are concerned as much with the generation of problems as with their solution (Estes, 1978; Greeno, 1978). Teaching problem solving frequently has been equated with teaching "critical thinking," "divergent thinking," or "creativity." However, problem solving consists of more mundane behaviors. In most disciplines and in everyday life problems are posed by outside agents, not discovered or generated by the problem solver. Solutions require not the formation of completely new knowledge but only the partial

restructuring and application of previously acquired, problem-related knowledge.

These differing views have resulted in contrasting, but not necessarily conflicting, definitions of problem solving. Educators traditionally have defined problem solving as a "transfer of learning;" however, many cognitive psychologists have come to view problem solving as alternate processes of "recognition" and "search" (Simon, 1981). In a simple, but elegant formulation of the established definition, Moses (1982) stated that problem-solving is the process of applying previously acquired knowledge to new, unfamiliar, but related situations. Problem solving in the established tradition is portrayed as a specific variety of "general-case learning" involving stimulus generalization and response differentiation (Carnine & Becker, 1982). In contrast, cognitive psychologists, inspired by the artificial intelligence research, initially defined problem solving as "search" through a "search space" to achieve a goal (Newell & Simon, 1972; Stefik et al., 1983). In the "search" metaphor, problems are represented as initial states, solutions as goal states, and the search space is defined as the set of possible moves or steps leading from the initial problem state to the solution. Problem-solving is the process of searching through the space of possible or

partial solutions for those solutions that satisfy the goal.

In the most recent literature of cognitive psychology, the problem-solving behaviors of experts have been contrasted with those of novices. Although the metaphor of search still appropriately describes the behavior of novice problem solvers, the behavior of expert problem-solvers has been characterized as a form of "perceptual recognition." Expert problem solving seems to rely less on reasoning skills than on skilled perception, matching perceptual patterns to a huge store of very specific information and experience stored in long-term memory (Simon, 1981). The notion that expert problem solving is based on "search" or "perceptual recognition," rather than "creative thinking," has pervaded the research literature, yet it has not been widely accepted by educators.

The cognitive psychologists' view of problem solving as search did not have an immediate pedagogical impact because it was developed out of an analysis of laboratory tasks, such as cryptoarithmetic and the Tower of Hanoi puzzle. Although these tasks are sources of well-structured problems, their solutions do not require extensive knowledge of specific subject areas (Greeno, 1976; Simon, 1973). Analyses of human problem solving based on these well-structured, conceptually simple puzzles initially seem of little use to educators, who are required

to teach information related to specific subject areas. Many of the procedures and much of the information, which educators are expected to teach, are conceptually complex and loosely coupled. Problems in these domains are ill-structured, and their solution often requires making inferences from inevitably incomplete knowledge (Collins et al., 1975; Stevens & Collins, 1980). Because of the intricacies of problem solving, educators need a greater understanding of how students, particularly elementary grade school children, best acquire conceptually complex knowledge. Educators also need to learn how to teach the use of this conceptually complex knowledge to solve problems within specific domains such as mathematics, the social sciences, or computer programming.

Computer Programming as Problem Solving

Computer programming is what Bhaskar and Simon (1977) call a "semantically rich" domain. "Conceptually complex" is a more apt term. Computer programs are conceptually complex devices. This complexity is reflected both in the density of information within a program and in the numbers of relations or connections among individual units of information. Reading, composing and debugging programs thus comprise problem-solving behavior in a conceptually complex domain.

The kinds of knowledge and behavior required by computer programming also are complex. Computer programming requires specialized perceptual activity, a knowledge of a complex set of propositions, and the flexible use of certain procedures. Perceptual behavior involves recognizing and classifying patterns, images, and layouts of stimulus fields. Programs consist of patterns of statements and patterns of connections between statements. Perception of familiar patterns within programs and program specifications evoke a person's propositional knowledge about the problem and the procedures appropriate for its solution (Hinsley, Hayes, & Simon, 1977; Larkin et al., 1980).

Propositional, or declarative, knowledge is verbal behavior, consisting of statements about concepts, word meanings, events, and episodes. In computer programming, propositional knowledge includes an understanding of the function of individual commands as well as the overall syntax of the programming language. Propositions about the problem domain also allow the classification of specific programs and programming problems, thereby making possible inferences about the appropriateness of certain solution procedures. Perceptions and propositions are what Sacerdotti (1977) calls "domain knowledge" because the important elements of problem situations, the lawful relations among these elements, and the permissible

procedures for achieving solutions are stored as propositions in long-term memory and are accessed from memory by perceptual events. Propositional and perceptual behavior, therefore, are the source of domain-specific constraints on the solution of programming problems.

Procedural knowledge is our knowledge of how to do things. It includes the "plan knowledge" (Sacerdotti, 1977), "strategic knowledge" (Greeno, 1980) and "planning" (Goldstein & Miller, 1976) involved in identifying and setting subgoals useful in problem situations and forming plans that guide in the search for solutions. Planning consists of arranging a sequence of actions from a set of constraints to achieve goals. Goldin (1979) makes a useful distinction among three kinds of procedural knowledge used in planning the solution of problems: algorithms, strategies, and heuristics. Each kind of knowledge may be regarded as a plan for restricting the number of alternate moves that must be evaluated in the search for a solution. An algorithm is a well-defined procedure for solving a class of equivalent problems. It defines a single, unique sequence of moves for solving any problem within the class. By comparison, a strategy is any well-defined procedure that narrows down the set of possible, alternative moves, without specifying a single, successful path to the solution. In contrast, a heuristic is a less precise rule-of-thumb that may suggest a strategy or algorithm

appropriate in a particular situation (Polya, 1974). Heuristics are considered to be less domain-specific than algorithms or strategies and can be employed in solving problems in different domains. Whatever the domain, planning knowledge eliminates blind search, the trial and error testing of every contending solution in the problem space.

Examination of the behaviors and knowledge involved in problem solving suggests a reconciliation of the psychologist and the educator's definitions of problem solving. Human problem solving can be viewed as a process of search constrained by a perceptual and propositional understanding of the domain in which the problem is set. Successful problem solving is determined by the degree to which this domain-specific solution information is transferred from previously-solved, related problems.

Learning Through Interactive Programming

The advent of microprocessor technology has revolutionized the design of computers, making them smaller, more portable, and less expensive. As a result, affordable, personal microcomputers have flooded the mass market, permitting private individuals as well as public schools to purchase these machines in vast quantities. The touting of "computer literacy" by popular periodicals has created a bandwagon effect in which the purchase of

microcomputer equipment by administrators and boards of education often precedes serious consideration of its instructional use. Consequently, the explosive appearance of microcomputers in the schools has provoked questions about the application of this new technology.

Schools have used microcomputers largely for computer-assisted instruction (CAI). CAI is an attempt to automate traditional methods of instruction such as drill or practice exercises, tutorials, and, occasionally, simulations and laboratory applications. Small microcomputer memories, the lack of quality instructional software, and poor teacher training have restricted the promise of computer assisted instruction. These limitations are being overcome. However, even as CAI courseware improves and microcomputer memories expand, some authorities question whether CAI represents education's best use of the technology.

A recent competitor to CAI's domination of instructional computing is what DeLaurenties (1980) called "Learning through Interactive Programming" (LIP). LIP advocates, such as Seymour Papert and other members of the LOGO group from the Massachusett's Institute of Technology, regard CAI's "linear mix of old instructional methods with new technologies" as an anachronism, a conservative and transitional stage in the childhood of instructional computing (Papert, 1980, p. 36). LIP proponents wish

to minimize computer-assisted programmed learning and develop computer programming activities as the vehicle for teaching, developing, and exercising children's logical, mathematical, and problem-solving skills (Feurzig & Nickerson, 1981). Unlike most high school and much college programming instruction, LIP emphasizes the process of problem solving inherent in programming rather than simply the acquisition of a programming language. Unlike CAI, the child interacts with the computer to learn the skills of problem solving, rather than specific information related to traditional academic subjects. The computer, according to this view, is most appropriately used as an "object to think with" rather than a teaching machine or the subject of study.

Wells (1981) suggested that the processes involved in computer programming are similar to the planning knowledge generally used in problem solving. In fact, she proposed that programming may be a "better catalyst for invoking problem-solving skills than the use of traditional problems" (p. 109). For Wells, Papert, and other LIP advocates, the problem solving skills acquired through programming are heuristics. Often, heuristics are cast in the form of mental operations: analogy, looking backwards, varying the problem, or induction. For example, the heuristic, "varying the problem," requires that the problem solver mentally decompose and recombine the elements of a

problem space (Polya, 1974). Sometimes, heuristics are presented as questions with which to interrogate the problem situation: "What is the unknown?" "Do I know the solution to a related problem?" "Can I work backwards?" or "Can I subdivide this problem into smaller problems?" (McClintock, 1979; Papert, 1980).

Statement of the Problem

Heuristics are said to be "domain independent" planning knowledge in the sense that they are independent of subject-matter and can be used to solve a wide variety of problems. It is tempting for educators to suppose that these common problem-solving techniques can be learned in a domain, such as computer programming, and then be applied to solving problems in another related domain such as mathematics or logical discourse. While Papert (1980) and others have held out that promise, the extent to which problem-solving techniques learned in one domain may be useful in solving problems in another is an open question.

Successful problem solving may be more dependent on domain-specific knowledge and skills than was formerly supposed by earlier investigators such as Polya (1974). Recently, researchers (Bhaskar & Simon, 1977; Greeno, 1980; Simon, 1981) argued that domain-specific perceptual behavior and propositional knowledge may be indispensible to a person's recognition, interpretation, and

representation of the problem situation. They suggested that the effectiveness of planning knowledge, including heuristics, may depend upon how well the problem solver understands the subject area, or domain, in which the problem is set. If this is the case, instructional techniques that assist learners in acquiring a meaningful understanding of the problem domain may translate into improved problem-solving performance within, but not outside, that domain.

The purpose of this study was to determine the effect that participation in a sequence of computer programming activities had upon the problem-solving performance of four groups of fifth and sixth graders in the domain of nonroutine, mathematical problem solving. The researcher also examined the effect of two instructional strategies upon the students' ability to solve computer programming problems. One strategy consisted of the use of a manipulative model to represent certain features of the BASIC programming language. The other strategy was a set of perceptual training techniques, called "inspection methods," which were derived from artificial intelligence theories of engineering problem solving.

Need for the Study

In this study, the researcher addressed the needs outlined in the following section.

LIP claims. Despite the assertions of its proponents, there is a lack of knowledge concerning the educational effects and outcomes of LIP activities. One of the most important and controversial assertions is that problem-solving skills learned through computer programming may result in improved problem-solving performance in other areas. Research on this question largely has been limited to pilot studies of various structured languages such as LOGO. This was the case in the Brookline Study, (Papert 1979; Watt, 1979) which has become the empirical cornerstone of the LOGO Group's arguments. This researcher sought to redress this deficiency by designing an experiment to test the degree to which problem-solving skills learned through computer programming transferred to a closely related domain: mathematical word problems of a nonroutine nature.

Methods of teaching computer programming. Questions concerning the most effective methods of teaching computer programming languages to children have inspired a great deal of speculation but very little research. Studies of techniques for teaching programming languages, such as McEntyre (1978), Owen (1978), and Mayer (1981), used adult populations in their experiments. Although their conclusions may have some bearing on programming instruction for younger subjects, developmental differences among adults and children caution against overgeneralizations. Because

this study used fifth and sixth grade students, it should provide more accurate guidance to those who design computer programming instruction for children.

Conceptually complex computer languages. A perennial question is concerned with which computer language provides the "best" introduction to programming. Much has been made of the beneficial effects of teaching structured programming languages like LOGO to children. Papert (1980) argued that LOGO's structured and procedural character qualifies it as the most appropriate "first language" for children. The LOGO used in most elementary schools is "informal LOGO," a simplied version of the original. "Informal LOGO" may be easier to teach to children than the PASCAL or BASIC programming languages, but the effects arising from its simplicity should not be confused with the effects arising from a particular method of programming instruction.

"Informal LOGO" is not a conceptually complex language. It does not provide the richness of commands and control structures present in BASIC, PASCAL, or even the full-blown LOGO programming language. Furthermore, "informal LOGO" employs the turtlegraphics system that provides immediate, graphic feedback for every command the user employs. Because of the simplifying and making concrete of many of the conceptually complex and abstract features that are part of most programming languages, informal LOGO might result in reduced needs for the

instructional techniques assessed in this study. Nevertheless, du Boulay, O'Shea and Monk (1981) found the use of "notational machines," instructional models similar to those used in this study, to be helpful in teaching even LOGO-like languages. Even if these techniques prove unnecessary, the choice of LOGO as a first language will not eliminate the need of many students to learn a second, more conceptually complex language. The instructional techniques outlined in this study might prove helpful in that task.

Modeling and instruction. The instructional use of pictorial models was studied by Owen (1978) and Mayer (1981) to determine if such use is helpful to novices in understanding the BASIC programming language in a more meaningful way than usually occurs as a result of purely expository methods of instruction. A better understanding of the programming language should result in more flexible and efficient problem-solving performance. Although Mayer found some beneficial effects of model presence upon his subject's problem-solving performance, Owen found very few. However, this study differed from the Mayer and Owen studies in at least three ways. First, Mayer and Owen used a pictorial model. In two experimental treatments in this study, the researcher employed a manipulative model that students used directly with program text. Second, these treatments were longer and more intense than those

that occurred in the Meyer and Owen studies. Third, Mayer and Owen used adult subjects, grade school subjects were used in this study. It was conjectured that grade school children have fewer and less efficient "schemas" for organizing computer program texts than do adults. The young students involved in this study, therefore, should have derived greater benefit from a model than did the adult subjects in Owen's study.

Perception and instruction. Learning theorists have gained a greater appreciation of the role of perceptual behavior in skilled problem solving. Investigators of expert performance in domains as diverse as chess and electronic circuit design have noted what Schoenfeld (1980) calls "automatic expert observation," the uncanny ability to instantly "see" the structure of a problem and provide a solution as if "without thought." With experience and learning, experts acquire the ability to recognize complex patterns, whether those patterns are embodied in chess games, electronic circuits, bridge hands, or computer programs. Greeno (1980) suggested that children's performance in solving arithmetic word problems might be improved by training them to attend to the relevant patterns of information present in the problem statements. Hence the question: could the perceptual skills of expert computer programmers be taught directly? Surely there is no shortcut to the expertise that is laboriously acquired

through reading, debugging, and composing a great many programs; however, a few, perhaps largely preliminary, skills might be taught to novices by means of perceptual training. In this study, a set of perceptual training techniques, called "inspection methods," was employed to test this supposition.

Presentation of the Study

In Chapter II of this study is a review of the literature relating to human problem solving and computer programming. An initial exploration of this literature and this researcher's own experiences in teaching computer programming provided the general outlines of the problem that was used to guide this study. The problem was precisely delineated by stating a series of hypotheses (nulls and alternatives) in Chapter III. To test these hypotheses, two experiments were conducted with 139 fifth and sixth graders from Jackson Intermediate School (Jackson, Wyoming) during its winter quarter, 1984. The results of these experiments are summarized in Chapter IV and discussed in Chapter V. In addition, a series of "think-aloud" protocols and clinical interviews was taken in order to clarify and extend the psychometric analyses. These results also are discussed in Chapter V.

Summary

Cognitive psychologists and educators have viewed human problem solving differently. These differences, coupled with the research paradigm that dominates cognitive psychology, have limited the instructional influence of recent research in the area of human problem solving. An examination of the various definitions of problem solving reveals that these differences are reconcilable and that the design of computer programming instruction could benefit from the insights provided by psychological research.

Reading, composing, and debugging computer programs can be viewed as problem solving in a "conceptually complex" domain. This complexity is reflected not only in the structure of computer programs but also in the kinds of knowledge employed in solving programming problems. Because computer programming encompasses a variety of problem-solving behaviors, some educators have advocated it as a vehicle for teaching general problem-solving skills.

One purpose of this study was to determine the effectiveness of computer programming as a vehicle for teaching general problem-solving strategies and to evaluate the degree to which strategies, learned through programming activities, improve students' ability to solve nonroutine, mathematics problems. Another purpose was to examine the effect of two instructional strategies that were designed

to enhance students' understanding of the BASIC programming language. These goals were related directly to wider empirical questions concerning the role of computer programming as a medium for teaching domain-independent, problem-solving strategies, the power of such strategies, and the effect of perceptual training and modeling techniques on the development of domain-dependent, problem-solving skills.

Chapter II.

REVIEW OF THE LITERATURE

The purpose of this study was to determine if two instructional methods, designed to improve understanding of the BASIC programming language, resulted in improved performance in solving programming problems. Research workers in fields as diverse as linguistics and artificial intelligence have produced ideas that help explain the problem-solving processes involved in computer programming. Their research also provides clues to instructional techniques for developing in children and adults an understanding of computer programming languages.

In this chapter, the research, related to understanding programming languages and developing programming skills, was approached from two directions. The chapter begins with an examination of the psychology of understanding and its relationship to problem solving. This is followed by a definition of computer programming as a set of problem-solving skills. The path then divides. The first path leads through the recently researched distinctions between expert and novice problem-solving behaviors. These studies provided clues to how experienced computer programmers understand and compose programs.

Studies of expertise also provided ideas for teaching expert-like problem-solving behaviors to novice programmers. A second path winds through studies of text comprehension, natural language programming, and program complexity analysis. Reviewing these studies clarified the inherent difficulties that the form and content of programming languages pose for novice programmers. Examining this literature also suggested ways in which instructional aids, such as manipulative models, can be used to assist beginning programmers to understand program text and to deal with program complexity.

Understanding and Problem Solving

Problem-solving depends upon an "understanding" of the conceptual relations of the domain in which the problem is set. "Understanding," however, lacks clear definition, although it gains clarity when contrasted with its opposite, "rote recall." This distinction between rote recall and understanding was a salient feature of Gestalt analyses of problem-solving (Wertheimer, 1959). Understanding also was an important category in the work of William Brownell (1945, 1946, 1956), who formulated the distinction in terms of "skill vs. understanding." Brownell defined understanding in terms of successful application of knowledge to situations of increasing dissimilarity or complexity to those in which the knowledge

originally was acquired. Brownell's view of understanding resembled the traditional definition of problem-solving as transfer of learning.

In recent years, analogies from artificial intelligence research have begun to flow into the older psychology of meaning represented by William Brownell to give greater analytical clarity to the term, understanding. Cognitive psychologists, such as James Greeno (1977, 1980, 1983), suggested that understanding in problem-solving is a "constructive process" in which the solver operates on the problem information to create a coherent, internal network of relationships that corresponds to the problem solution. This internal network of relationships is known as a "representation." By providing abstract versions of the relationships among the elements within the problem space, representations guide planning and reduce search of the problem space by permitting direct retrieval of relevant subgoals and by triggering relevant problem solving strategies.

Representations are the internal, mental structures that result from encoding or reorganizing the external environment. Representations may consist of mental "data structures," images, propositions or other verbal behavior, subvocalizations or nonverbal responses. In the behaviorist accounts, representations were subsumed under the category of "private behavior," stimuli and responses

inferred but not directly observable by a verbal community (Catania, 1979). Representations, according to many cognitive psychologists, are internal knowledge structures that can be described in terms of certain theoretical formalisms. In these formalisms are included the "schemata," "schemas," and "semantic networks" of language theory, the "production systems" and "procedural networks" of problem-solving theory, and the "chunks" and "plans" of computer science.

Researchers in the field of computer vision proposed that understanding of a problem results in a set of varied and redundant representations, arranged in a hierarchy of increasing abstraction (Brown, 1984). Larkin and Rainard (1984) provided an interesting model of this process of building multiple, internal representations during problem-solving. In a similar vein, Lochead (1981) argued that understanding is being able to represent a concept or problem in more than one way and to move flexibly among these representations. The notion of flexible representations was echoed in Stevens and Collin's conjecture (1980) that experts employ multiple "mental models" of single situations or problems and can "map" back and forth among models in arriving at solutions. Problem-solving skill, therefore, may partially consist of the ability to construct and coordinate multiple

representations in ways that provide direct access to effective strategies.

Procedural Specification

Computer programming belongs to a set of problem-solving tasks called procedural specification. These tasks require that a "sequence of actions be specified in some language such that, when executed by a designated agent, a particular goal can be accomplished" (Miller, 1981, p. 22). In specifying procedures, people use language to structure information and to produce algorithms. An algorithm is a plan for solving a specific class of problems. Within an algorithm is included definitions of an initial (problem) state to which the algorithm applies, a goal (solution) state, and a set of regular operations for transforming the initial state into the goal state. Procedural specification is the dominant feature of certain natural language texts, such as kitchen recipes and repair manuals, just as procedural specification is the dominate feature in computer programs. The fact that procedures can be specified in natural or artificial languages and can be executed by humans or machines suggests certain common problem elements and operators.

Computer programming consists of three phases: comprehension, composition, and debugging. Comprehension

actually encompasses two quite different problem-solving behaviors: understanding computer program specifications and understanding computer program text. Only one phase, the composition of programs, can be regarded as procedural specification. The skills involved in program comprehension and debugging are more akin to those involved in understanding sentences and editing manuscripts.

Greeno (1978) created a typology of problems and skills involved in solving those problems. He argued that most problems are mixtures of three ideal problem types: problems of inducing structure, problems of transformation, and problems of arrangement. This typology can be used to understand the three phases of computer programming. Understanding programs and program specifications are akin to problems of inducing structure. Composing and debugging programs resemble problems of arrangement and transformation.

Program comprehension requires the induction of structures, or patterns of relations, among sets of propositions within a program. These propositions consist of individual commands and reserve words of the programming language. As in analogy problems or series extrapolation problems, the programmer must identify the relations among the component propositions of a program text and fit those relations together into a pattern. It is this pattern, rather than the individual commands and reserved words,

that provides the semantics, or meaning, of the text. The induction of structure in programming text requires not only an understanding of the meaning of individual propositions, but also apprehending those patterns that represent lawful relations among component propositions. These patterns may take two forms: "chunks" and "program text schemas." Program text schemas are the general ways programs can be represented. Chunks are very specific representations based on functional groupings of programming code. Both will be discussed later in this chapter.

A program specification is a problem statement: it describes what the completed program should do. Although program specifications are stated in natural rather than programming languages, understanding them requires the same kinds of skills as understanding programs written in computer languages: the component propositions of the problem statement must be identified and then fitted into a meaningful pattern.

What are the meaningful patterns within program specifications? Reference to skilled interpretation of problem statements in other domains is suggestive. Hinsley, Hayes, and Simon (1977) found that high school and college students acquire the ability to classify algebra word problems into categories even before the students finish reading the problem statements. Acquired through

extensive experience with a wide variety of word problems, these categories provide useful planning knowledge (relevant subgoals, solution procedures, equations, diagrams etc.), information that was absent or only implied in the problem statement. Larkin and her associates (1980) found that the recognition of patterns, or "schemas," was an important skill to build into artificial intelligence programs designed to solve statics problems written in English. These investigators also produced empirical evidence that human problem-solvers use these same patterns in solving physics problems. Riley, Greeno, and Heller (1983; Greeno, 1980) observed that grade school students, who performed well in solving arithmetic word problems, acquired a skill in recognizing patterns, "problem type schemas," that enabled them to recognize and organize the information in a problem.

There is no conclusive evidence that programmers use problem type schemas to understand program specifications; however, there is evidence from artifical intelligence research (Rich et. al., 1978; Rich, 1981) and from a study that used think-aloud protocols (Wells, 1981) that some programmers do. Persuasive corroboration of problem type schema came from a study by Weiser and Shertz (1983), who found that expert programmers classified problem specifications according to the algorithm type needed to solve the problem (sort, search etc.). In contrast, novice

programmers focused on the surface features of programming problem statements. Ignoring classification by common algorithms, the novices sorted problems according to superficial similarities such as application area (operating systems, business etc.). A characteristic of expertise in computer programming, therefore, may be the ability to quickly match patterns of problem statements with algorithms that are appropriate for the problem solutions.

Another phase of computer programming is writing programs. The composition of computer programs resembles arrangement problems. In arrangement problems, the task involves producing some combination of given elements that meets some specified criteria. The elements used in composing programs consist of the vocabulary of the programming language, the criteria are found in the program specification, and the correct combination of elements result in a completed program. Unlike other arrangement problems, such as those found in chess, programming problems are "ill-structured." Ill-structured problems initially lack the elements necessary for a solution (Simon, 1973; Greeno, 1976; Greeno, Chailkin, Magone, 1979). Program specifications often are ambiguous and incomplete. Subgoals are left unstated or implicit and must be generated by the programmer, thereby allowing some goals to be achieved by alternate combinations of elements

(Gorgono & Nelson, 1982). Composing programs from specifications, therefore, entails more than the simple translation of English into programming code.

Because of the poverty and ill-structured character of the initial problem space, a creative effort is required when composing programs: the programmer must "enrich the problem-space" through the generation of subgoals and programming structures. Greeno described this process of planning the solution of ill-structured arrangement problems as "constructive search." (Greeno, 1978; Larkin, Heller, Greeno, 1980). Constructive search consists of generating partial solutions and testing their consequences against subgoals. Fluency in generating appropriate, partial solutions is an important skill in constructive search. For expert programmers, these partial solutions consist of well-organized "chunks" of programming statements that are generated almost automatically to achieve subgoals. This fluency is similar to the emphasis in solving transformation problems upon the selection of operators that will transform the initial state into the goal state (Newell & Simon, 1972; Sweller, 1983).

Interpreting programs and program specifications or engaging in constructive search depend upon understanding, upon specific representations of the programming structures that are supported by a given programming language. In standard programming practice, these representations

consist of functional groupings of code known as "chunks." For novices, interpreting or composing a computer program usually involves the laborious reinvention of these shopworn tools of the programmer's trade. These stereotypic programming structures are the basis for efficient planning of solutions because such structures are powerful constraints on what otherwise would be blind trial and error search through the problem space. As we shall see, expert programmers have a wide repertoire of these programming cliches and are capable of using them in an algorithmic fashion to generate solutions directly.

Novice-Expert Distinctions

A substantial body of research has grown up around the question of individual differences in problem-solving. Experts solve complex problems faster and more accurately than novices. Cognitive psychologists have ascribed expert performance to more powerful representations of the problem (Larkin, McDermott, Simon & Simon, 1980; Larkin & Rainard, 1984). These representations have been variously described as "schemas," "production systems," "procedural networks," or "chunks." In all cases, they are based on extensive knowledge of the problem domain.

The superior "representational ability" attributed to experts may account for differences in problem-solving style. Glaser (1980) proposed that novices, less able to

coherently represent problems, are more dependent upon extensive search of the problem space, planning, and use of solution strategies and heuristics. Novices devise strategies, replete with goals and subgoals, and laboriously follow them step by step until a solution is found or a new strategy is required. Experts, capable of powerful representations, can "automatically" produce well-organized, partial solutions that need only be fine-tuned to meet solution criteria. The contrast between expert and novice problem-solving behavior provides some insight into methods for teaching novices to become accomplished problem-solvers in the domain of computer programming. Although there is some theoretical work on expert performance in computer programming, the initial research is exploratory and somewhat derivative of work in other areas.

The literature about expert problem-solving in chess, physics, Gō, algebra, and other areas was well summarized in Greeno (1978), Greeno et. al (1980), and Adelson (1981). Problem-solving in the research accounts has been presented as behavior consisting of perceptual skills, actions and strategies, and factual knowledge of the problem domain. Much of the concern has focused on determining the functional unit of this behavior. Functional units are organized patterns of stimuli and responses having a significant degree of internal structure (Catania, 1979;

Larkin, 1980; Shimp, 1976). Analysis of these functional units, or "chunks," has acquired prominence in recent learning theories. In their groundbreaking study of the skills of expert chess players, Chase and Simon (1973) discovered that the skillful play of experts derives, not from superior memory capacity, but from a learned ability to recognize familiar patterns, or "chunks," of chess positions. They also found that master chess players could recall more chunks than novices and that the experts' chunks were larger and better organized, most often in hierarchial structures. Moreover, masters seemed more adept at quickly selecting and applying appropriate chunks to problem solutions.

Other researchers found similarities between the functional units of chess and those of other areas of expertise. Reitman (1976) discovered that master Gō players also "chunked" familiar gameboard configurations. Egan and Schwartz (1979) found that the unit of recall for skilled electronics technicians was functional chunks of circuit diagrams. Skinner (1980) observed an analogous skill in the memory for bridge hands exhibited by professional card players. He noted the tendency of complex behavior, originally acquired through a functional chaining of operant responses, to be emitted as a unity - "without thought" - in response to certain combinations of stimuli. For Chase and Simon's chess players, Reitman's Gō

masters, Egan and Swartz's electronic technicians or Skinner's bridge players, the functional units of behavior were large patterns of moves, piece positions, circuits, or bridge hands. In these instances, expert problem-solving behaviors were structured according to the functional principles of the area of expertise.

Expert problem-solving behavior is acquired through repeated experiences with familiar stimuli. A learner's behavior with respect to the stimulus to be remembered is called "encoding." "Chunking" (or substitution) is a special kind of semantic encoding in which the number of items to be remembered is reduced by organizing them in special ways (Catania, 1979). Chunking consists of building larger perceptual or behavioral units, and of deleting unessential or redundant parts, thereby simplifying recognition and execution. A "chunk" may be a pattern of stimuli which has become familiar from previous repeated exposure and hence has become recognizable as a single unit during problem-solving episodes, such as interpreting natural language texts or computer programs (Simon, 1980). Similarly, a set of responses can be "chunked" during learning and, hence, can be produced as a single unit in tasks such as writing computer programs. Problem-solving strategies and algorithmic procedures are particularly susceptible to chunking (Goldin, 1979; Neches & Hayes, 1978). Groups of operations or single steps in a

procedure, which initially are executed separately, can become grouped into a single complex act, and, thereafter, can be executed as one behavioral unit.

Simon (1980) proposed that chunks are stored as perceptual patterns in long-term memory. This perceptual knowledge "indexes" propositional knowledge about the area of expertise to problem-solving strategies and other planning knowledge. Recognition of a familiar pattern in a problem context elicits chunked behavioral routines that were successful in solving previously encountered problems in which similar patterns were present. Recalling a single chunk, rather than recalling its components separately, improves the efficiency of performance and increases the ease or recall and reconstruction of similar responses. In solving problems, therefore, pattern recognition may be linked to chunked responses in "condition-action pairs" that are the basis for what cognitive psychologists call "production systems" (Simon, 1980).

In cognitive psychology, production systems are mental representations consisting of actions (procedural chunks) paired with conditions (perceptual chunks) specifying when the action is to be taken. Several recent studies provide some evidence that skilled computer programming may be based on a large number of production systems. In understanding, composing, and debugging programs, computer programmers may recognize familiar groups (chunks) of

statements within programs and mentally recode these into a loose hierarchy of ever larger functional units. These patterns of code in turn serve as solution methods for classes of problems and problem statements encountered during programming. The condition-action pairs of programming expertise are perceptual chunks linked to chunked solution methods. Programming production systems are acquired through extensive programming experiences and probably are the basis for skilled performance in understanding, composing, and debugging computer programs.

Much of the earliest research in this area was derived from Sach's (1967) study of the syntactic and semantic features of memory for English sentences. This early research used "memorization-reconstruction" tests to assess the structure of programmers' knowledge. These tests required the subjects to memorize a program within some time interval (usually 10 to 15 minutes) and then to reconstruct it from memory. Performance was evaluated according to the number of lines literally recalled or the amount of "high-level logic" in figuratively reconstructed code (see Brooks, 1980).

Using memorization-reconstruction tests, Schneiderman (1976, 1977, 1979) and Di Persio et. al. (1980) investigated the extent to which experienced and novice programmers recognized and remembered the general semantic features of computer programs. Schneiderman (1980), for

example, reported a study that compared the performance of experienced and inexperienced programmers in memorizing a coherent computer program and a program consisting of random lines of code. The experienced programmers reconstructed many more lines of the coherent programs than did the inexperienced programmers, but performance dropped to equivalent levels when both groups attempted to reconstruct the less meaningful, randomly organized program. A similar experiment by McKeithen (1979) produced equivalent results.

The authors of these early studies sketched the broad outlines of expertise in computer programming, but did not delineate the specific structures of expert behavior and memory. They showed a connection between the ability to understand programs and to remember programs. Furthermore, they demonstrated that experts were able to grasp the semantics, or meaning, of program text and were able to use this knowledge to accurately reconstruct programs. Specific evidence of the chunking phenomena in computer programming came from research using different kinds of measures.

Three recent studies, using data from multitrial, free recall activities and data from recognition tasks, provided insight into chunk organization and its relationship to levels of programming experience. In a study of 22

subjects selected from three experience levels (naive, novice, and experienced), McKeithen (1979) was able to infer her subjects' "memory structures" (nested chunks that could be represented as tree graphs) from the order of their recall of ALGOL reserved words.

Using a more sophisticated analysis, Adelson (1981) compared the response times and recall orders of five experienced and five novice programmers. Her subjects observed and then recalled randomly presented lines from three procedurally and functionally similar programs. Adelson corroborated McKeithen's observation that the experienced programmers recalled more chunks than did the novices. Both researchers found the experts' chunks to be larger and better organized than those recalled by the novices. In addition, Adelson found that novices tended to recall one line segments which were weakly organized into syntactic categories. However, the more experienced programmers produced more complex chunks, encompassing whole routines and consisting of elements organized according to procedural (semantic), rather than syntactic, similarity.

In a study of the functional groupings of code to which programmers attend in debugging tasks, Weiser (1982) had twenty-one experienced programmers debug three ALGOL programs and then engage in a recognition task involving "slices" of code that were varied according to their

proximity to the program bugs. He found that, when tracing backwards from a variable in search of a bug in an unfamiliar program, experienced programmers "slice" away code without influence on the value of the variable and concentrate on chunk-like program segments that are organized according to data and control flow. Although somewhat exploratory, the work of McKeithen, Adelson, and Weiser supported the notion that skilled computer programmers interpret, compose, and debug programs in terms of large, interlocking chunks of code.

Schneiderman (1980) and Mayer (1975, 1976, 1979, 1981) have proposed a "syntactic/semantic" model of programming behavior partially based on the chunking phenomena. Both researchers distinguished the syntactic from the semantic aspects of programming languages. Syntactic knowledge is language-dependent, consisting of low-level details such as the proper forms for assignment and looping. Syntactic knowledge is acquired by memorization, maintained by frequent rehearsal, and poorly integrated into memory. In contrast, the semantics of a programming language consists of "general programming concepts that are independent of specific programming languages" (Schneiderman & Mayer, 1979, p. 222). These general programming concepts are embodied in specific groupings or chunks of programming statements. When examining, modifying or debugging a program, experienced programmers "chunk" groups of

statements and then use their semantic knowledge about the function of these chunks to construct multileveled, internal representations of progamming problems. Program knowledge thereby is organized in a hierarchy, placing the syntactic details in the lowest level, ascending in a chunkwise fashion to ever higher levels of abstraction, and culminating in an overall sense of program purpose and function.

In the semantic/syntactic model, learning a first programming language requires acquisition of both the specific syntax of the language and the semantic constructs that are common to the family of languages of which it is a member (Schneiderman, 1980). Learning a second language in the same family would only require learning a new set of syntactical details since the semantic structures would have been learned previously. Thus, after mastering LISP, learning LOGO would be easy because of semantic similarity, but attempting to learn PASCAL would be much more difficult because a new set of semantic structures would have to be mastered.

A number of British researchers called into question the semantic/syntactic model by asserting that there can be no clean separation of the syntactic from the semantic elements of programming languages. Using ALGOL-like microlanguages, they explored the cognitive effects of manipulating "surface" syntactic features such as

conditional forms (Green, 1977; Sime et al., 1977), nesting vs. branching-to-label constructions (Green, 1980; Sime et al., 1973) and scope markers (Arblaster, 1982; Sime et al., 1973). Drawing on these studies, Arblaster (1982) suggested that there is considerable interaction between the pragmatic, syntactic, semantic, and lexical levels of a programming language: "Small changes in the appearance of a language can make large changes in a person's understanding of that language and in his expectation of the meaning of a particular syntactic construction" (p. 215).

In a caustic review, Sheil (1981) took issue with the British and other studies of programming notation, citing weak experimental methods and small treatment effects. While these syntactic features may be varied alternately to confound or to assist the novice, she argued, the salient variables of programming performance lie not in differences in programming practices or notation but rather in the learned special-purpose skills that constitute expertise in computer programming. These special-purpose skills may be described in terms of theoretical formalisms, such as production systems, chunks, and schemas.

Barriers to Understanding Progamming Languages

Many novices encounter barriers in their efforts to acquire expertise in a programming language. This researcher found that children have difficulty in acquiring

anything more than a rote grasp of conceptually complex programming languages exemplified by BASIC or PASCAL. Paradoxically, these same children experience fewer problems mastering LOGO. "Informal" LOGO's small vocabulary may explain the ease with which it is learned. The conceptual complexity of any language, however, derives as much from language structure, the permissible connections among individual words, as from vocabulary size. Miller (1981) suggested that programming languages are difficult for novices because they have different structural requirements than those of English. Miller's supposition was bolstered by recent research of reading comprehension, specifically "schema theory." "Schemas," or "schemata," are representations of the content or structure of certain kinds of written or spoken language. An important implication of schema theory is that difficulty in understanding hitherto unfamiliar texts, such as computer programs, derives from a mismatch between the structure of program text and the reader's expectations and goals established by prior experiences with somewhat different text structures.

Propositional Structures

The structure of both natural language texts and computer programs consists of connected lists of propositions (Thorndyke, 1977). Propositions are composed

of a relational term (an operator) and one or more arguments or details (Kintsch, 1977a) A proposition is a distillation of the abstract relations among the individual words of a sentence and constitutes the molecular level of meaning in texts. Sachs (1967) found that adults, after reading a passage, are more likely to remember propositions and the abstract relations among the propositions than the individual words or sentences. Later work by researchers (Kintsch, 1974; Anderson, 1983) bolstered Sach's findings: propositions, rather than individual words, are the basic units of recall for natural text.

Atwood and Ramsey (1978) proposed that computer programs also are constructed from propositions rather than the individual keywords of the programming language. Using LISP-like notation, they constructed a taxonomy of propositions from the keywords of the FORTRAN programming language. Their propositional notation can be easily translated into BASIC. For example, the assignment statement, "N = 20," would have the propositional form, SET (N,20), where SET is the relational term (operator) denoting assignment and (N,20) is an argument that assigns the value of 20 to the variable N. Although both natural language texts and computer programs are composed of propositions, these propositions appear to be connected and structured in different ways.

In order to be understood, the propositions of a text must be connected and organized into a coherent structure. "Textual coherence" is established either through internal reference of one proposition to another or through external reference of a proposition to "real" or "possible world" phenomena (Fredrickson, 1977). In natural language texts, internal coherence is based upon concept repetition, temporal ordering, casual connecting, or conditioning (Anderson, 1974, 1983; Fredrickson, 1977; Hilgard & Bower, 1982; Townsend, 1983). Through internal coherence, propositions are interpreted not in an absolute way but in relation to one another.

Internal coherence by itself does not make a text meaningful. Consider the sentence, "The seams split, and the notes went sour." It possesses strong internal coherence but is meaningless without an external reference to a musical instrument, the bagpipe. External reference is based upon information about the physical entities or social events being described. Similarly, the writer's assumptions about the reader's knowledge create a "pragmatic context" that involves a real or imagined writer (speaker) and reader (listener) (Clark, 1978; Winograd, 1977). Contextual referencing supplements and often supplants internal propositional coherence in natural language texts.

Text Comprehension Strategies

Text comprehension has been described as a problem of inducing structures or patterns of relationships among sets of propositions (Greeno, 1978). In this view, understanding is achieved by inducing some overall pattern, or "propositional macrostructure," from the text (Kintsch, 1977b). Macrostructures are derived by means of two, complementary information-processing strategies: "bottom-up" or "top-down" decoding. When confronted with difficult or unfamiliar text, most readers use a "bottom-up" or "data-driven" strategy. "Bottom-up" strategies involve a careful word-by-word or phrase-by-phrase decoding of the text (Kintsch, 1977; van Dijk, 1977). When the text deals with familiar topics or possesses a familiar structure, many readers employ a "top-down" or "conceptually-driven" strategy based upon what they expect the text to say. These expectations are drawn from their experiences with similar texts and upon their general knowledge of the world (Anderson, 1978). Inducing a structure from text through "top-down" processing is a kind of hypothesis testing strategy in which inferences about the text are frequently, but not continuously, checked during reading.

The goals, expectations and prior knowledge about text structure or content that readers employ in "top-down" information-processing have been described as "schemata" or

"schemas." The schema notion has been used to characterize both the underlying organization of the written material and the reader's mental representation of the class of text types to which the written material belongs (Brooks & Dansereau, 1983). Children's stories and other narratives display conventional patterns of organization, "story schemas," that condition sets of expectations about the kinds of units found in narratives and about the way these units are organized (Mandler & Deforest, 1979; Mandler & Johnson, 1977; Taylor, 1982). Expository materials such as textbooks have their own characteristic formats and structures that pose difficulties for young readers who possess only the rudimentary story schemas acquired through their experiences with children's literature (Brooks & Dansereau, 1983; Englert and Hiebert, 1984). Schemas for expository texts are similar to the problem-type schemas used in decoding counting, arithmetic, and physics problems (Greeno, Riley & Gelman, 1984; Hinsley, Hayes & Simon, 1977; Riley, Greeno & Heller, 1983). Similarly, the meaning of events and situations described in texts has been ascribed to the existence of "event schemas" or "scripts." A script is a "predetermined, stereotyped sequence of actions that defies a well-known situation." (Schank & Abelson, 1977, p. 41).

In cognitive theories, schemas and scripts aid recognizing, understanding, and remembering textual

materials by provding familiar categories, or "slots" to which propositions can be assigned and from which they can be readily retrieved. Schemas function as strategies for encoding and retrieving macropropositions. As such, schemas vary in specificity. Brooks & Dansereau (1983) made a useful distinction between content schema and form schema. Content schemas involve very concrete and specific knowledge of information found in a class of texts. Form schemas involve general and more abstract knowledge of the format and organization of particular kinds of text types such as exposition, narration, or procedural specification.

Schemas in Natural and Programming Languages

There is some evidence that propositional coherence is established differently in programming languages than in natural languages and that these differences can place strenuous demands on naive and novice programmers. Procedural specification in computer programs places different contextual and textual constraints on the comprehension process than does procedural specification in natural language texts such as recipes or repair manuals. In a summary of studies comparing procedural specification in natural and computer languages, Miller (1981) concluded that these differences are so great that the "direct translation of natural-language programs into formal

computer languages may be feasible only for rather simple programs." (p. 25)

Coherence in natural language texts is partially established through contextual referencing. Through contextual referencing, a writer is able to make certain assumptions about his readers' schematic knowledge. Computer languages, in contrast, demand a level of "absolute" referencing not found in natural languages. Computer interpreters and compilers require that individual propositions be directly mapped to a fixed set of commands as specified by a small grammar of context-free rules (Reisbeck & Schank, 1978). This absolute referencing frequently is a problem for naive programmers. In studies of natural language programming by non-programmers, Miller (1981) observed that his subjects wrote code contextually, as if it were intended for people rather than for computers. Since the cohesion among propositions in natural languages is so dependent upon the experiences and information shared by writer and reader, the naive programmer often expects the computer to operate as if it were sensitive to the pragmatic context, inferring the programmer's desires and producing a configuration of complex outputs when only one simple command was typed.

Providing a "referential field," a domain that simulates some of the pragmatic context of natural languages, might give novices an easier entry into

programming languages. Moeser's work (1975a, 1975b, 1976) illustrates the positive effect that the presence of an iconic reference field has upon the acquisition of minature artificial languages. Referential fields in the form of random arrays of differently shaped blocks ("blocks worlds") also proved important in efforts by artificial intelligence researchers to develop computer programs that learn and solve problems (Walz, 1982). In fact, the ease with which even young children learn to use Seymour Papert's (1980) Turtlegraphics system probably derives from the manner in which "informal Logo" maps a limited set of simple propositions to the iconic "microworld" of the turtle.

The absence of pragmatic context contributes to the initially alien quality of programming languages; however, textual differences present even more critical obstacles. In programming languages, propositions are organized differently than they are in natural languages. Consider conditioning, the use of conditional forms, such as IF-THENs, REPEAT-UNTILs, and DO-WHILEs, to specify the conditions under which actions are to be performed. Conditioning is the single most important way of binding together computer programs. In both natural and programming languages, conditional propositions, those employing conditional IF...THEN's, OR's, and AND's, provide cohesion by stating the relations or contingencies

between propositions specifying different events and states.

Because programming languages are formal systems such as predicate calculus, the permissible relational terms in conditionals are defined without reference to the meanings of the propositions they bind together. In natural languages, however, the meanings of the relational terms vary according to the meaning of propositions they connect (Winograd, 1977). As result of his research on the vernacular use of conditionals, Fillenbaum (1974, 1975) suggested that there is no single meaning for conditional operators, such as IF and OR, in the English language. In English, the meaning of IF and OR derives from the meaning of the propositions that these conditional terms bind together and condition.

Miller (1974, 1981) conducted a series of experiments to explore the manner in which non-programmers organized natural English commands into programs for solving various sorting, selecting, and attribute testing problems. In addition to the relative referencing of conditional terms, Miller observed a critical difference in the order in which fundamental actions were conditioned. In natural language procedures, actions usually are expressed first and then the special conditions or circumstances under which they will be executed are stated. This "action conditional" order is reversed in programs. Most programming languages

require the stating of conditions under which actions are to be performed before they are performed. This "conditionalized action" allows for a great deal of unambiguous, specific control not usually practiced when procedures are stated in English.

Miller's most surprising observation was the infrequent appearance of even loosely defined conditional terms in his subjects' English language programs. This is not surprising. Most of the control aspects of procedural specification in English are provided not by the use of conditional terms but by simple, linear sequences of instructions, bound together by the repetition of concepts and propositions ("Do A, then B, then C ..." etc.). The fundamental organization of almost all procedural specification in English, therefore, is linear-sequential: propositions are organized in linear block structures, basically lists, with little or no explicit conditioning. When reading computer programs, however, naive or novice programmers encounter control structures that are explicit, highly specific, and quite different from those found in the more familiar, natural language procedures.

Consider the program in Figure 1. It can be represented in at least two ways. Because of his experiences with natural language procedures, the novice might expect that the program is organized as a linear sequence of programming commands. His representation of

the program, therefore, would reflect the temporal-sequential order by which computers execute program code. However, this "temporal-sequential schema" is not the only way to represent a program. A number of artificial intelligence researchers proposed that the domain of control ("A has control over B") rather than the domain of time ("Do A then B") provides the links that bind together the propositions of programming languages (Green et al., 1980; Rich, 1978, 1979, 1981, 1982; Rich et al., 1978; Sussman, 1973; Waters 1979, 1982). In their view, programs are constructed of "input-output segments connected by control and data flow." (Rich & Shrobe, 1978, p. 461). These segments are the cliches of programming practice, well-known and reliable groupings of code that experienced programmers emit with little thought.

Figure 1. A Computer Program Expressed in BASIC and in Propositional Notation

BASIC Notation	Propositional Notation
5 DIM A (40)	DIM(A,40)
10 N = 20	SET(N,20)
20 Z = 0	SET(Z,0)
30 I = 0	SET(I,0)
40 IF A(I) > 0 THEN Z = Z + A(I)	LIF: COND(A(I)>0)→ SET(Z(Z+A(I)) GOTO (next statement)
50 I = I + 1	
60 IF I = < N THEN GOTO 40	AIF: COND(I=<N) → GOTO(40) GOTO (next statement)
70 END	

A segment, or program "chunk," is a group of propositions that interact by controlling each other or by passing data to one another. A chunk performs a clearly defined function within a program. A chunk need not be a closely grouped set of code; the component elements of a chunk may be scattered some distance from each other, may be embedded in "free" syntactic elements such as formatting commands, or may be associated with parts of other chunks. Consider the "chunkwise" decomposition of the program in Figure 2.

Figure 2. Chunkwise Decomposition of a BASIC Program.

---

1. Remove "Augmentation" Chunk:

```
A. DIM A(40)
   Z = 0
   Z = Z + A(I)
```

Leaving:

```
 5 ...
10 N = 20
20 ...
30 I = 0
40 IF A(I) > 0 THEN ...
50 I = I + 1
60 IF I =< N THEN
     GOTO 40
70 END
```

2. Remove "Filter" Chunk:

```
B. IF A(I) > 0 THEN
```

Leaving:

```
 5 ...
10 N = 20
20 ...
30 I = 0
40 ...
50 I = I + 1
60 IF I =< N THEN
     GOTO 40
70 END
```

Figure 2 (Continued)

3. Remove "Termination" Chunk:

```
C. N = 0
   IF I = < N THEN GOTO 40
```

Leaving:

```
 5 ...
10 ...
20 ...
30 I = 0
40 ...
50 I = I + 1
60 ...
70 END
```

4. Remove "Counter" Chunk:

```
D.  I = 0
    I = I + 1
```

Leaving:

```
 5 ...
10 ...
20 ...
30 ...
40 ...
50 ...
60 ...
70 END
```

(Adapted from Lukey, 1980a)

This program consists of a larger loop segment that adds the sum of the positive members of the first twenty elements of A to the initial value of Z. An analysis based on data and control flow considers D as sending a stream of numbers to C which terminates the flow at N, and then sends them to B which "filters out" all nonpositive numbers, sending the remaining numbers to A which uses them to access the elements of the array. All control flow links between these chunks can be described in terms of three controlling relations: "next," "invokes," and "returns." (Rich & Shrobe, 1976). This is not the only way to

represent a program; however, it is a method, Rich and Shrobe (1978) argued, that many experienced programmers "find intuitive."

This may not be the program representation that most naive or novice programmers find intuitive. Experiences with natural language procedures and texts prepare naive and novice programmers to expect quite different forms of organization. In a study of children's sensitivity to four types of expository text structure, Englert and Hiebert (1984) found that third and sixth graders were best able to deal with text organized according to sequence (the chronological ordering of events) and enumeration (the ordering of facts, details, and components as a list of points). Organization by sequence and enumeration characterizes "temporal-sequential" schemas for text. Not surprisingly, comparison/contrast, the structure most similar to the conditional organization of computer programs, was the most difficult text structure for children at both grade levels.

Although they were found to dominate children's "top-down" decoding of textual information, temporal-sequential schemas may not be the most efficient way of organizing and recalling text. In a study of the relationship between story structure and recall, Mandler and Johnson (1977) observed that causally connected episodes were remembered better by both children and adults

than were temporally connected episodes. Events that are causally connected were integrated ("chunked") into larger episodes, therefore requiring the recall of fewer items; temporally connected episodes remained fragmented and required greater effort to recall the same amount of information.

In their attempt to understand or compose a program, beginning programmers initially may try to interpret or arrange the propositions of the programming language in terms of a pattern derived from their experiences with English procedural specification and expository texts. This was suggested by Miller's (1981) observation that the specificity and completeness of his subjects' natural language programs decreased markedly as the complexity of the programming problems increased. In problems whose solutions required simple, linear sequences of commands, the propositional structures of English and programming languages are sufficiently comparable to allow a direct translation of natural language programs into formal computer programs on a line-by-line basis. Miller's subjects could solve simple problems by generating sequences of commands set in linear block structures; however, procedural specification in English had not equipped them with the rich array of control structures (IF...THEN...ELSE's, GOTO's, etc.) provided by formal programming languages for the solution of more complex

problems. Lacking familiarity with the special structures of programming text, the novice often has no alternative but to impose in a top-down fashion the temporal-sequential schema that characterizes the organization of natural language procedures such as repair manuals and recipes.

Most beginning programmers, especially young students, simply do not have schemas for the special way propositions are organized in computer programs. Failing to find the customary coherence of natural language procedures, the novice programmer encounters a similar, albeit more extreme, form of a difficulty encountered by young readers who, taught to read through the narrative structures of children's literature, falter on the expository structures of content textbooks (Kintsch & Young, 1984; Taylor, 1982; Thorndyke, 1978).

Sequence and centrality are two independent ways of representing the propositional structure of a computer program. A "temporal-sequential schema" for procedures emphasizes a sequential, command-by-command connection between the propositions in the order that they are executed. A "chunking schema" for procedures emphasizes an organization based on "centrality": propositions are bound together in chunks, and the chunks are nested within each other according to the functional relations of control and data flow. These two ways of representing program structures are illustrated in Figures 3 and 4.

Figure 3. Temporal-Sequential Program Schema (Expressed in Propositional Notation)

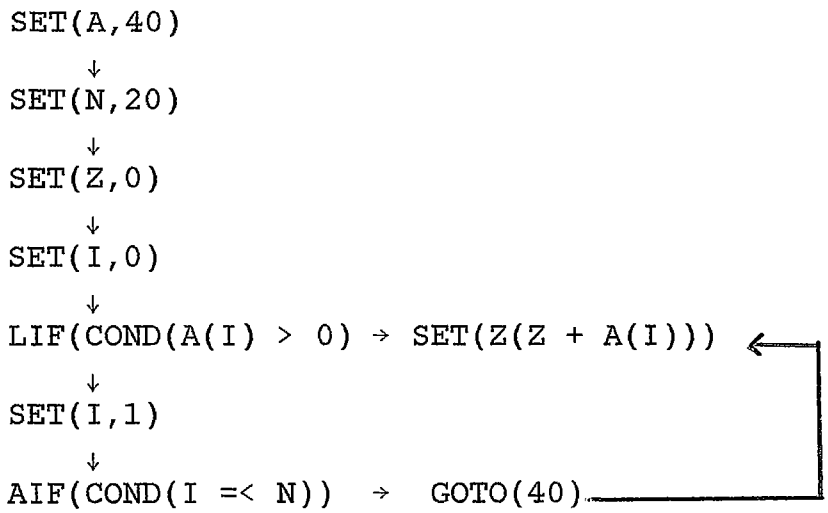
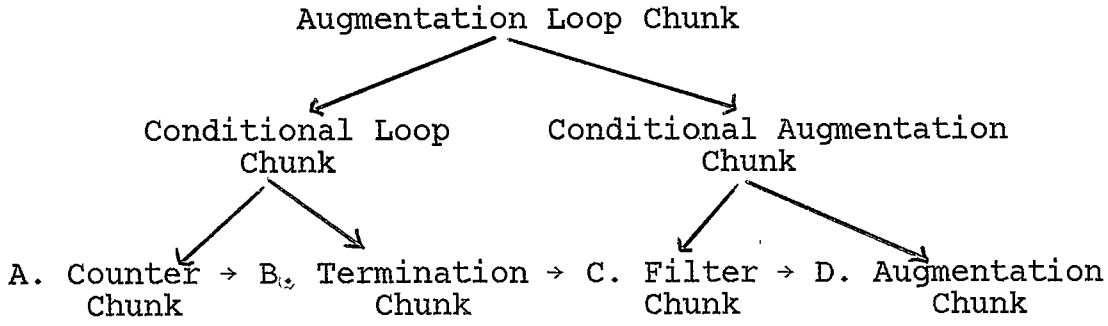


Figure 4. Chunking Program Schema[a]



[a] The programming code for chunks A, B, C, and D is presented in Figure 2.

In the temporal-sequential schema, individual propositions are integrated "vertically" according to order of program line execution. In contrast, a chunking schema integrates its propositions along two dimensions:

horizontal and vertical. Horizontal (computational) integration is achieved by control and data flow ("Chunk A controls or sends data to chunk B, B to C, etc."). Vertical (semantic) integration is achieved by abstracting the description of the chunk from the syntactic details of its code ("A conditional loop contains a counter and a termination chunk ...," etc.). Chunkwise representation of programs embrace both the sequential organization of control flow and the hierarchal organization of centrality. However, there is no conclusive evidence in the research that one or the other schema is a more effective way of understanding complex programs or of solving complex problems.

Program Complexity

The test of any instructional method, which claims to develop in novice programmers more efficient programming schemas or production systems, is how well it equips the programmer to deal with the "psychological complexity" of programs. Psychological complexity refers to the characteristics of programs that make them difficult to understand, modify, or debug (Curtis et al., 1979). Many computer scientists have argued that software complexity is a function of the number of control flow and data flow paths through a program. These paths, also known as "transfers of control," are associated with the looping and

branching characteristics of programs. It is assumed that increasing the number of paths within a program will increase the difficulty of understanding, testing, and modifying it.

Numerous techniques were developed to measure the density of transfer of control within a program, thereby indirectly assessing its psychological complexity. McCabe (1976), Myers (1977), Hansen (1978), Oviedo (1980), Chen (1978), Harrison and Magel (1981) devised software complexity metrics. These methods were summarized and evaluated by Harrison (1982). One of the most widely acclaimed techniques, Halstead's "Software Science," also was reviewed and criticized by Coulter (1983). Almost all of these complexity measures are derived from counts of program operators, from analyses of program flow typology, or from a combination of both.

A number of researchers proposed that complexity metrics are most predictive of the problem-solving performances of less experienced programmers or of performances involving long programs. Using a loose definition of programming complexity, Mayer (1975a, 1975b, 1976, 1980) found that program complexity negatively affects transfer of problem information in problem-solving tasks. In a series of studies, Curtis and his associates (1979a, 1979b) observed only weak correlations between programming complexity (as measured by the McCabe and

Halstead metrics) and the time subjects took in software maintenance tasks such as comprehending, modifying, and debugging programs. A replication of the study (Curtis, et.al., 1979c) resulted in stronger correlations when more subjects and longer programs were used. In both studies, the predictive power of their complexity metrics increased with program length and programmer inexperience.

The preceding results were corroborated by Chaudhary and Sahasrabuddhe (1983) in an experiment involving students in an introductory programming course. They compared the effects of two dimensions of programming complexity upon program comprehension. Complexity was defined in terms of control structure (the number of control statements) and execution structure (the number of operands and assignments). Program comprehension was measured by a memorization-reconstruction test. An increase in control structure or in execution structure was found to decrease the number and logical consistency of program lines recalled. Since there was no interaction between control and execution structures, Chaudhary and Sahasrabuddhe concluded that program complexity may be a "multidimensional entity."

It may be inferred from this preliminary work that most beginning programmers find it easy to work with linear sequences of computer programming code because of their structural similarity to the schemas of natural language

procedures. In contrast, most novices find it much more difficult to work with branching and looping structures, especially when set in longer programs. This is entirely consistent with the impoverished repertoire of control structures that Miller (1980, 1981) found in his subjects' English language programs. The control structures of computer languages are alien to the kinds of procedural specification one finds in natural languages, and people need to be trained in using those structures before they can adequately deal with the "connective tissue" of computer programs.

Program Structure and Skilled Performance: Instructional Implications.

Universities, secondary schools, and programming manuals traditionally have taught programming either by the "coding method" or, less frequently, by the "whole program" method. In the coding method, novices are taught to interpret and write programs from the "bottom-up." This instructional method prescribes teaching of programming languages through a careful, statement-by-statement exposition of the syntax. Instruction begins by having students construct short, simple programs and gradually progresses to programs of greater length and sophistication. In the whole program method, novices are taught to interpret and write programs from the "top-down."

Students are presented with entire programs, parts of which they are asked to modify and run to discover the purpose of the language's commands and structures. Actual programming instruction, of course, may contain elements of both methods.

Research studies of the effectiveness of these two methods have been inconclusive. In an investigation using two unrandomized groups of college students enrolled in beginning FORTRAN classes, Lemos (1975, 1980) discovered no differences between a group taught by the coding method and the group taught by the whole-program method. Using similar samples of students, McEntyre (1978) found a statistical difference on tests of problem-solving in favor of two groups taught by the whole-program method. Unfortunately, the two studies are not directly comparable because McEntyre used natural language programming experiences as part of her instructional treatments.

Inspection Methods

The ambiguity of the preceding results may have been due to the researchers' failure to attend to the kinds of planning knowledge produced by each method. In both the coding and whole-program methods, a novice could acquire any program schema - including chunking schemas - through analysis. Analytic, or "student-induced," schemas would occur if the student independently induced patterns of

program organization as the result of repeated programming experiences. Richard Mayer (1979a, 1979b, 1980), however, proposed a synthetic approach. He suggested that it may be possible to teach common programming chunks directly, thereby giving novices a well-organized network of programming schemas that they could use when attempting to interpret or write computer programs.

Mayer's suggestion coincided with artificial intelligence research at MIT and the University of Sussex in England. This work, which involved the study of engineering design and problem solving with artificial intelligence methods, may provide a rich source of ideas for improving the teaching of computer programming. One set of ideas, which emerged from several studies of problem solving with artificial intelligence methods, was dubbed "inspection methods." Two of the instructional treatments in this study were named "inspection methods," but they diverged somewhat from the definition originally given to this term by the MIT researchers.

Inspection methods "are the distillation of the collective experience of solving many problems in a particular domain" (Rich, 1981, p. 1). Sussman (1973, p. 9) proposed a theory of problem-solving skills as sets of "answer procedures." Each answer procedure was indexed by a description of the problem types for which it was appropriate and was linked to a set of pitfalls to avoid

when it was necessary to construct a new answer procedure. Elaborating on Sussman's work, Rich and Waters (1982) supposed that engineers in all fields use three techniques for solving complex design problems: abstraction, inspection, and debugging. Abstraction consists of adopting a simplified view of the problem without changing its essential character. This allows the problem solver to rough out an approximate solution without having to deal with its final complexity. Inspection is the act of recognizing the problem type and applying the appropriate design "chunks" as a tentative solution. Debugging involves the gradual retracting of the initially simplified view of the solution and the fine-tuning of the approximate design to the more detailed requirements of the solution.

When applied to computer programming, inspection methods consist of a taxonomy of common problem forms indexed to a set of algorithmic fragments ("cliches," "chunks," "segments," "schema," etc.) that are partial solutions to those problems. Both the abstraction and inspection phases of inspection methods are based on perception ("schema recognition"). The process begins with the recognition of familiar forms embedded in a given problem, proceeds to the recall of an appropriate chunk, and ends with application of the chunk to the solution of the problem. This is very similar to what cognitive

psychologists have described as a "production system" (Simon, 1980; Greeno, 1979, 1983).

Inspection methods are based on a finite collection of chunks that are built up from the reserved words and other primitive vocabulary of the programming language and on a set of procedures for determining when application of particular chunks to a programming problem is appropriate. As an instructional strategy, inspection methods involve teaching the beginning programmer how to construct several coordinated levels of program representation based upon control and data flow links that bind together the propositional structures of programs. This training should enable a novice programmer to segment a program into meaningful chunks, to describe the data flow and flow of control between these chunks, to describe the values of variables resulting from chunk interaction, and to provide an overall description of how the program and its constituent parts function (Lukey, 1980a, 1980b).

When applied to program composition, inspection methods consist of teaching the novice how to recognize the form of the solution in the program specification, or problem statement. This training involves alerting the novice programmer to the kinds of conditions and constraints that must be present in the problem statement before a particular programming chunk can be applied to its solution. The novice is taught how to select the

appropriate chunk and then fine-tune it to fit the specific requirements of the solution.

Beginning programmers will spontaneously devise methods for organizing their perception of and their access to information from computer text. Inspection methods, however, are used to maximize this perceptual behavior in ways consistent with efficient, expert-like programming practices. A number of researchers found "structural schema training" to be successful in improving children's processing of expository text (Brooks & Dansereau, 1983; Taylor, 1982). The question is whether similar expert-like schemas can be taught to novice programmers and whether novices can effectively apply these schemas in solving programming problems.

Modeling

The use of diagrams, pictures, and concrete objects as models for mathematical concepts has long been advocated as a means of developing in children a more meaningful understanding of arithmetic (Resnick, 1980). The effectiveness of these models in instruction is still a subject of some debate, a debate muddied by a lack of common definitions and wide variation in experimental treatments (Davis, 1980).

During the past decade, several researchers examined the effect of "models," "graphic organizers," and "concrete

analogies" upon comprehension of unfamiliar technical text. Royer (1976, 1976), for example, found that experimental groups reading a text about electrical conductance with the assistance of concrete analogies recalled more information than those groups not using these aids. Alverman (1981) observed that the use of a pictorial "graphics organizer" aided students' recall when they were required to reorganize expository, textual information, but had no effect when reorganization was not required. Model-like instructional aids seemed to improve understanding of textual materials with unfamiliar propositional structures for which the subjects lacked appropriate "schemas."

Programming shares many of the computational procedures found in mathematics, and programming languages present some of the same difficulties for novices as technical text. Perhaps for this reason, Richard Mayer proposed that concrete models might be used to foster a more meaningful understanding of computer programming languages and conducted a series of experiments (Mayer 1975, 1976, 1981) to test this notion. Mayer (1979) argued that concrete models facilitate meaningful learning because they function like Ausubel's advance organizers. However, Mayer's "models" were so different from Ausubel's "organizers" that Alvermann (1981) called the models "graphic organizers." Like Ausubel's organizers, "graphics organizers" were designed to encourage the learner to

actively assimilate unfamiliar material and to integrate it into relevant, previously-acquired knowledge. Unlike Ausubel's organizers, graphic organizers used pictures rather than prose to depict text structure and conceptual relationships. Mayer's model actually was a picture, portraying the major functional units of a computer as analogues of everyday objects. Memory, for example, was represented as an erasable scoreboard. Unlike advanced organizers, therefore, graphic organizers were constructed at an equivalent and concrete, rather than at higher and more abstract, conceptual level than the materials to be assimilated.

Mayer's research was designed to test the effect of a pictorial model upon novices' assimilation of programming text. Learning was measured by tests of transfer involving two variables: programming task type and program complexity. The two levels of the program task type were program interpretation and program generation. The program complexity variable encompassed simple statements, block structures, and looping. Mayer proposed a span of transfer based on task type and program complexity. Solving problems that involved the generation of statements and block structures was said to reflect simple retention ("close transfer"). Solving problems that required the generation of loops, the interpretation of loops, or the interpretation of block structures was said to reflect the

"near," "far," or "creative" transfer of information to new problem situations (Mayer, 1981). Mayer's experimental variables and transfer categories were loosely coupled and tended to vary from experiment to experiment.

In a study of model presence or absence in reading an instructional manual for a BASIC-like language (Mayer, 1975), a pictorial model group exceeded a no-models group on problems involving near transfer, but not on close transfer (retention). Both groups did poorly on problems requiring far transfer. Subsequently, the effect of model presence before or after reading a text was studied (Mayer, 1976). While the group receiving the model treatment after reading the text excelled on problems involving "near transfer," the group receiving the model treatment before reading excelled on problems involving "creative transfer."

In a similar experiment which used a recall/reconstruction test assessing "idea units" rather than transfer problems, Mayer (1975b) found that a models-before group recalled more conceptual (semantic) idea units, while a models-after group recalled more format (syntax). Because subjects in the models-after group referred more often to the model in their post-treatment tests, Mayer argued that this group intergrated the information better than did the other groups. Mayer (1980) also observed that these same results obtained in an experiment with another programming language. He suggested

that presenting a model after reading the text did not enhance learning because the subjects had already learned the text in a rote way.

In both these experiments, the models-before condition was tested for its effect upon comprehension of well vs. poorly organized text (segments of the instructional manual were placed in original vs. randomized order). Mayer (1978) found the models to be most useful when the material was poorly structured because they helped "the reader to hold the information together" (Mayer, 1981, p. 133). The model condition interacted not only with text structure but also with student aptitude. This bolstered some discoveries made in earlier studies. In an earlier work, Mayer (1975b) observed that students with low SAT (Scholastic Aptitude Test) mathematics scores scored higher under the models treatment than did the control group; however, students with high SAT scores performed less well under the models condition than did the control group. Meyer inferred from this finding that high ability students failed to profit, were even hindered, by the models treatment because "they already possessed useful 'models' for thinking about how a computer works..." (Mayer, 1981, p. 132). The models interfered with their pre-existing cognitive structures. In contrast, the low-ability could profit from the models because they lacked the prerequisite cognitive structures that might have caused interference.

Owens (1978) continued the study of the interaction between models and cognitive ability in computer-based problem-solving. Using a refinement of Mayer's problem-solving test, she tested the effect of two instructional variables: the presence or absence of Mayer's pictorial model, and an "algorithmic" (algorithm development) vs. a coding approach to presentation of programming syntax. Owens assigned her subjects to three, high to low treatment groups on the basis of their performance on the IBM Programmers' Aptitude Test. Upon completion of introductory BASIC programming class, the subjects were administered the problem-solving test. Owen found no significant treatment by ability interactions. An examination by gender, however, revealed that females with low arithmetic reasoning skills profited from an algorithmic approach, while high-ability women did better under the coding approach. In contrast, males as a group benefited from the algorithmic approach, but were hampered by the coding approach.

Mayer and Owen's studies raise important questions. Any instructional design, which seeks to use programming as a vehicle for developing problem-solving skills, requires techniques for developing in students a more meaningful understanding of the program language. In her conclusion, Owen raised the possibility of weak treatment effects arising from her use of the model. The models used in

subsequent studies must be different from the ones used by Mayer and Owen. Their models were static, pictorial, non-manipulative, and were presented only intermittently during instruction. Mayer and Owen's results may have been different had their models been dynamic, concrete and pictorial, manipulative, and presented continuously during instruction.

Summary

Successful problem solving depends upon the complimentary processes of understanding and planning. Planning is arranging a sequence of actions from the constraints of the problem domain to achieve goals and subgoals. Cognitive psychologists have defined understanding as the transformation of problem information into multiple, mental representations of the problem space. Representations provide abstract versions of the problem, and thereby guide planning by permitting direct retrieval of subgoals and by triggering relevant problem-solving strategies. In cognitive psychology, the notion of representation is embodied in a variety of theoretical formalisms such as "procedural networks," "production systems," "scripts," "schemas," and "chunks."

Some learning theorists have asserted that expert problem solving is based on the ability to form powerful, multiple representations of the problem space. "Chunks"

and "schemas" are the functional units of behavior associated with problem representation. In studies of computer programming, researchers observed that expert programmers interpret, compose, and debug computer programs in terms of large, well-organized "chunks" of code that can be flexibly generalized to a wide variety of programming situations. Novice programmers initially find it difficult to acquire similar expertise because procedural specification in computer programs is organized differently than it is in natural language programs such as repair manuals and recipes. In order to respond appropriately and efficiently to program specifications and program text, the novice must acquire the proper "schemas," knowledge about the form and content of programming structures and their relationship to different classes of programming problems. The test of this knowledge is how well it enables the programmer to deal with program complexity.

Traditionally, computer programming has been taught by either the whole program or by the coding method. Using the insights of contemporary studies of problem solving, this researcher developed two alternate methods, modeling methods and inspection methods, each appropriate for teaching programming to grade school children. The modeling methods involved the use of a concrete, manipulative device that helped the student better understand program text schemas. Inspection methods

consisted of teaching program languages as sets of well-organized chunks of code that served as stereotypical solutions to certain classes of programming problems. Two variants of each method were tested in the experiments in this study.

Chapter III.

PROCEDURES

The main goals of this study were accomplished by conducting two simultaneous experiments with four groups of fifth and sixth graders enrolled in a computer programming class. The first goal was to determine the effect that participation in a sequence of computer programming activities had upon the students' skill in solving nonroutine, mathematical word problems. This was the subject of the first experiment. The second, perhaps more important, goal was to examine the effect of two instructional strategies on the students' problem-solving behavior in computer programming. In the second experiment, therefore, two levels of a modeling strategy and two levels of a perceptual training strategy were tested. The modeling strategy involved the use of a manipulative model to represent certain features of the BASIC programming language. Modeling strategies were used to teach the "simple modeling" and the "complex modeling" treatment groups. The other strategy was a set of perceptual training techniques, "inspection methods," inspired by artificial intelligence theories of engineering problem solving. Inspection methods were used to teach the

"simple inspection methods" and "complex inspection methods" treatment groups. The two levels of each instructional strategy ("simple" and "complex") were created by varying the complexity of the programs and program fragments used in instruction.

Population Description and Sampling Procedures

Research Site

The experimental portions of this study took place in Jackson, Wyoming, from January through March, 1984. Pilot studies of the instructional treatments and of the experimental instruments had been conducted on groups of upper elementary grades, junior and senior high students from September, 1982 through December, 1983. Experimental treatments and pre/post testing took place in four beginning programming classes for upper elementary students sponsored by the Community Education Department of Teton County Public Schools (Wyoming). These classes were held in the computer lab at Jackson Intermediate School, Jackson, Wyoming.

Recruitment of Subjects

The subjects were 139 fifth and sixth graders drawn from Teton County Public Schools. One-hundred and thirteen students were assigned to the four treatment groups. Twenty-eight students were assigned to a control group for

purposes of the first experiment. Most fifth and sixth graders within Teton County attend Jackson Intermediate School, but four students were recruited from Wilson Elementary School, Wilson, Wyoming.

In December, 1983, the researcher visited and spoke to the fifth and sixth grade homerooms in those schools involved in the study. The classes were announced to the students and a brochure for their parents was sent home. In the brochure, the classes were described and an application form was provided. Permission to test and to release selected student records was stipulated as a condition for application. The application form contained a short questionnaire on which parents indicated the extent to which their children were acquainted with computers and computer programming.

The wide availability of microcomputing equipment in the Teton County Public Schools probably precluded recruitment of a completely "naive" sample of students. However, at the time of the experiment, computer use by most students in the nine to thirteen age group had not included more than a cursory introduction to programming. Although programming was a new experience for most students, a special effort was made to detect and classify the degree of programming experience among the students and to use this information in assigning them to the treatment and control groups.

Assignment of Subjects and Sample Size

One hundred and thirty-nine fifth and sixth graders were recruited for the experiment. The control group and each of the four treatment groups were comprised of 25 to 30 subjects, a number sufficiently large to meet the assumptions underlying the analysis of variance used in the data analysis. Since the sample was fixed by class enrollment, this researcher adopted a stratified random assignment of subjects to groups on the basis of their pretreatment performance on the Iowa Problem Solving Test (IPST).

Results of the Iowa Problem Solving Test (Shoen & Oehmke, 1980) were used as the basis for assigment of subjects to groups. All fifth and sixth graders in Jackson Intermediate and Wilson Schools were tested with this instrument in the second week of December, 1983. Each child was then assigned to one of four strata on the basis of his or her relative performance on the test. The four strata were determined by the quartile deviations in the pretest score distribution. Members of each of these strata were then randomly assigned in a proportional manner to either the control group or to one of the four treatment groups.

Table 1. Pretest: Iowa Problem Solving Test (Form 561)

Group	# Subjects	Means	SD
Simple Modeling	29	19.79	5.96
Simple Insp. Meth.	25	21.88	4.60
Complex Modeling	27	20.41	5.53
Complex Insp. Meth.	30	20.23	6.02
Control	28	20.11	5.67

The unequal distribution of subjects to treatment groups (29, 25, 27, 30) resulted from reassignments due to conflicts in student after-school schedules. Concerned that subject assignments or reassignments might have resulted in an unequal distribution of problem-solving abilities, this researcher performed an ANOVA on the results of the IPST pretest. Results of this ANOVA indicated that there were no significant differences among pretest means and variances (see Tables 1 and 2). These reassignments, therefore, did not upset the equal distribution of problem-solving abilities among subjects in the control and treatment groups.

Table 2. Anova: IPST Pretest by Experimental Groups

Test	Mean Squares (between)	Mean Squares (within)	F Ratio	F Prob.
IPST (Total)	17.10	31.46	.54	.70
Subtest 1	6.78	6.04	1.13	.35
Subtest 2	1.41	3.01	.47	.76
Subtest 3	3.71	5.06	.73	.57

This researcher also was concerned that differences in age, gender, grade placement, and programming experiences might introduce unintended variations in the experiments unless these factors were equally distributed among the five groups. Consequently, the groups were examined for relative equivalence of these factors. Group data were arranged in factor X group contingency tables, and chi square tests of independence were calculated to determine if the levels of each factor were distributed in the same fashion across all groups.

Results of this analysis indicated that group membership was independent of grade level, $\chi^2(3) = 1.11$, $p > .80$. Treatment group membership also was independent of gender, $\chi^2(3) = .15$, $p > .98$. Group membership was not significantly associated with previous programming experience, $\chi^2(3) = 5.41$, $p > .10$. Access to computers in the home or the parent's workplace also was found to be independent of treatment group, $\chi^2(3) = 3.53$, $p > .30$. All five groups, therefore, were accepted as equivalent in terms of mathematical problem-solving skill, grade level, gender, and access to computers outside the school.

Description of the Treatments

Treatment Groups

This study was centered on two simultaneous experiments. The first experiment, which tested the

"spin-off" effects of programming instruction upon nonroutine, mathematical problem solving, consisted of a five group, pretest-posttest design with four treatment groups and one quasi-control group. In the second experiment, the instructional treatments were arranged to ascertain the effect that the use of manipulative models and inspection methods had upon the development of the subject's programming skills. This was accomplished with a four group design that included two levels of the models condition and two levels of inspection methods. One models group was designated the "simple modeling" group; the other, the "complex modeling" group. The inspection methods condition consisted of the "simple inspection methods" group and the "complex inspection methods" group. Each group in each design contained from 25 to 30 subjects. Further explanation of group treatments is presented in the next section.

Experimental Validity

The experiments were arranged to minimize unintended variations in teaching behavior, instructional settings, and testing environments. Instruction consisted of a twelve-week series of computer programming classes. Meeting once a week for three hours, each group received approximately 20 hours of direct instruction in programming and spent 10 hours of supervised work on individual

programming problems and projects. In addition to the 30 hours of instruction, three hours were spent in post-testing. Each group was tested in three different settings: the Iowa Problem Solving Tests were administered in the students' homerooms, the first programming subtest was administered by treatment group, and the other two subtests were given in controlled mixed-group settings.

While varying the experimental conditions across the four groups, the researcher held constant all other features of the instruction. First, all four groups were taught the same subset of the BASIC programming language. Second, all four instructional sequences were built around the common goal of having each child build a simple video game program. The researcher divided each sequence into a series of smaller units organized around the acquisition of a small number of programming tools that could be used in some part of the video game program. During each unit, solving programming problems provided the vehicle for practicing these fresh skills and for learning general problem-solving methods. Third, the researcher taught each unit with a mastery learning approach: all students had to demonstrate mastery of a set of criterion tasks or problems before the treatment group was advanced to the next unit. Thus, the form and content of all four instructional treatments were similar: all students were taught the same set of programming syntax at approximately the same time,

they encountered the same programming problems, and they worked on the same kind of programming projects. The four experimental conditions were the only difference in the instructional sequences.

Since this researcher was the instructor for all groups, it was important to assure that instruction, which was given to all groups, was held constant, while the experimental treatments were allowed to vary according to plan. Several precautions were taken to assure continuity and consistency of the instructional treatments. All sessions were tape-recorded and were reviewed after each session to assess and correct the instructor's teaching behaviors. During each of four separate weeks, all instructional sessions were observed by a teacher volunteer from Jackson Intermediate School. Salient teaching behaviors were recorded and then analyzed at the end of each session. In addition, the instructor kept a log in which he recorded his own impression of his teaching and his students' learning. The audio tapes, the observers' notes and comments, and the instructor's own log provided corrective feedback that assured that the instructor's teaching behaviors did not deviate significantly from the standards set in the experiment.

General Problem-Solving Methods

For one-half hour during every session, the students were engaged in solving programming problems. This practice provided the opportunity for developing, discussing, and applying the strategies learned during the instructional treatment portion of each programming class. In addition, a limited number of general problem-solving methods were introduced and discussed. Although seeming to complicate the experiment, exposing all the subjects to these general strategies during the problem sets actually produced greater experimental validity. For example, a major goal was to determine if general problem-solving skills transferred to a related domain. This transfer could not occur without teaching strategies in the domain from which they are expected to transfer. Moreover, during the pilot courses, this researcher observed that higher performing students developed the problem solving methods outlined in Figure 5, but lower-performing students did not. Teaching these methods, therefore, was expected to promote greater among-subjects homogeneity.

Figure 5. General Problem-Solving Methods Taught in the Problem Sets

Generate and Test.	Solutions are generated one at a time and tested to see if they are acceptable.
Heuristic Search.	A basic set of operators, or "rules of thumb," are used to simplify search in a problem space by eliminating alternatives. The heuristics used were "working backwards," "search for a pattern," "construct a table or diagram," "determine extraneous or insufficient information," "solve a related problem," and "vary problem conditions."
Means-Ends Analysis	An iterative heuristic in which the differences between the problem situation and goal state is estimated and an operator is applied to reduce that difference. If this fails, the goal is replaced by a sequence of easier sub-goals, operators applied, differences estimated, and the process repeated.
Subgoal Decomposition	Divide a problem into subtasks and attack each separately.

Modeling Methods

For two treatment groups (simple modeling and complex modeling), an extensive use of a manipulative model supplemented the expository presentation of the programming language.

A model is a procedure or a device to represent in a concrete and familiar way that which is abstract, unfamiliar, or hidden. A model simplifies or reorganizes the stimulus features of the object or process being

modelled such that it becomes analogous to familiar elements in the user's past experience. This analogical relationship in effect becomes a "knowledge bridge" that facilitates transfer between what the learner already knows and the unfamiliar object or process (Royer & Cable, 1975; Gick & Holyoak, 1980).

Subjects in both the "simple modeling" and "complex modeling" groups used models that represented in a rough, analogical fashion the way a computer processes programs. These devices allowed the students to "play computer" with the printouts of simple programs or program fragments.

The models were constructed of clear 8 1/2 X 11 " acetate overlays upon which were imprinted rectangular subdivisions that represented three major functional units of a computer. (1) Executive control was represented by a "pointer arrow" set in a long, narrow window that was placed over the line of code to be executed. After each line of code was "processed" by the student, the acetate sheet was moved so that the next line of code appeared in the window. (2) Memory was represented by a series of tabbed rectangles. When a line of code declared and assigned a value to a variable, the student printed the variable "name" on the tab and its string or numerical value in the rectangle. Printing was done with an erasable, felt pen. Thus, if a line of code mandated the change or exchange of a variable value, the old value was

erased with a small, moist sponge and the new one was substituted. (3) Output (the monitor or printer) was represented by a large rectangle upon which the student printed what should appear on the screen given the program line currently being executed. When graphics output was expected, the output window was overlaid with a clear, coordinate grid to represent the Apple IIe computer's low resolution, graphics screen.

Students in the simple modeling and complex modeling groups used the model described above. Throughout the course, the students in both groups modeled the execution of programs and program fragments containing the programming concepts presented during the expository portions of each lesson. The modeling exercises consumed between 25 and 35 minutes of each instructional session.

The modeling groups differed in the size and computational complexity of the programs or program fragments that their students modeled. Computational complexity, as defined in Chapter II, consists of those characteristics (noteably the number of lines of code and density of control transfers) that make programs difficult to understand, compose, and debug. The "simple modeling" group modeled the execution of programs or program fragments of relatively low complexity; and the "complex modeling" group modeled the execution programs or program fragments of relatively high complexity.

Relative low and high complexity was determined by applying the software complexity measure developed by Hansen (1978). When applied to a computer program, Hansen's measurement results in a 2-tuple (a,b), where a is the number of forward branching and looping statements and where b is a count of operators in the program. Branching and looping statements include IF-THEN constructs, GOSUBs, and conditioned or unconditioned GOTO's. The count of branching and looping statements also is known as the cyclomatic number. Operators include primitive operators (+, -, *, AND, etc.), assignment, subroutine calls, and input or output statements (PLOT, INPUT, etc.).

Each week, all groups were taught with parallel, or equivalent, programs in the sense that the programs performed equivalent functions. However, the programs used with the complex modeling group were relatively more complex than those used with the simple modeling group. The number of programs modeled by the complex group was reduced, hence both groups were exposed to the same number of operations and control structures during any one session.

Several rules of thumb guided the composition of these programs. First, in writing programs to illustrate simple statements or linear blocks, the fewest operators necessary were used in the simple modeling group's programs, and twice as many operators were used in the complex modeling group's programs. Second, with few exceptions, the simple

modeling group's programs contained no more than one loop or forward branch, thereby rarely exceeding a cyclomatic number of two; the cyclomatic numbers of the complex programs were always one or two degrees greater. Examples of parallel programs, each differing from the other in its relative complexity, are displayed in Figure 6.

Figure 6. Two Parallel BASIC Programs Used in Instruction

Less Complex Program	More Complex Program
10 INPUT A 20 IF A > 31 THEN GOTO 40 30 PRINT A 40 END	10 HOME 20 INPUT A,B 30 IF A => 30 THEN GOTO 70 40 IF A = 0 THEN GOTO 20 50 PRINT A,B 60 GOTO 20 70 END

These same rules were used to design programs for the simple and complex inspection methods groups. Whenever possible, the programs designed for each modeling group were used with the inspection methods counterpart. For example, programs used to teach the simple modeling group would often be used to teach the simple inspection methods group.

Inspection Methods

One goal of this study was to determine if developing programming skills as sets of productions through

perceptual training enhanced the students' problem-solving skills. Two groups, therefore, were taught a set of programing skills subsumed under the title "inspection methods." Every subject in these two groups received between 25 to 35 minutes of training in inspection methods each weekly session.

Expert problem solvers possess flexible repertoires of solutions mapped on recurrent problem conditions. These repertoires consist of organized pairs of recognizable problem conditions linked to sequences of actions that are partial, proximate solutions to those problems. These sets of condition-action pairs are known to the artificial intelligence community as "production systems" (Newell & Simon, 1972; Simon, 1980).

Inspection methods are ways of developing within students expert-like, production systems for solving programming problems. This is done in two steps. First, students are trained to recognize "problem type" clues contained in the program specification. Second, an understanding of these problem conditions is linked with appropriate actions ("chunks") by means of perceptual training techniques. These techniques are ways of developing in naive and novice programmers the ability to respond to a problem statement by quickly recalling a repertoire of programming "chunks," deciding which is the best, approximate solution to the problem, then fine-tuning

this approximate solution to the specific requirements of the problem statement.

A programming chunk is a group of statements which directly interact (either by exchanging information or by controlling one another) to produce a clearly defined function within a larger program. During instruction, each chunk is presented as a "tool" and is given a name taken from the common objects of the students' everyday world. These were part of a "BASIC Toolbox" of fifteen chunks with names like "number-holders," "loops," "filters," and "counters" (see Figure 7). These names in effect "tagged" (Kintsch, 1978) a chunk to a set of propositions about its behavior, a set of rules for combining it with other chunks, and a set of problem classes for which it was an appropriate, approximate solution.

Figure 7. Chunks used in the Inspection Methods Treatments (The BASIC Toolbox)

Name	Example(s)
1. Switch	30 NEW 100 HOME 20 END
2. Number Holders	45 A = 5 130 INPUT A
3. Word Holders	35 A$ = "HELP" 55 INPUT A$
4. Output Chunks	10 PRINT "HELP" 55 PLOT 2,16
5. Single Swap	15 A$ = B$ 15 A = B
6. Double Swap	65 A = C 70 C = B 75 B = A

Figure 7 (continued)

7. Arithmetic Machines	10 INPUT A 20 INPUT B 30 C = A + B 40 PRINT C
8. LoRES Paintbox	50 GR 60 COLOR = 3 70 PLOT X,Y
9. One-Line Filters	45 IF A>6 THEN PLOT 3,12
10. Branching Filters	15 IF A = 6 THEN GOTO 25 20 PRINT A 25 END
11. Counters	30 X = X + 1 25 A = A * 2
12. Filter Loop	10 X = 0 20 PRINT X 30 X = X + 1 40 IF X>40 THEN GOTO 20
13. Loop Filter	35 INPUT X 40 IF X=0 THEN GOTO 60 45 X = X * 2 50 GOTO 35 60 END
14. For ... Next Loops	
14a. Number Changers	10 FOR X = 1 TO 10 20 PRINT X 30 NEXT X
14b. Repeaters	10 FOR X = 1 TO 10 20 PRINT "HELP" 30 NEXT X
14c. Timers	10 FOR X=1 TO 10: NEXT X
15. Random Number Generators	60 A=INT(RND(1)*100 + 1)
16. Paddles	45 X = PDL(1)

Inspection methods consisted of five skills derived from theories of engineering problem solving: chunk

recognition, chunk segmentation, recognition of problem schema, debugging approximate solutions, and description of control flow and variable values (Sussman, 1975; Lukey, 1980; Rich & Waters, 1982). A training technique was developed for teaching each of these skills. Perceptual training by means of a tachistoscope helped the students recognize and name specific chunks. Through segmentation training, the students were taught to decompose programs into their component chunks or chunks into their component subchunks. Schematic training assisted students in matching chunks to descriptions of their behavior and matching problem statements to chunks that were approximate solutions to those problems. By means of approximation training, or debugging, the students were taught to fine-tune chunks that were approximate solutions of programming problems into more satisfactory solutions. Finally, the students were trained how to ascertain the values of chunk variables and the state of control flow within and among chunks at any given stage of program execution.

Like the groups in the models condition, treatments for the inspection methods groups differed in the computational complexity of the programs and program fragments used in instruction. Recognition exercises for students in the "simple inspection methods" group focused on isolated chunks, while students in the "complex inspection

Group	Recognition Training	Segmentation Training	Schema Training	Control Flow Training
Simple Inspection Methods	Isolated chunks.	Individual chunks into component subchunks.	Single chunk specifications.	Within-chunk data flow and the value of variables resulting from the execution of a single chunk.
Complex Inspection Methods	Multiple chunks embedded within larger programs.	Larger programs into chunks.	Multichunk specifications.	Among-chunk data flow and the value of variables resulting from the interaction of two or more chunks.

Figure 8. Comparison of the Instructional Treatments of the Simple and Complex Inspection Methods Groups.

methods" group were trained to recognize chunks embedded in larger programs, consisting of two or more chunks. Students in the simple inspection methods group were trained to decompose individual chunks into their component subchunks; students in the complex group divided large programs into their component chunks. Schematic training for the simple inspection methods group focused on problem specifications whose solutions consisted of a single chunk; schematic training for the complex group consisted of matching problem specifications to solutions requiring two or more chunks. Instruction for the simple inspection methods group involved describing data and control flow within isolated chunks and the values of variables that resulted from the execution of a single chunk; in contrast, instruction for the complex inspection methods group was directed toward describing data and control flow among the component chunks of large programs and the values of variables that were the result of the interaction between two or more chunks within a program.

Methods of Collecting Data

Testing Procedures

Two weeks before Christmas vacation, 1983, the subjects were administered the Iowa Problem Solving Test (Form 561). In March, 1984, all subjects were tested with an equivalent form of the Iowa Problem Solving Test (Form

562) and with a computer programming test designed by this researcher. The Iowa pretest and posttest were administered in the subjects' home rooms within the week after the conclusion of classes. One programming subtest was given during the final treatment session. The remaining two subtests were given to mixed groups during the school day.

The Iowa Problem Solving Test

The Iowa Problem Solving Test (IPST) was chosen as the basis for proportional stratification and a measure of "distant" transfer because it is a paper-and-pencil test of reasonable validity and reliability that measures nonroutine, verbal problem solving involving mathematical topics. Two equivalent forms, each consisting of three ten-problem subtests, were designed specifically for fifth and sixth grade students. Administration time is forty minutes. Its reliability is reported in Table 3.

Although it requires grade-level proficiency in elementary arithmetic, the Iowa Problem Solving Test was designed primarily as a measure of heuristics and other general problem-solving methods. It is based on a modified version of Polya's (1974) four-step model of the

problem-solving process: getting to know the problem, choosing a strategy, doing the problem, and looking back. Each test form consists of three subtests, designed to measure the various heuristics and procedural skills required to perform three of the four problem-solving steps.

Shoen and Oehmke (1980) have established the IPST's content, concurrent, and construct validity in a three-year

Table 3. Alternate Forms Reliability of the Iowa Problem Solving Test by Grade and Subtest.

Form 561 (Pretest)			Form 562 (Posttest)		
Grade	n	r[a]	Grade	n	r
Grade 5:	1215		Grade 5:	1161	
Step 1.		.77	Step 1.		.69
Step 3.		.72	Step 3.		.58
Step 4.		.78	Step 4.		.71
Total:		.77	Total:		.81
Grade 6:	1314		Grade 6:	1184	
Step 1.		.77	Step 1.		.71
Step 3.		.68	Step 3.		.59
Step 4.		.77	Step 4.		.71
Total:		.86	Total:		.81

(Schoen & Oehmke, 1980)

[a]Spearman-Brown estimate.

study. Content validity was established against interview-based judgements of student performance of the three steps measured by the IPST. Because of its validation with "think-aloud" protocols and clinical interviews, the IPST is a useful paper-and-pencil alternative to protocol analysis of problem-solving skills when dealing with many subjects.

The Programming Test

Construction. This researcher designed a measure of those problem-solving operations that comprise computer programming. Since the test consisted of 120 problems, it was broken into three subtests, each administered on separate days. Each subtest consisted of a "target" test of eight problems and a main test comprised of 32 problems. Children taking the test were asked to solve the eight target problems. Upon completion of the target section, the children were given the correct answers to the problems. The administrator explained why the answers were correct, answered any questions, and provided whatever additional instruction was necessary to bring the children to mastery on the target problems. After a three-minute rest period, the children completed the main test of 32 problems. Excluding the time taken for the target section, each subtest took approximately 50 minutes to complete.

The test was built around three variables: program interpretation and generation (Mayer, 1981; Owen, 1978); problem relatedness (Wickelgren, 1974); and program complexity (Curtis, Sheppard, & Milliman, 1979; Harrison & Magel, 1982; McCabe 1976). The main test might be thought of as a 2 X 4 X 4 factorial design: PROBLEM TYPE X COMPLEXITY LEVEL X PROBLEM RELATEDNESS. This design can be represented as a three dimensional matrix composed of 32 cells, each cell defined by the intersection of one degree of each of the three factors (see Figure 9). Each cell contains three problems, one from each of the three subtests. The target sections can be represented as a similar matrix, except that they lack three degrees of the problem relatedness variable.

Each subtest consisted of eight target problems and 32 main test problems. Four target problems and sixteen main test problems involved the interpretation of program statements; four target problems and sixteen main test problems involved the generation of statements to solve programming problems. The test was in a multiple choice format that required the test taker to match a program to four descriptions of its function or to match a problem statement to four possible programming solutions. Interpretation problems required the subject to study a short program or program fragment and then select the best of four alternative descriptions of the program or program

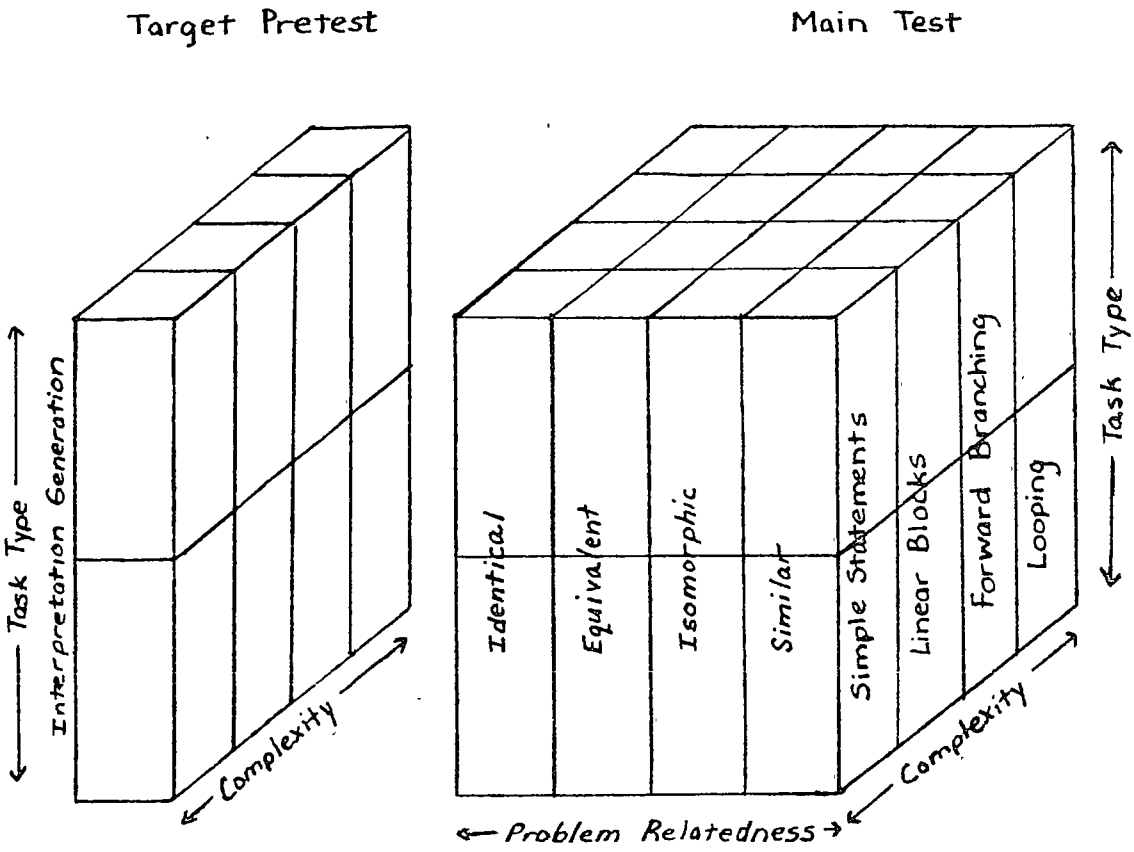


Figure 9. Programming Test Matrix

fragment's function. Generation problems presented the subjects with a problem statement, or program specification, and required them to select from four alternative programs or program fragments the best solution to the problem or the specification.

Each interpretation problem and each generation problem represented one of four levels of computational complexity: single statements, linear block of statements, forward branching within a linear block, and looping. McCabe's (1976) complexity metric was used to differentiate among each level of complexity. Because the content and structure variables of each target problem varied, comparable complexity levels based on operator counts, as in Hansen's (1978) metric, were not possible.

The target pretests were included to minimize the reactive and interactive effects of pretesting and to assess the degree of transfer of problem information from the target problems to the main subtest problems. Each of the main test problems was either identical, equivalent, isomorphic, or similar to the target problems. These problem relatedness categories were created by systematically varying the syntax, context, or content of the target problems (see Figure 10). This feature was consonant with the definition of problem solving as the successful application of previously acquired problem knowledge across a related class of problem statements or

programs in which syntax and context were allowed to vary, but the content or structure of the problems and programs were not.

Syntax variables included the arrangement or appearance of words and phrases in a problem statement and the arrangement or appearance of symbols in a line of program code. In interpretation problems, variable name and numerical quantity were the syntactic features varying from target pretest to main test. In generation problems, voice (active-passive), names of actors, sequence of information, and position of the question in the problem statement were the syntactic features changed.

Varying the context of a program or problem statement changed major syntactic features but preserved the meaning of the problem statement or the function of the program. The context of both programs and program specifications were varied in relation to the target problems. In interpretation problems, the chunk form (including the order of lines of code and arrangement of conditionals) or the computational object (strings, numbers, or graphics) were the contextual variables altered. In generation problems, the context was varied by changing keywords in the problem statement (for example, "sum" for "result of this addition") or by changing problem types (e.g., a problem involving the concatenation of strings instead of the addition of numbers).

Content or structural variables refer to the meaning, the semantics, of the problem statment or to the function of a particular program or program fragment. The content of an interpretation problem was changed by rewriting the code of the target item in such a way that it appeared the same as its corresponding target problem but performed a different function. The content of a generation problem was altered by changing the goals and elements of the problem statement while maintaining certain superficial similarities of phrasing.

Figure 10. Problem Relatedness Variables

Variable	Interpretation Problems (Vary Code)	Generation Problems (Vary Statements)
Identical	Same as Target	Same as Target
Equivalent (Syntax Variables)	Variable names. Numerical quantities.	Names of actors. Sequence of Information. Voice. Question position
Isomorphic (Context Variables)	Chunk form. Arrangement of operators. Computational object.	Keywords. Problem type.
Similar (Content or Structure Variables)	Different algorithm; similar expression.	Similar expression; different algorithm.

By manipulating syntax, context, and content variables, the researcher was able to write interpretation and generation problems that bore one of four degrees of relatedness to a single target problem: identity, equivalence, isomorphism, or similarity (see Figure 11). Identical problems were identical to the target problems. In equivalent, isomorphic and similar interpretation problems, the program code was changed, while the multiple choice statements were the same as those found in the target items. This order was reversed in generation problems. In composing the generation problems, the problem statements were varied while keeping the multiple choice program fragments the same as those in the target problems.

Figure 11. Relationship of the Main Test Problems to the Syntax, Context, and Content of the Target Problems

	Syntax	Context	Content/Structure
Identical	same	same	same
Equivalent	change	same	same
Isomorphic	same	change	same
Similar	same	same	change

Equivalent problems were created by varying syntax variables while holding context and content variables constant. In isomorphic problems, the context of the

target problem was changed; however, the content variables and, wherever possible, the syntax variables were left unchanged. When isomorphic transformations involved changing the computational object (numerical to graphic operations, for example), critical changes in syntax were unavoidable. Similar problems were created by changing content or structural variables, while maintaining superficial syntactic and contextual similarities to the target problems. On the surface, similar problems appeared to use the same commands and structures as the target problems, but the programs had quite different functions. Similar problems served as foils, designed to entrap those test takers who did not see the underlying functional, or semantic, differences.

Two other features of this test design should be noted. The three foils in the multiple choice format involved a syntactic, data flow, or control flow bug. In the interpretation problems, each of these bugs was represented by one of the multiple choice statements. In the generation problems, each bug type was embodied in one of the three programs that were incorrect choices. This arrangement allowed an item-by-item analysis of error patterns and an extensive correlational analysis with the other factors measured by the test. An analogous association existed among the correct items at each level of complexity and relatedness. In almost all cases, the

correct answer in an interpretation problem served as the problem statement for its corresponding generation item, whose correct answer in turn was the program for the interpretation problem.

Validity and Reliability. The researcher established the content validity of this instrument through inclusion of those key programming concepts that constitute the instructional objectives common to all four experimental treatments. These objectives were derived from a careful selection of fundamental programming concepts embodied in the Applesoft BASIC programming language. This process of analysis and selection was supervised by Dr. Lawrence Ellerbruch of Montana State University.

In reviewing the literature, this researcher found no intensively validated measures of problem-solving in the domain of computer programming. The absence of validated measures, against which this test could be compared, made establishment of criterion-referenced (concurrent and predictive) validity difficult. Nevertheless, this test has precedents. It was an elaboration and refinement of testing concepts successfully employed by Mayer (1978) and Owens (1978) in their studies. However, their instruments were validated on adult subjects, contained fewer test items, and consisted of fewer dimensions than does the instrument used in this study.

Several general psychological constructs were measured by this test. One was the psychological difficulty of various kinds of programming structures. Several studies reported in Chapter II have established high positive correlations between the computational complexity of these structures and the difficulty of understanding and debugging computer programs. Another construct was problem relatedness. Syntax, context, and content variables are well established categories in problem-solving research and have been shown to be predictive of problem-solving performance in mathematics (see Goldin & McClintock, 1979). This was confirmed in a series of pilot studies of various kinds of reconstruction/recognition programming tests conducted by the researcher at the Instructional Computing Lab, Montana State University (Bozeman, Montana), from 1982 through 1983. The problem relatedness construct was validated further through analysis of think-aloud protocols and clinical interviews gathered from 25 seventh-graders enrolled in a BASIC programming course held at Jackson Intermediate School (Jackson, Wyoming) during the autumn quarter, 1983.

Split-half reliability was established on the programming test. Split-half reliability, using the Spearman-Brown adjustment, was calculated on test results from 22 college students enrolled in instructional computing classes at Montana State University, 45 high

school students enrolled in a BASIC programming class at Jackson High School (Jackson, Wyoming), and 30 seventh graders enrolled in a similar class at Jackson Intermediate School (Jackson, Wyoming). The results are displayed in Table 4.

Table 4. Split Half Reliability, Computer Programming Test

Test	College	High School	Seventh-Grade
Subtest 1.	.85	.81	.80
Subtest 2.		.83	.75
Subtest 3.		.85	.78
Total:		.87	.83

Think-Aloud Protocols

In order to clarify and extend the conclusions based on the results of the Iowa Problem Solving Test and the programming test, two sets of think-aloud protocols were taken on twelve students. In these protocols, the students were instructed to describe everything that came into their head while attempting to solve two problems - one involving the interpretation of a computer program, the other involving the composition of a computer program from a problem statement. The purpose of these protocols was to provide clues to the internal, otherwise private, cognitive processes underlying performance on the paper-and-pencil tests (Ginsburg et al., 1983). The protocol tasks are found in Figure 12.

Three students were selected from each treatment group - one each from the lowest quartile, the middle two quartiles, and the highest quartile of the programming test distribution. Each student participated in two, 15 minute test sessions within three weeks of the programming test. Each session consisted of scripted introductory remarks by the researcher, a problem-solving session, and a debriefing. Six students were participants in a third session during which they solved related problems. These sessions, including the introductory scripts, closely followed the procedures outlined by Larkin and Rainard (1984). Every session was tape-recorded and cursory notes were made of the student's nonverbal behaviors. All students received payment for their participation.

Figure 12. Think-Aloud Protocols: Computer Program Interpretation and Composition Tasks.

Program Interpretation Task

```
90 PRINT "PLEASE TYPE IN A NUMBER"
100 INPUT VG
110 IF VG = 36 THEN GOTO 150
120 AV = VG/3
130 PRINT AV
140 GOTO 90
150 END
```

Program Composition Task

Write a program that allows a computer user to type in a series of numbers one at a time as input, prints out half of each number, and stops when the user enters a 0 as input.

Statement of Hypotheses

The hypotheses of this study deal with the general effect of computer programming experiences and the specific effect of two methods of programming instruction upon the problem-solving performance of 113 fifth and sixth grade students. This section contains a statement of each null hypothesis with an alternate hypothesis. A statement of this researcher's initial prediction of any observed differences resulting from the experimental treatments also appears after each null. The .05 probability level was used to determine the retention or rejection of all null hypotheses. In the statement of hypotheses, the term "scores" refers to the number of solutions either on the Iowa Problem Solving Test (IPST) or on the programming test.

#1. Ho: There will be no difference between the average IPST pretest and posttest scores of each of the five groups.

H1: There will be a difference between the average IPST pretest and posttest scores of each of the five groups.

The difference between a subject's pretest and posttest scores is his or her "change score." Failure to find a significant, positive difference between pretest and posttest averages indicates no change (gain or loss) in

mathematical, problem-solving skills during the pretest to posttest period. This researcher predicted retention of the null form because the students would not be able to apply the problem-solving skills they learned in programming to assist them in solving nonroutine, mathematical word problems.

#2. Ho: There will be no difference among the average change scores of the five groups.

H1: There will be a difference among the average change scores of the five groups.

This researcher expected that the null form would be retained because any gains in problem-solving skills, resulting from the computer programming experiences, would not generalize to the kinds of mathematical problem solving found in the IPST. If the null form of hypothesis #1 were to be retained, the lack of significant pretest to posttest differences would make the change scores meaningless, thereby rendering the testing of these hypotheses unnecessary.

#3. Ho: There will be no difference among the average IPST posttest scores for each of the four experimental groups and the control group.

H1: There will be a difference among the average IPST posttest scores for each of the four experimental groups and the control group.

This researcher predicted retention of the null hypothesis. He anticipated that there would be no significant gains in solving nonroutine, mathematical word problems as the result of learning how to solve programming problems. The research literature strongly supports the conclusion that these skills are domain-specific and do not generalize well outside the domain in which they are acquired.

The following hypotheses relate to the score totals from the researcher-designed programming test.

#4: Ho: The average score of the simple modeling group will be less than or equal to that of the complex modeling group.

H1: The average score of the simple modeling group will be greater than that of the complex modeling group.

This researcher predicted rejection of the null and acceptance of the alternate hypothesis because modeling large, complex programs would introduce unnecessarily confusing stimuli that would interfere with perceptual processing of key programming concepts.

#5. Ho: There will be no difference between the average score of the simple modeling group and that of the simple inspection methods group.

H1: There will be a difference between the average score of the simple modeling group and that of the simple inspection methods group.

This researcher anticipated that these methods of teaching key programming concepts would be equally effective given instruction with programs of relatively low complexity. Therefore, he predicted retention of the null.

#6. Ho: The average score of the simple inspection methods group will be less than or equal to that of the complex inspection methods group.

H1: The average score of the simple inspection methods group will be greater than that of the complex inspection methods group.

This researcher predicted rejection of the null form because presenting programming chunks in a setting of computational complexity would interfere with perceptual processing.

#7. Ho: The average score of the simple modeling group will be less than or equal to that of the complex inspection methods group.

H1: The average score of the simple modeling group will be greater than that of the complex inspection methods group.

Rejection of the null form was anticipated because of the negative effect of relatively high program complexity

upon the perceptual processing involved in program comprehension.

#8. Ho: The average score of the simple inspection methods group will be less than or equal to that of the complex modeling group.

H1: The average score of the simple inspection methods group will be greater than that of the complex modeling group.

Rejection of the null form was predicted because the complex programs used in instruction would interfere with the perceptual processing involved in program comprehension.

#9. Ho: The average score of the complex modeling group will be less than or equal to that of the complex inspection methods group.

H1: The average score of the complex modeling group will be greater than that of the complex inspection methods group.

This researcher predicted rejection of the null form because of his observations of earlier pilot studies in which modeling methods were more effective than other instructional techniques in promoting understanding of relatively complex computer programs, especially those characterized by nested looping.

Given its multidimensional character, it was possible to "unpack" the cells of the programming problem-solving

test and examine the last six hypotheses in terms of the scores from the programming task, program complexity, and problem relatedness items.

Statistical Analyses

Most statistical analyses were performed using the Statistical Package for the Social Sciences as documented in Nie (1975) and as implemented on Montana State University's Honeywell Level 66 computer. A small number of ANOVAs and multiple comparison tests (The Newman-Keuls, the Duncan etc.) were hand calculated, using the formulas presented by Ferguson (1981).

Summary

This study had two purposes. The first was to determine the effect that participation in a computer programming course had upon the performance of four groups of fifth and sixth graders in solving non-routine, mathematical problems. The experiment to accomplish this goal consisted of a five group, pretest-posttest design with four treatment groups and one quasi-control group. The second purpose was to ascertain the effect that the use of manipulative models and guided inspection methods had upon the development of these students' programming skills. To achieve this end, this researcher designed a four group, posttest experiment.

Equivalent experimental groups were assured by a stratified random assignment of subjects to group, coupled with post hoc checks for equal distribution of subjects according to age, gender, grade placement, and programming experience. Unintended variations in teaching and testing were minimized by providing identical class schedules, identical course content and class projects, monitoring and control of teaching behaviors, and uniform testing procedures. The only difference in treatment was the four experimental conditions. These were computer programming instruction by simple or complex inspection methods or by simple or complex modeling.

Modeling and inspection methods are instructional methods for promoting coherent and efficient representations of programming problems. Models accomplish this task by providing concrete analogies to familiar objects. Inspection methods are ways of developing in students expert-like representations known as production systems. In this study, inspection methods consisted of four distinct, but interlocked, instructional strategies: perceptual training, segmentation training, schema training, and debugging. Two variations of each instructional strategy were devised by varying the size and computational complexity of the programs used in instruction. In the simple modeling or inspection method groups, programs of relatively low complexity were

employed; in the complex modeling or inspection method groups, programs of relatively high complexity were used.

Assignment to treatment groups and analysis of differences among group performances in non-routine, mathematical problem solving were based on the results of pretests and posttests using alternate forms of the Iowa Problem Solving Test. Analysis of group differences in solving programming problems was based on the results of a programming test designed by this researcher. The programming test measured three dimensions of problem solving in computer programming: task type (interpretation or generation of programs), program complexity, and transfer of solution information from similar, previously-solved problems. Content, construct, and criterion-referenced validity was established on this test. Split half reliability of each subtest was established using groups of college, high school and seventh grade subjects.

This researcher hypothesized that there would be neither a significant increase in mathematical problem solving skills as the result of participation in the computer programming courses nor a significant difference in problem solving skills between subjects in the experimental groups who received programming instruction and those in the control group who received none. The remaining six null hypotheses related to the effect of the

four instructional treatments upon solving programming problems. It also was hypothesized that there would be no significant differences among the average programming performances of the four groups. If the results warranted rejection of any null, this researcher predicted that it would be due to complexity, not instructional method, variables.

Chapter IV.

RESULTS

The purpose of this study was to assess the general effects of computer programming instruction upon the development of skills for solving nonroutine, mathematical problems and to determine the specific effects of two methods of teaching the BASIC programming language upon solving programming problems. One hundred and thirty-nine fifth and sixth graders from Jackson Intermediate School (Jackson, Wyoming) participated in two, 12-week experiments to test a series of hypotheses related to these purposes. The results of these experiments are summarized in the next two sections. The last sections of this chapter are devoted to a post hoc analysis of other data produced by the experiment.

Experiment 1
The Effect of Programming Instruction upon Mathematical Problem Solving

Hypothesis 1

The null form of the first hypothesis was that there would be no difference between the average IPST pretest and posttest scores of each of the five groups. Given the exclusive emphasis in Jackson Intermediate School's math

curriculum upon computation and given the domain-specific character of problem-solving skills, the researcher expected that neither the school's mathematics curriculum nor the computer classes would develop over a three month period the skills for solving the non-routine, mathematical word problems found in the IPST.

The significance of differences between pretests and posttests of subjects within each group was determined by use of t tests. The results of this analysis confirmed the null hypothesis which predicted little skill development: there were no significant differences between pretest and posttest means for all groups ($p < .22$). Table 5 illustrates the absence of differences and the resulting low change scores.

Table 5. Group Performance: Iowa Problem Solving Test

Group	n	Pretest Means	Posttest Means	Change Score Means	t Prob.
S. Modeling	29	19.79	20.83	1.03	p<.22
S. Insp. Meth.	25	21.88	21.44	- .44	p<.52
C. Modeling	27	20.41	20.31	.11	p<.89
C. Insp. Meth.	30	20.23	19.40	- .83	p<.34
Control	28	20.10	19.82	- .29	p<.69

Since there were no differences between pretest and posttest, the null form of hypothesis #1 was retained.

Hypothesis 2

The null form of the second hypothesis was that there would be no differences among the average change scores of the five groups. Differences among groups, especially between the experimental groups and the control group, would have indicated changes in mathematical problem-solving skills resulting from participation in programming activities. Although their means were calculated, the change scores were null because of the insignificant differences between pretest and posttests for each of the five groups. On the average, the mathematical problem-solving skills of subjects in all five groups neither improved nor deteriorated over the course of the experiment.

The absence of any changes in problem-solving performances from pretest to posttest made testing hypothesis #2 unnecessary. Nevertheless, an ANOVA was performed on the change scores, but it did not reveal any differences at the specified level of significance, F (1, 138) = .87, $p < .49$. If the change scores had been valid, these results would have provided the basis for retaining the null form of hypothesis #2.

Hypothesis 3

The null form of hypothesis #3 was that there would be no difference among the average IPST posttest scores of

each of the five groups. A series of one-way ANOVAs was performed on both the main tests and on the three subtests to ascertain any such differences. The posttest ANOVA, displayed in Table 6, revealed no differences significant at the .05 level.

Table 6. ANOVA: IPST Posttest by Treatment Groups

Test	Mean Squares (between)	Mean Squares (within)	F Ratio	F Prob.
IPST (Total)	17.80	22.82	.78	$p<.54$
Subtest 1	2.56	3.80	.67	$p<.61$
Subtest 2	1.50	2.18	.69	$p<.60$
Subtest 3	4.36	5.34	.82	$p<.52$

Because there were no significant differences among posttest means, the null form of hypothesis #3 was retained.

The foregoing analysis precluded rejection of the null forms of the first three hypotheses. Computer programming was taught as a set of problem-solving skills to the students in the experimental groups. The means and distributions of solutions from the programming test (See Table 8) provided strong evidence that the students learned the skills necessary to solve problems involving the interpretation, composition, and debugging of computer programs. However, one cannot say that the problem-solving skills acquired within the domain of computer programming

transferred to any significant extent to non-routine, mathematical problem solving.

Post Hoc Analysis of IPST Data

A corollary question was whether the subjects' performance on the Iowa Problem Solving Test predicted their subsequent performance on the computer programming test. Pearson product-moment correlation coefficients were calculated between the IPST pretests and the totals of the programming test. The results are summarized in Table 7.

Table 7. Pearson r's: IPST Pretest with Programming Test Totals

Totals	Pearson r	Prob.	r^2
Solutions	.70	p<.001	.48
Syntax Errors	-.60	p<.001	.35
Control Flow Errors	-.67	p<.001	.45
Data Errors	-.55	p<.001	.30

There was a significant, but relatively moderate, positive correlation between number of solutions on the Iowa Problem Solving Test (IPST) and number of solutions on the programming test. There also were moderate, negative correlations between IPST solutions and the three types of programming errors measured by the programming test. The strongest subtest correlations were those that obtained between programming test solutions and the IPST subtest #2, "Looking Back" ($r = .77$, $p < .001$). The "Looking Back"

subtest measures the ability to vary problem conditions, to match solutions to problem conditions, and to relate problems to those that can be solved in the same way.

These significant associations with the programming posttest lent some validity to the use in this study of the IPST pretest as the basis for the proportionally stratified assignment of subjects to treatment groups; however, this does not justify concluding that the results of the IPST will predict problem-solving performance in computer programming, let alone other domains. These were only moderate correlations in which the results on both tests possibly were correlated with a third factor. Moreover, when the variance interpretation of the Pearson r is considered, these associations at best accounted for only half of the total variation.

Experiment 2
Hypotheses Related to Programming Instruction And Problem Solving

Five hypotheses were devised to guide an investigation of the effects of modeling and inspection methods on solving computer programming problems. These original hypotheses applied only to differences among average group solutions of problems on the programming test. The forthcoming analysis was held to that original commitment, but additional analyses were made. Since the test has a multidimensional structure, each dimension was examined

with reference to the original hypotheses. This analysis was arranged in the following order. First, the solution and error distributions were summarized. Second, hypotheses #4 through #9 were considered. These hypotheses were examined in terms of both total solutions and error patterns. Third, a post hoc examination was made of the problem task type, problem relatedness and program complexity categories in terms of the original hypotheses. Finally, the effects of grade level, gender, and programming practice upon programming skills were analyzed. These latter analyses required the pooling of the four treatment groups according to method of instruction (modeling vs. inspection methods) and complexity of instructional materials (simple vs. complex) so any differences caused by these factors could be tested with parametric statistics.

Solution and Error Distributions

Before considering the hypotheses, it is necessary to examine the distributions of solutions and errors from the total programming test. These are summarized in Table 8.

Table 8. Solution Means: Total Test.

Target Problems included:[a]

Group	n	Means	SD	Range	Skew.
S. Model.	29	61.26	20.80	34-96	.240
S. Insp. M.	25	62.75	22.85	34-115	1.113
C. Model.	27	61.29	16.54	32-91	.374
C. Insp. M.	30	52.70	20.28	26-108	.984

Target Problems excluded:[b]

Group	n	Means	SD	Range	Skew.
S. Model.	29	59.41	20.46	31-93	.229
S. Insp. M.	25	60.70	22.23	33-112	1.156
C. Model.	27	59.54	16.26	31-89	.386
C. Insp. M.	30	50.96	19.73	25-112	.984

[a]Maximum solutions = 120. [b]Maximum solutions = 96.

Notice the apparent differences between the solution means of the complex inspection methods group and the solution means of other treatment groups. There also were relatively large ranges and skewness values for both inspection methods groups, indicating positively skewed distributions. This was the result of exceptional performances by two subjects in each of the two groups. Consider also the distribution of error types, which is summarized in Table 9.

Table 9. Error Means: Total Programming Test.

Target problems included:

Group	Mean Syntax Errors	Mean Control Flow Errors	Mean Data Errors
S. Modeling	18.11	19.93	20.70
S. Insp. Methods	18.10	20.15	19.00
C. Modeling	19.38	19.50	19.83
C. Insp. Methods	23.85	21.33	22.11

Target problems excluded:

Group	Mean Syntax Errors	Mean Control Flow Errors	Mean Data Errors
S. Modeling	17.44	19.85	20.30
S. Insp. Methods	17.60	19.95	18.75
C. Modeling	18.79	19.25	19.42
C. Insp. Methods	23.15	21.11	21.78

Control flow and data error means appeared to be fairly homogeneous; however, an apparent difference, similar to that witnessed among the solution means, seemed to exist between the mean syntax errors for the complex inspection methods and each of the other treatment groups. As indicated in the following sections, these apparent differences were not significant at the prescribed level ($p < .05$).

Hypothesis 4

It was stated in the null form of hypothesis #4 that the average solutions made by the simple modeling group would be less than or equal to those of the complex modeling group. A series of one-way ANOVAs was used to

test hypotheses #4 through #9. The results are summarized in Table 10.

Regardless of whether the target items were included or excluded or whether the analyses include all subjects or were restricted to those attending three-fourths or more programming classes, no significant differences ($p < .05$) among solution means and variances could be discerned. On the basis of these results, the null form of hypothesis #4 was retained. On the average, subjects in the simple modeling solved as many programming problems as those in the complex modeling group.

Hypothesis 5

It was stated in the null form of hypothesis #5 that there would be no difference between the average solutions of the simple modeling group and those of the simple inspection methods group. Since the programs were of equally low complexity, it was anticipated that instruction by simple modeling or by simple inspection methods would result in equivalent performances.

The ANOVAs, summarized in Table 11, revealed no significant differences among all treatment groups and, therefore, no significant differences between the simple modeling and simple inspection methods group. Consequently, the null form of hypothesis #5 was retained.

Table 10. ANOVAs of Total Test: Solutions and Errors by Treatment Groups.

	F Ratio		F Probability	
Solutions	8/9	All[a]	8/9	All
Target Problems Included	1.32	(0.68)	.27	(.57)
Target Problems Excluded	1.33	(0.69)	.27	(.56)
Syntax Errors				
Target Problems Included	2.38	(1.66)	.07	(.18)
Target Problems Excluded	2.42	(1.70)	.07	(.17)
Control Flow Errors				
Target Problems Included	.30	(0.12)	.82	(.95)
Target Problems Excluded	.30	(0.09)	.83	(.97)
Data Errors				
Target Problems Included	.88	(0.51)	.46	(.68)
Target Problems Excluded	.90	(0.58)	.44	(.66)

[a]8/9 refers to F ratios and probabilties based on the performances of subjects attending nine or more programming classes; "All" refers to ratios and probabilities performed on all subjects.

Hypothesis 6

It was stated in the null form of hypothesis #6 that the average number of solutions of the simple inspection methods group would be less than or equal to those made by the complex inspection methods group. Rejection of this null was predicted because it was thought that presenting chunks in the context of relatively complex programs would interfere with perceptual processing of key programming concepts or inhibit generalization of these concepts to new problem situations.

Contrary to expectations, the null form of hypothesis #6 was retained. Results of ANOVAs on the solution means and variances revealed no differences among treatment groups and, therefore, no differences between the simple and complex inspection methods groups.

Hypothesis 7

It was stated in the null form of hypothesis #7 that the average number of solutions made by the simple modeling group would be less than or equal to those made by the complex inspection methods group. Again, rejection of the null hypothesis was predicted because of the program complexity factor.

Results of the ANOVAs revealed no differences among treatment groups and, thus, no differences between the

simple modeling and complex inspection methods group. The null form of hypothesis #7, therefore, was retained.

Hypothesis 8

It was stated in the null form of hypothesis #8 that the average number of solutions made by the simple inspection methods group would be less than or equal to those made by the complex modeling group. Rejection of this null was anticipated because of the difficulties posed by relatively complex programs.

Examination of Table 10 reveals no differences among treatment groups and, therefore, no differences between the simple inspection methods group and the complex modeling group. Accordingly, the null form of hypothesis #8 was retained.

Hypothesis 9

It was stated in the null form of hypothesis #9 that the average number of solutions made by the complex modeling group would be less than or equal to those made by the complex inspection methods group. This researcher predicted rejection of this null. He anticipated that modeling techniques would better prepare students to deal with relatively complex programs.

Contrary to expectations, the complex modeling and complex inspection methods groups solved an equivalent number of problems. These conclusions were based on the

results of the ANOVAs displayed in Table 10. The null form of hypothesis #9, therefore, was retained.

Post Hoc Analysis

Pooled Group Analysis of Solutions

Since no differences among the average solutions of the four treatment groups were found, the subjects were pooled to form two larger groups (modeling vs. inspection methods). ANOVAS then were performed on the number of solutions made by students in each group to determine the main effects of instructional technique. The differences among means and variances were not significant, F (1, 96) = 1.10, $p < .29$. Similarly, the solutions of the simple modeling and simple inspection methods groups were contrasted with those of the complex modeling and complex inspection methods groups to test the main effects of instructional materials. No significant differences were found between the performances of students trained with programs of relatively high complexity and those trained with programs of relatively low complexity, F (1, 96) = 1.11, $p < .29$.

Analysis Without Outliers

The distributions of problem solutions for both inspection method groups were positively skewed. This skewness was caused by the extreme scores of four

individuals, two in each of the inspection methods groups. The scores of these four individuals were removed from the analysis of variance to determine if their exceptional performances masked significant among-group differences. Although the F ratio was not significant, F (3, 91) = 2.12, $p < .10$, a comparison of means using t tests revealed a significant difference between each of the modeling groups and complex inspection methods in favor of the former, t (49) = 2.05, $p < .05$; t (47) = 2.28, $p < .03$. Because tests of among-group differences with ANOVA provide an estimate of total variance, the results of the t tests were discounted, and a finding of no difference among treatment groups was accepted.

Error Analysis

The main hypotheses also were tested in relation to error patterns. Inspection of Table 10 reveals no significant differences among groups in terms of average control flow, data, and syntax errors. The Duncan multiple range test was used to determine differences between group means. These results are summarized in Table 11. It should be noted that, in Table 11 and in subsequent tables that report the results of multiple range tests, no significant differences exist among those means listed under each numbered subset.

Table 11. Comparison of Total Syntax Errors Using the Duncan Studentized Range Test

	Subsets of Homogeneous Means		
Target Problems Included			
Groups	Subset 1.	Subset 2.	Prob.
C. Inspection Methods		23.9	$p<.05$
C. Modeling	19.4		
S. Modeling	18.1		
S. Inspection Methods	18.1		
Target Problems Excluded			
Groups	Subset 1.	Subset 2.	Prob.
C. Inspection Methods		23.2	$p<.05$
C. Modeling	18.8	18.8	
S. Modeling	17.6		
S. Inspection Methods	17.4		

Note. Harmonic mean used because of unequal cell sizes.

While no significant differences ($p < .05$) were apparent between the simple inspection methods group and each modeling group, subjects in the simple modeling and simple inspection groups seemed to have made substantially fewer syntax errors than did the complex inspection methods group. These differences were even more marked when the four outliers were removed from the inspection method groups, F (3, 91) = 3.12, $p < .03$.

Syntax error differences, however, probably reflect the relative complexity of instructional programs rather than method of instruction. When subjects were grouped according to the relative complexity of the programs used in their instruction, the results of ANOVA indicated that

students trained with comparably less complex programs made fewer syntax errors than subjects trained with more complex programs, F (1, 96) = 4.06, $p < .05$. When subjects were pooled according to method of instruction, neither instruction with models nor with inspection methods proved superior in reducing syntax errors.

The Program Task Type Variable

Within the programming test, half of the problems (60 items) involved interpreting programs and half involved generating programs. The interpretation tasks consisted of matching a program fragment to the correct description of its function; generation tasks required the selection of a program fragment that solved a stated programming problem.

Interpretation problems. A series of one-way ANOVAs was performed on group mean solutions and error types for the interpretation problems. The results of this analysis are summarized in Table 12. Inspection of Table 12 reveals that there were no significant differences ($p < .05$) among the group means and variances when subjects were asked to solve program interpretation programs.

Generation problems. In Table 13 is summarized the results of a series of analyses of variance performed on group performances involving program generation problems. There was no overall difference among mean solutions and errors, significant at the prescribed probability level

($p < .05$). Because the p values of the syntax and data error categories approached the prescribed level, multiple range tests were performed to ascertain the sources of these variations. From results of a Duncan test (see Table 14), it appeared that the simple inspection methods group made significantly ($p < .05$) fewer syntax errors when attempting to solve generation problems than did the complex inspection methods group.

Table 12. ANOVAs: Programming Task Type by Group, the Interpretation of Programs

	F Ratio		F Probability	
Solutions	8/9	All[a]	8/9	All
Target Problems Included	.90	(.89)	.42	(.45)
Target Problems Excluded	.90	(.92)	.44	(.44)
Syntax Errors				
Target Problems Included	1.78	(1.24)	.16	(.30)
Target Problems Excluded	1.80	(1.26)	.15	(.29)
Control Flow Errors				
Target Problems Included	1.03	(1.43)	.39	(.24)
Target Problems Excluded	1.13	(1.46)	.34	(.23)
Data Errors				
Target Problems Included	.14	(.09)	.94	(.97)
Target Problems Excluded	.12	(.21)	.95	(.89)

[a]The "All" column indicates the results of ANOVAs performed on test results from all subjects ($n=113$). The "8/9" column indicates results of ANOVAs performed on subjects attending nine or more programming classes ($n=98$).

Table 13. ANOVAs: Programming Task Type by Group, the Generation of Programs

	F Ratio		F Probability	
Solutions	8/9	All[a]	8/9	All
Target Problems Included	1.93	(.85)	.13	(.48)
Target Problems Excluded	1.95	(.94)	.13	(.43)
Syntax Errors				
Target Problems Included	2.18	(1.73)	.10	(.17)
Target Problems Excluded	2.01	(1.64)	.12	(.18)
Control Flow Errors				
Target Problems Included	1.37	(.38)	.26	(.77)
Target Problems Excluded	1.26	(.13)	.29	(.94)
Data Errors				
Target Problems Included	1.96	(1.26)	.13	(.29)
Target Problems Excluded	2.63	(1.97)	.06	(.12)

[a]See note in Table 12.

Table 14. Comparison of Syntax Errors in Generation Problems Using the Duncan Studentized Range Test

	Subsets of Homogeneous Means		
Target Problems Excluded			
Groups	Subset 1.	Subset 2.	Prob.
C. Inspection Methods		9.5	$p<.05$
C. Modeling	7.1	7.1	
S. Modeling	7.0	7.0	
S. Inspection Methods	6.6		

Note. Harmonic mean used because of unequal cell sizes.

With regard to group means involving data errors, the simple inspection methods group and, when target problems were excluded, the simple modeling group appeared to have made significantly fewer data errors when solving program generation problems than did the complex inspection methods group. These results are summarized in Table 15.

Table 15. Comparison of Data Errors in Generation Problems Using the Duncan Studentized Range Test

	Subsets of Homogeneous Means		
Target Problems Included			
Groups	Subset 1.	Subset 2.	Prob.
C. Inspection Methods		12.6	$p<.05$
S. Modeling	11.0	11.0	
C. Modeling	10.7	10.7	
R. Inspection Methods	9.8		
Target Problems Excluded			
Groups	Subset 1.	Subset 2.	Prob.
C. Inspection Methods		10.1	$p<.05$
C. Modeling	8.9	8.9	
S. Modeling	8.0		
S. Inspection Methods	7.6		

Note. Harmonic mean used because of unequal cell sizes.

There appeared to be no significant differences in the way the students solved interpretation or generation problems which could be ascribed to the instructional treatments. There were no significant differences among treatment groups in the average number of problems solved.

Method of instruction or program complexity may have had the effect of reducing the number of syntax and data errors made by students in attempting to solve generation problems. Even these conclusions must be tempered by a consideration of the restricted range of values involved in the calculations and the small differences between means. These differences may not be instructionally significant when they consist of one or two errors within a range of eighteen errors.

The Problem Relatedness Variable

The programming test was administered as three subtests on separate days. Each subtest began with pretest of eight target problems. Upon completion, the target problems were corrected, and the students reviewed and practiced the correct solutions under the researcher's direction. The main test of thirty-two problems was then administered. Each main test problem is either identical, equivalent, isomorphic or similar to a single target problem.

The transfer hypothesis. The problem relatedness categories define a span of transfer from a known solution (the target problem) to problems whose forms are increasingly dissimilar but whose functions are not. For example, equivalent and isomorphic programs are functionally identical to the target programs, but perform

their functions by means of code that is increasingly dissimilar from that used by the target programs. In contrast, similar programs have different functions than the target programs but consist of falsely similar code. If the student mastered the target problem solution, one would expect that he more likely would solve an identical or equivalent problem then an isomorphic or similiar one. Identical and equivalent problems, therefore, define a situation of "close" transfer; isomorphic and similar problems establish the reach of "far" transfer.

This researcher wished to determine the extent of transfer resulting from prior knowledge of the target solutions. Transfer of solution information from the target test items to main test items would be indicated by average solutions in any relatedness category that were significantly higher than those in the corresponding target problem category. An equally important goal was to determine the effect of instructional methods and materials upon the subject's solutions of problems within each relatedness category. The among-groups and among-categories solution means are summarized in Table 16.

The presence of transfer and the extent to which problem solutions were contingent upon problem relatedness categories were tested by means of one-way ANOVAs initially among all subjects and then within each group. The results are tabulated in Table 17. Noteworthy was the

Table 16. Solution Means Along the Problem Relatedness Axis

	Problem Type				
Group	Target	Identical	Equivalent	Isomorphic	Similar
S.Model.					
All	11.19	13.37	13.37	12.26	10.74
8/9	11.19	13.37	13.37	12.26	10.74
S.In.M.					
All	12.57	12.90	12.86	10.95	12.18
8/9	12.75	13.15	13.10	11.20	12.45
C.Model.					
All	10.89	12.29	12.64	11.86	10.46
8/9	11.33	13.17	13.13	12.54	11.13
C.In.M.					
All	10.55	11.52	10.73	10.75	10.83
8/9	10.15	11.19	10.26	10.63	10.48
All Subjects	11.21	12.43	12.38	11.49	11.00

Note. "All" indicates that means were calculated on the performances of all subjects. "8/9" indicates that means were calculated only on those subjects attending at least nine of the programming classes.

Table 17. ANOVAs of Problem Relatedness Categories: Solutions by Treatment Group

Group	F Ratio	F Probability
Simple Modeling	2.14	$p<.07$
Simple Inspection M.	.62	$p<.25$
Complex Modeling	1.27	$p<.25$
Complex Inspection M.	.35	$p<.25$
All Subjects	2.34	$p<.06$

contrast between the F ratios for the modeling groups and those of the other treatment groups.

The presence or absence of transfer of solution information from the target to main tests was determined by the significance of differences between the target problem category and every other relatedness category. If each relatedness category defines a separate degree of programming difficulty because of its relative similarity to a target problem, one should expect significant differences among the means and variances of every problem relatedness category. Thus, positive transfer of problem-solving information from target to main test would be indicated by identical, equivalent, and isomorphic categories significantly larger than the means of their target category. To test this proposition, pairwise comparisons of the category means within each group were conducted with t tests. A summary of these results is shown in Tables 40 to 43 (see Appendix B).

Analysis of the t test results revealed few differences between pairs of problem relatedness categories within either inspection methods group. There were no significant differences between category means within the complex inspection methods group. The only differences were between the similar and every other category within the simple inspection methods group. In neither inspection methods group was the target mean significantly different from the mean of any other problem relatedness category, suggesting little positive transfer of solution information from target to main test.

In contrast, relatedness categories in the simple modeling group were highly differentiated, although those in the complex modeling group were somewhat less so. In both groups the means of the identical and equivalent categories were homogeneous, but were significantly greater ($p < .05$) than those in the target category. It appears that students in the modeling groups acquired useful solution information during the target test episodes and were able to use this information in solving identical and equivalent problems. However, subjects in the modeling groups seemed to have transferred and effectively used this information only in solving identical and equivalent problems. There were no significant differences between the solutions of target and isomorphic problems or between the solution of target and similar problems. This indicated the ability of

students in the modeling groups to effectively use solution information in situations of "close transfer," involving closely similar problems, but not in situations of "far transfer," involving problems whose surface features had been radically changed.

Effects of transfer on problem-solving. Did positive transfer from target to main test give the modeling groups an advantage over the inspection methods groups in solving problems within each of the main test categories? To help answer this question, one-way ANOVAs, treatment group by relatedness categories, were performed on means and variances of both solutions and errors. The results are summarized in Table 18.

The differences among group mean solutions were not significant for identical, isomorphic, or similar problems. Results of ANOVA also indicated that the difference among mean solutions of equivalent problems was not significant at the prescribed level ($p < .05$). Because the p value ($p < .06$) was close to the prescribed level, a multiple range test was performed to ascertain the source of variation. Comparison of solution means with the Duncan range test indicated that students in the simple modeling group on the average solved more equivalent problems than did those in the complex inspection methods group. These results are summarized in Table 19. Another comparison of group means

Table 18. ANOVAs of Problem Relatedness Categories: Solutions and Error Patterns by Treatment Groups

	F Ratio		F Probability	
Target Problems	8/9	All	8/9	All
Solutions	1.47	(0.97)	.23	(.41)
Syntax Errors	1.29	(0.93)	.28	(.43)
C. Flow Errors	.34	(0.41)	.80	(.75)
Data Errors	2.61	(1.91)	.06	(.13)
Identical Problems				
Solutions	1.40	(0.89)	.25	(.45)
Syntax Errors	2.56	(1.40)	.06	(.25)
C. Flow Errors	.43	(0.34)	.74	(.80)
Data Errors	.26	(0.16)	.86	(.92)
Equivalent Problems				
Solutions	2.59	(1.82)	.06	(.15)
Syntax Errors	2.16	(1.88)	.10	(.14)
C. Flow Errors	1.20	(0.68)	.31	(.57)
Data Errors	1.86	(1.11)	.14	(.35)
Isomorphic Problems				
Solutions	.89	(.59)	.45	(.62)
Syntax Errors	1.64	(1.42)	.19	(.24)
C. Flow Errors	.97	(0.34)	.41	(.80)
Data Errors	.25	(0.50)	.86	(.69)
Similar Problems				
Solutions	1.11	(.75)	.35	(.53)
Syntax Errors	.83	(.48)	.48	(.70)
C. Flow Errors	.27	(.11)	.85	(.96)
Data Errors	.60	(.93)	.62	(.43)

(see Table 20) revealed that in attempting to solve equivalent problems students in the complex inspection methods group made significantly more syntax errors than did students in the other groups.

Table 19. Comparison of Solutions of Equivalent Problems Using the Duncan Studentized Range Test

	Subsets of Homogeneous Means		
Groups	Subset 1.	Subset 2.	Prob.
S. Modeling	13.7		$p<.05$
S. Inspection Methods	13.1	13.1	
C. Modeling	12.8	12.8	
C. Inspection Methods		10.4	

Note. Harmonic mean used because of unequal cell sizes.

Table 20. Comparison of Syntax Errors in Equivalent Problems Using the Duncan Studentized Range Test

	Subsets of Homogeneous Means		
Groups	Subset 1.	Subset 2.	Prob.
C. Inspection Methods		4.8	$p<.05$
C. Modeling	3.8		
S. Inspection Methods	3.5		
S. Modeling	3.3		

Note. Harmonic mean used because of unequal cell sizes.

Table 21. ANOVAs: Pooled Group Performances in Equivalent Problems

Solutions			
Groups	Means	F Ratio	F Probability
Modeling Insp. Methods	13.3 11.5	3.33	$p<.07$
Low Complex. High Complex.	13.4 11.5	3.95	$p<.05$

Syntax Errors			
Groups	Means	F Ratio	F Probability
Modeling Insp. Methods	3.5 4.2	2.0	$p<.16$
Low Complex. High Complex.	3.4 4.3	4.3	$p<.04$

Inspection of the rank ordering of means in Tables 19 and 20 indicated that complexity of programs used in instruction rather than method of instruction was the primary influence on transfer to equivalent problems. An examination of pooled group differences bore this out (See Table 21). When the students were pooled into high and low complexity groups, the difference in number of solutions of equivalent problems was significant in favor of the low complexity group, F (1, 99) = 3.95, $p < .05$. In contrast, an ANOVA on the performance of students pooled according to method of instruction (modeling vs. inspection methods) revealed no significant differences, F (1, 99) = 3.33, $p < .07$.

Teaching with low complexity programs also seemed to have resulted in improved debugging performance. Results of an ANOVA performed on complexity groups X syntax errors showed that on the average low complexity group subjects made significantly fewer syntax errors than did the high complexity group in attempting to solve equivalent problems, F (1, 99) = 4.30, $p < .05$. In contrast, when subjects were pooled according the method of instruction, there were no significant differences in the average number of syntax errors made by subjects in the modeling and inspection method groups, F (1, 99) = 2.04, $p < .15$.

In the results of an ANOVA, it was also revealed that there were no significant differences ($p < .05$) among means

and variances of syntax errors committed by students when attempting to solve identical problems. Because the p value ($p < .06$) was close to the prescribed level, a multiple range test was performed to ascertain the source of this variation. Comparison of group means with the Duncan test showed that the complex inspection methods group made significantly more syntax errors when attempting to solve identical problems than did the other groups. These results are shown in Table 22.

Table 22. Comparison of Syntax Errors in Identical Problems Using the Duncan Studentized Range Test

	Subsets of Homogeneous Means		
Groups	Subset 1.	Subset 2.	Prob.
C. Inspection Methods		4.6	$p<.05$
S. Inspection Methods	3.3		
C. Modeling	3.3		
S. Modeling	3.2		

Note. Harmonic mean used because of unequal cell sizes.

Summary. The foregoing data provided some basis for asserting that instruction employing modeling techniques helped students acquire and transfer solution information from the target tests. The differences between the average number of target problems solved and the average number of related problems solved bolster the argument that these students relied more on information acquired during the

target test episodes than did students in the inspection method groups. The benefits enjoyed by the modeling students in situations of "close" transfer did not extend to situations of "far" transfer, those involving isomorphic and similar problems.

Enhanced acquisition and transfer of solution information, however, may not have given students in the modeling groups a comparative advantage over other students in problem solving. Results of multiple range tests supported the conclusion that the solution and error means of the simple inspection methods group were not significantly different from the solution and error means of the modeling groups. Furthermore, when the groups were pooled according to method of instruction (modeling vs. inspection methods), the results of ANOVA did not show any significant disparity ($p < .05$) among group means in any of the relatedness categories.

Complexity of instructional materials may have been a more powerful factor than method of instruction, at least in influencing the solution of equivalent problems. Although there were no significant differences between combined modeling and inspection method groups in any of the problem categories, analysis of student solutions pooled according to the complexity of instructional programs lent credibility to the argument that students

instructed by means of low complexity programs solved significantly more equivalent problems.

The Program Complexity Variable

The third dimension of the programming test was program complexity. Each problem involved a program or program fragment whose structure was either a simple statement, a linear block of statements, a forward branch, or a loop. Through software complexity metrics, it has been predicted that the more complex looping and branching structures are more difficult to compose and to understand than those involving simple statements or simple statements set in block structures (Harrison et al., 1983). The complexity dimension, therefore, establishes a "difficulty gradient" in which problems involving simple statements should be easier to understand and solve than those characterized by linear block structures. Problems involving linear block structures, in turn, should be easier to solve than those involving forward branching and looping structures (simple statement < linear block < forward branching < looping).

The complexity dimension allows us to test whether the categories of the difficulty gradient may be used to predict the problem-solving performances of young programming novices. If the categories are predictive, students should solve more problems involving less complex

structures (simple statements and linear blocks) than those characterized by looping or branching. A perfectly predictive gradient would result in significant differences ($p < .05$) between the means of every complexity category (four separate subsets of means) because each category would represent one distinct degree of difficulty.

A second question was whether one or any combination of instructional methods or materials prepared students to deal with solving problems of relatively high complexity. For example, did using more complex programs in instruction prepare students to deal with program complexity in the test? When students were given training with programs of equal complexity, did modeling or did inspection methods better prepare those students to solve the problems contained in the programming test?

The first question was answered by testing the differences among the means and variances of correct items within each group's complexity categories, the horizontal axis of Table 23. The second question was answered by testing the among-group means and variances within each complexity category, the vertical axis of Table 23.

Table 23. Solution Means along the Complexity Axis

Group	S. State.		Linear B.		F. Branch.		Looping	
	8/9	All	8/9	All	8/9	All	8/9	All
S.Model.								
+ Tar:	18.8	(18.2)	16.4	(16.1)	13.7	(13.3)	12.3	(11.9)
- Tar:	15.4	(14.9)	13.3	(13.1)	11.3	(10.9)	10.1	(9.9)
S.In.M.								
+ Tar:	20.3	(19.2)	17.6	(16.3)	12.6	(11.6)	12.4	(11.4)
- Tar:	16.4	(15.6)	14.0	(12.9)	10.0	(9.2)	9.8	(9.2)
C.Model.								
+ Tar:	19.6	(18.6)	16.7	(15.9)	12.2	(11.5)	12.6	(11.8)
- Tar:	16.2	(15.4)	13.2	(12.5)	10.0	(9.6)	10.1	(9.6)
C.In.M.								
+ Tar:	16.8	(17.2)	14.1	(14.4)	11.0	(11.5)	10.8	(11.2)
- Tar:	13.8	(14.1)	11.1	(11.3)	8.9	(9.2)	9.0	(9.2)

α+ Tar: = Target problems included (Max. correct = 30).
- Tar: = Target problems excluded (Max. correct = 24).

The gradient hypothesis. The ANOVAs and multiple comparison tests of each groups's complexity categories are summarized in Tables 24 through 27. Inspection of the tabular summaries indicated that the difficulty gradient may be generally used to predict problem difficulty. Students in all treatment groups solved more problems, involving programs characterized by simple statements or linear block structures, than problems involving programs consisting of looping or forward branching structures. Nevertheless, the degree of difficulty posed by each complexity category varied specifically with treatment group.

Table 24. ANOVAs and Multiple Comparisons of Mean Solutions within the Complexity Categories of the Simple Modeling Group

ANOVA: Mean Corrects X Complexity Categories
F (3, 23) = 6.70, p < .01.

	Subsets of Homogeneous Means[a]		
Category	Subset #1.	Subset #2.	Subset #3.
S. Statements:	18.90		
Linear Blocks:	16.37	16.37	
F. Branching:		13.74	13.74
Looping:			12.26

S. Statements > F. Branching (p<.01)
S. Statements > Looping (p<.01)
Linear Blocks > Looping (p<.05)

[a]Multiple comparisons were calculated by means of the Newman-Keuls method, using the harmonic mean because of unequal cell sizes.

The difference among the simple modeling group's complexity categories was significant (p < .01). The multiple range test revealed a difficulty gradient which was highly differentiated, but which made no sharp distinction between linear block and forward branching or forward branching and looping structures (See Table 24).

Table 25. ANOVAs and Multiple Comparisons of Mean Solutions within the Complexity Categories of the Complex Modeling Group

ANOVA: Mean Corrects X Complexity Categories
F (3, 24) = 12.83, p < .01.

	Subsets of Homogeneous Means[a]		
Category	Subset #1.	Subset #2.	Subset #3.
S. Statements:	18.62		
Linear Blocks:		15.86	
Looping:			12.00
F. Branching:			11.64

S. Statements > Linear Blocks (p<.05)
S. Statements > F. Branching (p<.01)
S. Statements > Looping (p<.01)
Linear Blocks > F. Branching (p<.01)
Linear Blocks > Looping (p<.01)

[a]Multiple comparisons were calculated by means of the Newman-Keuls method, using the harmonic mean because of unequal cell sizes

The difference among the complexity categories of the complex modeling group also was significant (p < .01). The difficulty gradient was divided into three subsets with significant differences between every complexity category except looping and forward branching (See Table 25). Unlike the simple modeling group, however, a difference existed between the difficulty of linear blocks and simple statements with no overlapping variance between the two categories.

Table 26. ANOVAs and Multiple Comparisons of Mean Solutions within the Complexity Categories of the Simple Inspection Methods Group

ANOVA: Mean Corrects X Complexity Categories
F (3, 20) = 8.08 (p<.01)

	Subsets of Homogeneous Means[a]	
Category	Subset #1.	Subset #2.
S. Statements:	20.10	
Linear Blocks:	17.05	
F. Branching:		12.27
Looping:		12.01

S. Statements > F. Branching (p<.01)
S. Statements > Looping (p<.01)
Linear Blocks > F. Branching (p<.05)
Linear Blocks > Looping (p<.01)

[a]Multiple comparisons were calculated by means of the Newman-Keuls method, using the harmonic mean because of unequal cell sizes.

The difference among the complexity category means of the simple inspection methods group was significant (p < .01). The difficulty gradient consisted of a simple bifurcation between statements, linear blocks (the simplest programming structures) and the more complex looping and branching structures (See Table 26).

Table 27. ANOVAs and Multiple Comparisons of Mean Solutions within the Complexity Categories of the Complex Inspection Methods Group

ANOVA: Mean Corrects X Complexity Categories
F (3, 24) = 6.11, p<.01.

	Subsets of Homogeneous Means[a]	
Category	Subset #1.	Subset #2.
S. Statements:	17.23	
Linear Blocks:	14.37	14.37
F. Branching:		11.48
Looping:		11.21

S. Statements > F. Branching (p<.01)
S. Statements > Looping (p<.01)

[a]Multiple comparisons were calculated by means of the Newman-Keuls method, using the harmonic mean because of unequal cell sizes.

Although the difference among the category means of the complex inspection methods group was significant (p < .01), the difficulty gradient consisted of a simple contrast between simple statements and the more difficult looping and branching structures with linear blocks straddling this division. These results are summarized in Table 27.

Several general conclusions can be drawn from these analyses. The difficulty gradient prevailed, but did not perfectly distinguish between each complexity category. For all groups, program fragments consisting of simple statements were easier to interpret and generate than were those characterized by looping and branching structures.

Looping and branching structures consistently represented a single, albeit the most rigorous, degree of difficulty. Linear block structures varied with treatment group: only for the complex modeling group did linear block structures represent a distinct degree of difficulty. The effect of instructional method also was reflected in the more highly differentiated difficulty gradients of the modeling groups as compared to those of the inspection methods groups. Students in the modeling groups seemed more sensitive to the fine differences among program structures.

Instruction and complexity. If the difficulty gradient prevailed, what was the effect of instructional method on preparing novices to successfully solve programming problems involving each of the four different programming structures? The consistent differences between simple statements and branching-looping structures suggested a logical division between "program elements," programming syntax as represented by simple statements, and "program connections," the linear blocks, branches, and loops that organize the syntactic elements of programming languages.

To assess the effect of instructional method, a series of ANOVAs was performed on program elements by treatment groups and program connections by treatment groups. These results are displayed in Tables 28 and 30.

There were no significant differences among problem solutions, but there were significant differences among the syntax errors committed when attempting to solve problems consisting of simple statements ($p < .005$). In Table 29 are summarized the results of a series of multiple range tests used to determine the source of these differences.

A review of Table 29 indicated that students in the complex inspection methods group committed significantly more syntax errors ($p < .01$ & $p < .05$) than did those in the other treatment groups. A main effect in favor of the modeling groups seems to have occurred here: when the treatment groups were pooled according to instructional method (modeling vs. inspection methods), students receiving instruction using models committed significantly fewer syntax errors than those instructed by means of inspection methods, F (1, 99) = 4.61, $p < .04$. No comparable effect was significant in the results of ANOVAs on groups pooled according to complexity of instructional materials, F (1, 99) = 3.40, $p < .06$.

Neither instruction with models nor with inspection methods resulted in giving students a comparative advantage in solving problems involving simple statments. Analogous effects were evident in the results of ANOVAs performed on the group mean solutions of problems involving "program connections," that is, linear block, forward branching, and looping structures. Inspection of Table 30 indicates that

Table 28. ANOVAs: Program Complexity by Treatment Group, Program Elements: Simple Statements.

	F Ratio		F Probability	
Solutions[a]	8/9	All	8/9	All
Target Problems Included	1.75	(0.49)	.16	(.68)
Target Problems Excluded	1.60	(0.46)	.19	(.71)
Syntax Errors				
Target Problems Included	4.53	(3.27)	.005	(.02)
Target Problems Excluded	4.54	(3.18)	.005	(.03)
Control Flow Errors				
Target Problems Included	.67	(0.72)	.57	(.54)
Target Problems Excluded	1.33	(1.14)	.27	(.34)
Data Errors				
Target Problems Included	.21	(0.30)	.89	(.83)
Target Problems Excluded	.09	(0.19)	.96	(.90)

[a]Maximum score = 24 (with Targets), 18 (without Targets).

Table 29. Comparison of Syntax Error Patterns in Simple Statements Using Studentized Range Tests

Target Problems Included

	Subsets of Homogeneous Means		
Groups	Subset 1.	Subset 2.	Prob.
C. Inspection Methods		6.39	Duncan(.01)
S. Modeling	3.96		LSD (.01)
S. Inspection Methods	3.67		S.N.K.(.05)
C. Modeling	3.60		TukeyB(.05)

Target Problems Excluded

Groups	Subset 1.	Subset 2.	Prob.
C. Inspection Methods		4.96	LSD (.01)
S. Modeling	3.10	3.10	Duncan(.01)
S. Inspection Methods	2.67		
C. Modeling	2.52		
C. Inspection Methods		4.96	TukeyB(.05)
S. Modeling	3.10		S.N.K.(.05)
S. Inspection Methods	2.67		
C. Modeling	2.52		

αS.N.K. is the Newman-Keuls procedure. In all multiple range tests, the harmonic mean was used because of unequal cell sizes.

Table 30. ANOVAs: Program Complexity by Treatment Group, Program Connections: Linear Block, Forward Branching and Loop Structures.

	F Ratio		F Probability	
	8/9	All	8/9	All
Solutions[a]				
Target Problems Included	.76	(0.37)	.52	(.77)
Target Problems Excluded	.83	(0.51)	.48	(.68)
Syntax Errors				
Target Problems Included	1.36	(1.30)	.26	(.28)
Target Problems Excluded	1.23	(1.06)	.31	(.37)
Control Flow Errors				
Target Problems Included	.81	(0.59)	.49	(.62)
Target Problems Excluded	.74	(0.47)	.53	(.71)
Data Errors				
Target Problems Included	1.81	(1.36)	.15	(.26)
Target Problems Excluded	1.84	(1.54)	.15	(.21)

[a]Maximum score = 120 (with Targets), 96 (without Targets).

method of instruction did not give students in any treatment group a relative advantage in solving problems involving more complex programming structures. Nevertheless, multiple comparison of mean data errors using studentized range tests indicated that students in the complex inspection methods group committed significantly more data errors when solving problems involving linear block structures (See Table 31).

Table 31. Comparison of Data Error Patterns in Linear Blocks Using Studentized Range Tests

	Subsets of Homogeneous Means		
Target Problems Included			
Groups	Subset 1.	Subset 2.	Prob.
C. Inspection Methods		6.21	LSD (.01)
C. Modeling	5.04	5.04	SNK (.05)
S. Modeling	4.83		TukeyB(.05)
S. Inspection Methods	4.42		
Target Problems Excluded			
Groups	Subset 1.	Subset 2.	Prob.
C. Inspection Methods		5.18	LSD (.01)
C. Modeling	4.16	4.16	S.N.K.(.05)
S. Modeling	3.86	3.86	TukeyB(.05)
S. Inspection Methods	3.58		

Did training with more complex programs prove to be an advantage in solving problems involving the more complex linear block, forward branching and looping structures? On the ANOVAs performed on each complexity category and on all

complexity categories pooled in the "program connections" variable, no evidence was provided that students in any group solved more problems as a result of the relative complexity of the programs used in their programming instruction. When student scores were pooled into two groups according to the complexity of the programs used in instruction, however, students trained with relatively less complex programs made significantly fewer syntax errors in attempting to solve problems involving forward branching, F (1, 104) = 4.21, $p < .05$. They also committed fewer data errors when working with linear block structures, F (1, 104) = 5.65, $p < .02$.

Several conclusions were drawn from this analysis of program complexity variables. As a result of no experimental treatment, did students in one group have an advantage or disadvantage over those in other groups in solving programming problems involving either simple statement, linear block, forward branching, or looping structures. Nevertheless, as a result of both instructional methods and materials, "debugging" performance was affected. Students in the modeling groups made fewer errors involving syntax bugs when solving programming problems requiring simple statements. The inspection methods groups did less well: the complex inspection methods group in particular seems to have been disadvantaged in perceiving programming bugs involving

syntax and data errors. Furthermore, students trained with less complex programs committed significantly fewer syntax, control flow, and data errors in attempting to interpret or generate more structurally complex programs.

The Effect of Practice upon Problem Solving

Practice was an uncontrolled variable in this experiment. No effort was made to keep students from using Jackson Intermediate School's computer lab before school and during their free time. A log was meticulously kept on the number of minutes each subject spent programming - as opposed to playing games, word processing, or engaging in some other nonprogramming computer activity.

At the end of the experiment, practice time for each student was summed, and the entire sample was divided into a low practice group (0 to 2 hours of practice outside the experiment treatments) and a high practice group (2 to 20 hours practice). The two hour division was chosen because it represented the fiftieth percentile of the sample and because it provided adequate cell sizes for the MANOVAs used to test the effects of practice.

It was determined by means of chi square tests that practice level was independent of treatment group, $\chi^2(3) = 5.63$, $p > .10$. The independence of practice level from treatment group provided some assurance that practice had

additive, rather than confounding, effects upon the treatments.

Treatment group analysis. A series of MANOVAs was calculated on total solutions and errors by practice level and treatment groups pooled according to method of instruction. The results are summarized in tables 32 through 35.

High practice had a strong main effect of increasing the number of problems solved. Method of instruction made no significant difference, and there were no interactions between instructional method and practice level. A review of the data summarized in Tables 33 through 35 indicated that high practice also had the main effect of lowering the number of syntax, control flow, and data errors. Noteworthy were the relatively low, but nonsignificant ($p < .05$), probabilities for method of instruction. There were no interactions. The strong, main effect of high practice on increasing solutions and decreasing errors was also evident along the three dimensions of the programming test: problem task type, program complexity, and problem relatedness ($p < .001$).

Pooled group analysis. An effort was made to determine the effect of practice level upon the treatment groups pooled according the relative complexity of the programs used in their instruction. The results of a

Table 32. MANOVA: Total Solutions, by Instructional Method and Practice Level

(Main Effects)			
Target Items Included		Target Items Excluded	
Modeling	Inspection M.	Modeling	Inspection M.
61.27 (N=51)	56.98 (N=47)	59.47 (N=51)	55.11 (N=47)
($p<.26$)		($p<.24$)	
Low Practice	High Practice	Low Practice	High Practice
49.26 (N=42)	56.98 (N=56)	47.50 (N=42)	64.79 (N=56)
($p<.001$)		($p<.001$)	
(No Interactions)			

Note. Means calculated on the performances of subjects attending nine or more programming classes.

Table 33. MANOVA: Total Syntax Errors by Instructional Method and Practice Level

(Main Effects)			
Target Items Included		Target Items Excluded	
Modeling	Inspection M.	Modeling	Inspection M.
18.71 (N=51)	21.40 (N=47)	18.08 (N=51)	20.79 (N=47)
($p<.13$)		($p<.12$)	
Low Practice	High Practice	Low Practice	High Practice
23.48 (N=42)	17.39 (N=56)	22.83 (N=42)	16.79 (N=56)
($p<.001$)		($p<.002$)	
(No Interactions)			

Note. Means calculated on the performances of subjects attending nine or more programming classes.

Table 34. MANOVA: Total Control Flow Errors by Instructional Method and Practice Level

(Main Effects)

Target Items Included			Target Items Excluded		
Modeling	Inspection M.		Modeling	Inspection M.	
19.73 (N=51)	20.83 (N=47)		19.57 (N=51)	20.62 (N=47)	
	($p<.42$)			($p<.44$)	
Low Practice	High Practice		Low Practice	High Practice	
23.62 (N=42)	17.73 (N=56)		23.48 (N=42)	17.52 (N=56)	
	($p<.001$)			($p<.001$)	

(No Interactions)

Note. Means calculated on the performances of subjects attending nine or more programming classes.

Table 35. MANOVA: Total Data Errors by Instructional Method and Practice Level

(Main Effects)

Target Items Included			Target Items Excluded		
Modeling	Inspection M.		Modeling	Inspection M.	
20.29 (N=51)	20.79 (N=47)		19.88 (N=51)	20.49 (N=47)	
	($p<.70$)			($p<.64$)	
Low Practice	High Practice		Low Practice	High Practice	
23.64 (N=42)	18.20 (N=56)		23.19 (N=42)	17.91 (N=56)	
	($p<.001$)			($p<.001$)	

(No Interactions)

Note. Means calculated on the performances of subjects attending nine or more programming classes.

series of two-way ANOVAs calculated on mean problems solved by practice level and complexity of instructional materials are summarized in Table 36. Again, students who practiced programming more than two hours outside class solved significantly more programming problems than those students who practiced fewer than two hours.

With one exception, all MANOVAs for solutions and error types by practice level and instructional materials' complexity indicate that high practice significantly ($p < .05$) increased solutions and decreased every category of error. Differences due to the effect of instructional methods were negligible. This was the case for the analysis along all three dimensions of the programming test.

Table 36. MANOVA: Total Solutions by Complexity of Training Materials and Practice Level

(Main Effects)

Target Items Included		Target Items Excluded	
Low Complex.	High Complex.	Low Complex.	High Complex.
61.89 (N=51)	56.75 (N=47)	46.45 (N=51)	69.96 (N=47)
($p<.16$)		($p<.17$)	
Low Practice	High Practice	Low Practice	High Practice
50.09 (N=42)	61.79 (N=56)	48.45 (N=42)	59.97 (N=56)
($p<.001$)		($p<.001$)	

Note. Means calculated on the performances of subjects attending nine or more programming classes.

[a]Target items excluded.

Gender and Problem-Solving Performance

The issue of equity between boys and girls in using and learning about computers was a concern throughout the experiment. The researcher, therefore, was interested in determining if any differences in problem-solving performance could be attributed to gender, particularly gender by grade level.

In order to detect any effects due to gender, a number of MANOVAs were performed on the salient variables in each of the three programming dimensions. The first set of MANOVAs was performed on solutions and errors by instructional method and gender. The results of the first MANOVA, total solutions by method of instruction and gender, are summarized in Table 37.

Although there were no significant main effects or interactions for instructional method, a main effect attributable to gender was detected. The young females in the treatment groups solved significantly fewer programming problems than did the young males, F (1, 4) = 9.33, p < .003. Moreover, data from MANOVAs of error categories provided evidence that female students made significantly more syntax errors, F (1, 94) = 8.49, p < .004, more control flow errors, F (1, 94) = 8.84, p < .004, and more data errors, F (1, 94) = 4.74, p < .03, than did their male counterparts. This difference in favor of males was significant (p < .05) for mean solutions and errors along

Table 37. MANOVA: Solutions by Instructional Method and Gender

(Main Effects)			
Target Items Included		Target Items Excluded	
Modeling	Inspection M.	Modeling	Inspection M.
61.27 (N=51)	56.98 (N=47)	59.47 (N=51)	55.11 (N=47)
($p<.26$)		($p<.24$)	
Males	Females	Males	Females
64.96 (N=52)	52.72 (N=46)	61.51 (N=52)	50.15 (N=46)
($p<.001$)		($p<.001$)	
(No Interactions)			

Note. Means calculated on the performances of subjects attending nine or more programming classes.

all three programming test dimensions. These general results also obtained when the treatment group scores were pooled according to complexity of instructional programs. Tabular summary of these results is found in Tables 44 to 51 (See Appendix B).

Observations throughout the experiment provided hints that sixth grade girls did not spend as much of their free time in the computer lab as other students, certainly far fewer hours than the fifth grade girls. A close inspection of the practice log, the results of which are summarized in Table 38, indicated that this was the general case.

Table 38. Comparison of Practice Level, Grade and Gender

	Grade/Gender[a]				Total	
	5M	6M	5F	6F	Male	Female
Practice Level #1: (0-2 hrs)	6	9	10	23	15 26.8%	33 57.9%
Practice Level #2: (2-20 hrs.)	15	26	15	9	41 73.2%	24 42.1%

Note. "5M" signifies 5th grade males etc.

[a] Practice level was independent of grade ($\chi^2 = 1.54$, $p > .20$), but was dependent upon gender ($\chi^2 = 10.83$, $p > .001$) and upon the grade/gender category ($\chi^2 = 17.10$, $p > .001$).

Practice level was found to be independent of both treatment group and grade level ($p > .20$), but dependent upon both gender and gender by grade. Fully 73% of the male students belonged to the high practice group as compared to only 42% of the female students; only 26% of the males belonged to the low practice group in contrast to nearly 58% of the females. A closer examination of Table 38 showed a disproportionally large number of sixth grade females (72%) in the low practice category. In contrast, large proportions of fifth grade females (71%) were in the high practice category.

The researcher also wanted to determine how female participation in programming activities outside the treatments was distributed among the four treatment groups.

Results of this analysis are summarized in Table 39. An initial impression was that there was a disproportionally large number of male students from the simple modeling and complex inspection methods group in the high practice category and an equally uneven grouping of female students from the simple inspection methods group in the low practice category. Data obtained from chi square tests of independence did not support this impression: practice was independent of treatment group, even when female students were considered separately.

Table 39. Comparison of Practice Level and Treatment Group

	Group[a]							
	S.Mod.		S.In.M.		C.Mod.		C.In.M.	
	M	F	M	F	M	F	M	F
Practice Level #1: (0-2 hrs)	3	5	4	11	5	8	3	9
Practice Level #2: (2-20 hrs.)	12	9	9	2	9	6	11	7

[a]Practice was independent of treatment group ($\chi^2 = 5.36$, $p > .10$), and was independent of treatment when the female subjects of each group are considered separately ($\chi^2 = 6.81$, $p > .05$).

Female students in all treatment groups solved fewer problems and made more syntax, control flow, and data errors on the programming test than did male students, but male students enrolled in the programming course spent more

time out of class practicing their programming skills than did female students. The fifth grade female students did show greater interest in programming outside the treatment sessions, but their participation did not offset the very low practice rates of the sixth grade females. The simple modeling and complex inspection method groups were marked by a disproportional number of high practice males. In contrast, the simple inspection method group included a large number of low practice females.

Grade Level and Problem-Solving Performance

The subjects in this experiment consisted of 46 fifth graders and 67 sixth graders. A goal of this experiment was to determine the effect of grade level on solving programming problems. It was predicted that sixth graders on the average would solve significantly more problems and would commit signficantly fewer errors in each category than would the fifth graders ($p < .05$). Assignment of subjects to groups was such that grade level was independent of treatment. In the analysis, student test scores were pooled into two grade level groups (fifth and sixth graders) to create the cell sizes required by MANOVA. One series of MANOVAs was performed on mean solutions and errors by grade level and instructional method. A second series of MANOVAs was performed on mean solutions and errors by grade level and the relative complexity of

programs used in instruction. A summary of these analyses is found in Tables 51 to 54 (See Appendix B).

Unlike practice, the grade level variable did not produce differences in total solutions along each of the three test dimensions. In considering solution means by instructional method and grade level, main effects for grade level were not significant at the prescribed level. Likewise, there were no main effects for grade level when the target items were subtracted out of the analysis, F (1, 94) = 2.75, $p < .10$. The main effects for grade level also were not found to be significant when total solutions were analysed by complexity of instructional programs and grade level, F (1, 94) = 2.72, $p < .10$.

No significant differences in mean total solutions were discernable for the problem task type or program complexity categories, but some differences were observed in the problem relatedness category. The sixth grade group displayed significantly higher means for solution of target problems, F (1, 94) = 6.42, $p < .02$, and equivalent problems, F (1, 94) = 3.96, $p < .05$ in the instructional method by grade level analysis. Interactions and main effects for instructional method were absent in both the grade level by instructional method and grade level by instructional materials analyses.

Grade level main effects were found through an analysis of syntax errors. The sixth grade group made

significantly fewer syntax errors in both the grade level by instructional method analysis, F (1, 94) = 3.95, p < .05, and grade level by complexity of instructional programs analysis, F (1, 94) = 4.12, p < .05, but this advantage was nonsignificant when the target problems were removed from consideration, F (1, 94) = 3.01, p < .08; F (1, 94) = 3.05, p < .08. In both sets of MANOVAs, the sixth grade group on the average made fewer syntax errors in attempting to solve problems involving simple statements, F (1, 94) = 4.60, p < .04; F (1, 94) = 4.63, p < .04. Sixth graders also committed significantly fewer syntax errors when attempting to solve target problems, F (1, 94) = 6.42, p < .02; F (1, 94) = 6.61, p < .02, and when attempting to solve identical problems, F (1, 94) = 3.96, p < .05; F (1, 94) = 3.95, p < .05.

Inspection of the target relatedness dimension also revealed several interactions. Two grade level by instructional method interactions were significant, suggesting that fifth graders in the inspection methods group made significantly more syntax errors in their attempt to solve isomorphic problems, F (1, 94) = 3.90, p < .05, and similiar problems, F (1, 94) = 4.46, p < .04. This difficulty with similar problems was shared by sixth graders in the modeling groups. In the syntax errors by grade level and complexity of instructional materials analysis, sixth graders, whose instruction centered upon

programs of relatively low complexity, made fewer syntax errors in attempting to solve equivalent problems, F (1, 94) = 6.17, $p < .02$. There were no significant main effects or interactions ($p < .05$) for grade level by instructional method or by materials complexity that involved either data or control flow errors.

Grade placement had an effect upon programming performance, but this effect was limited to spotting and avoiding syntax errors. Sixth graders were somewhat more adept than the fifth grade students at debugging syntax errors in certain limited situations. They were better able to avoid these errors in settings of "close" transfer or low complexity, that is, when such errors appeared in target and identical problems or in simple statements. Fifth graders in the inspection methods groups appeared particularly disadvantaged in avoiding syntax errors set in the context of isomorphic or similar problems, both involving the "far" transfer of problem-solving information from the target problems.

Summary

Participation in the computer programming course exerted a negligible effect on the participants' ability to solve nonroutine, mathematical word problems. Little skill development was observed from the IPST pretest to posttest, and no significant differences were observed among the

average number of problems solved by the four treatment groups and one control group. A significant, but moderate, positive correlation was found between the number of problems students solved on the IPST pretest and the number of problems they solved on the programming posttest. Moderate, negative correlations were also observed between IPST pretest solutions and the three types of programming errors measured by the programming test.

The other experiment was designed to assess the effect of two levels of two instructional methods on the solution of computer programming problems. Contrary to expectations, neither the instructional method nor program complexity factors produced significant differences in average group solutions. Students in each treatment solved as many programming problems as students in any other group. Nevertheless, it was found, through analyses of error patterns, that students trained with comparably less complex programs made fewer syntax errors than students trained with more complex programs.

A post hoc analysis was made of group problem-solving performance in the program task, problem relatedness, and program complexity categories of the programming test. Neither the method of instruction nor the complexity of programs used in instruction produced significant differences in the average number of problems solved in either the generation or interpretation of programming

problems. It was observed that instruction with models helped students acquire and transfer solution information from the target tests. Nevertheless, this ability to take advantage of the corrective feedback provided by the target test did not result in students showing a comparative advantage over other students in problem solving. The relative complexity of the instructional programs resulted in a more powerful influence on the solution of equivalent problems then did method of instruction.

In analyzing the program complexity category, it was observed that a "difficulty gradient" prevailed: program fragments consisting of simple statements were easier to understand than those characterized by looping and branching structures. No experimental treatment resulted in students having an advantage or disadvantage in solving either simple statements, linear blocks, forward branching, or looping structures. Furthermore, students in the modeling groups made fewer syntax errors in attempting to solve problems involving simple statements, and students trained with less complex programs committed fewer syntax, control flow, and data errors in attempting to interpret or generate more structurally complex programs.

The effects of practice, gender, and grade level on programming performance also were examined. Students who programmed over two hours outside the instructional

sessions solved more programming problems and made fewer errors than did those students with less than two hours outside practice. Female students in all treatment groups solved fewer programming problems and made more syntax, control flow, and data errors on the programming test than did male students. However, female students spent less of their free time programming computers than did male students. Grade placement also influenced programming performance, but its influence was limited to debugging. Sixth graders on the average were somewhat more adept than fifth graders in spotting and avoiding syntax errors, especially in situations of "close" transfer.

Chapter V.

CONCLUSIONS

The purpose of this study was to determine the specific effect of two methods of teaching the BASIC programming language upon solving programming problems and to assess the general effects of computer programming experiences upon the development of problem-solving skills in a related domain, nonroutine mathematical problem-solving. One hundred and thirty-nine fifth and sixth graders from Jackson Intermediate School (Jackson, Wyoming) participated in two, 13 week experiments to test a series of hypothesis related to these purposes. A four group, posttest experiment was designed to test the effect of instructional methods on programming skills. It was run simultaneously with the experiment to test the effects of programming instruction on mathematical problem solving. This latter experiment consisted of a five group, pretest-posttest control group design with four treatment groups and one control group.

Summary of Results

Participation in the computer programming course had no discernable effect on the participants' ability to solve

nonroutine, mathematical word problems. No skill development was observed in the IPST pretest to posttest period, and no significant differences were observed among the average number of problems solved by the four treatment groups and one control group. Significant associations were found between performance on the IPST pretest and the programming test. Students who performed well on the IPST pretest also did well on the computer programming posttest.

Contrary to expectations, neither the instructional method nor program complexity factors resulted in significant differences among the average solutions of each treatment group. Students in each treatment group solved as many programming problems as students in any other group. Analysis of error patterns, however, revealed that students trained with less complex programs made fewer syntax errors than students trained with more complex programs.

Post hoc analyses were made of group problem-solving performance related to the program task, problem relatedness, and program complexity variables of the programming test. Neither the method of instruction nor the complexity of programs used in instruction resulted in significant differences in the success with which subjects solved problems involving either interpreting or generating programs. However, instruction with models resulted in students acquiring and transferring solution information

from the target tests to the main test, although this ability to use the corrective feedback provided by the target test did not give students a comparative advantage over other students in problem solving. The relative complexity of the instructional programs resulted in a more powerful influence on the solution of equivalent problems then did method of instruction.

In analyzing the program complexity category, it was observed that a "difficulty gradient" existed: program fragments consisting of simple statements were easier to understand than those characterized by looping and branching structures. No experimental treatment resulted in students having an advantage or disadvantage in solving either simple statements, linear blocks, forward branching, or looping structures. However, instructional factors did influence debugging. Students in the modeling groups made fewer syntax errors in attempting to solve problems involving simple statements, and students trained with less complex programs committed fewer syntax, control flow, and data errors in attempting to interpret or generate more structurally complex programs.

The effects of practice, gender, and grade placement on programming performance were also examined. Students with more than two hours programming practice outside the instructional sessions solved more programming problems and made fewer errors than did those students with less than

two hours outside practice. Female students in all treatment groups solved fewer programming problems and made more syntax, control flow, and data errors on the programming test than did male students. Female students, however, spent less of their free time programming computers than did male students. Grade placement also had an effect upon programming performance, but it was limited to spotting and avoiding syntax errors. Sixth graders on the average were somewhat more adept at debugging these errors than fifth graders, especially in situations of "close" transfer.

Discussion of Results

The transfer question. Problem solving skills consist of procedures for transforming, arranging or perceiving patterns among the elements of a problem space. These procedures are guided and constrained by an understanding of the domain in which the problem space is set. A fundamental question in teaching problem-solving skills is whether general problem-solving strategies can be divorced from specific subject-matter content and can be taught as a set of general planning procedures applicable to a wide variety of domains. If a set of broadly transferable strategies, or heuristics, can be taught and learned, it might be worthwhile to establish general problem-solving courses at all levels of schooling. If the human capacity

for transfer is limited or if these general strategies are not widely applicable, our resources might be better spent concentrating on specific subject matter and developing domain-specific problem-solving skills.

This question of domain-specific and domain-independent problem-solving skills was addressed by this study, first, by assessing the effect of problem-solving instruction in the domain of computer programming upon problem solving in the domain of mathematics and, second, by estimating the span of transfer from specific programming problems to those of varying degrees of similarity. The experimental data outlined in Chapter IV provided no evidence that the general problem-solving skills learned in the computer programming classes resulted in students having any comparative advantage in solving mathematical word problems of a nonroutine nature. Data from the second experiment, however, supported the conclusion that students in the modeling groups were able to acquire information in a test-study situation and transfer it to solve closely related problems in a subsequent testing situation. The transfer of solution information occurred within the domain of computer programming and did not result in students in the modeling groups having any comparative advantage over other students in solving programming problems.

The absence of general transfer effects in this experiment was not suprising. The domain-independent heuristics taught in the programming course are "weak methods." Their strength is their potential usefulness in a wide variety of domains; their weakness is their abstractness and, hence, the difficulty of actually applying them to specific problems in particular domains. Schoenfeld (1980) cautioned that, although stated as simple maxims, general problem-solving strategies are in reality very complex and, for the uninitiated, difficult to apply to new situations without substantial instruction. Because the instructional treatments did not teach for transfer from the programming domain, one cannot reasonably expect young students to infer that these general strategies can be used to solve problems in a related domain such as mathematics.

Teaching for transfer of general strategies from programming to mathematical problem solving was not included in the instructional treatments. Teaching for transfer would have taken a great deal of time. It also might have resulted in two sets of parallel, but essentially domain-specific, strategies that would neither reinforce each other or generalize to other domains. The association between student performance of the programming test and on the Iowa Problem Solving Test (IPST) provided some tentative support for this conjecture.

The relatively high association between performance on the math pretest and performance on the computer programming posttest was a clue that some general problem-solving skills might undergird exceptional performance in both domains. Exceptional programming performance was most highly correlated with performance on the "looking backward" subtest of the IPST. The skills associated with "looking backward" involve perceiving and analyzing the schematic structure of word problems and relating them to similar problem statements. This ability to perceive "problem type" schemas also is a skill implicated in understanding computer program specifications. Nevertheless, students in the inspection method groups received specific training in this skill, but did not demonstrate any appreciable improvement in their ability to solve math word problems. Training to improve the ability to recognize the schemas of computer program specifications clearly did not automatically improve the ability to recognize the schemas of mathematical word problems.

Transfer of problem-solving information was clearly evident within the domain measured by the programming test. The modeling, but not the inspection method, groups were able to transfer solution information from the target test episode and use this information to improve their solving of identical and equivalent problems. This is not to say

that transfer of solution information was not involved in the performances of students in the inspection method groups. A useful distinction can be made between transfer from "local" exposure to a target pretest and the "global" transfer of knowledge acquired during the entire course of study. All but the very lowest performing students were able to transfer globally the programming skills learned during the course to assist them on the programming test, but only students in the modeling groups were able to exploit the local opportunity presented by the target pretest.

Why did local transfer not enable the modeling students to demonstrate a comparative advantage in solving programming problems? It may have, but in ways that were not measured by the programming test. Reed, Ernst, and Banerji (1974) found that the primary effect of transfer between related problems was to reduce the time needed to arrive at a solution. Since solution time was measured neither by the programming test nor by the student protocols, differences in the speed and rate with which students solved problems went undetected.

The ability to transfer and utilize information acquired in local situations such as study problems, pretests, or recently solved problems also may depend on the way students were trained to access solution procedures. Accessing can be cued by perception of information in the

problem statement that suggests an analogy to previously solved problems, by perception of conditions in the problem statement that automatically trigger appropriate actions, or by explicit instructions to retrieve relevant information. Since students were not informed of the relationship between the target and main test problems, transfer probably occurred either by analogical reference to the target problems or by condition-triggered actions.

Accessing by means of concrete analogies to previously solved problems or to previously observed solutions is characteristic of instruction by modeling. One would expect students in the modeling groups, therefore, to be more sensitive to related problems and to have relied more on the target problems. Instruction in inspection methods, by contrast, assists students to develop production systems, sets of solution plans in the forms of chunks that are triggered by patterns of problem statements. Students trained in these methods can be expected to evoke general solutions in response to idiosyncratic problem conditions and to rely less on related or recently solved problems unless specifically cued to do so.

The fact that students in the simple modeling group solved significantly more equivalent problems may indicate the comparative advantage of analogical methods in solving closely related problems. No method of accessing solution procedures resulted in any group of students having a

special advantage in solving remotely related problems or having an advantage in solving unrelated sets of programming problems.

Analogical reference and production systems are examples of "strong," domain-specific problem-solving methods. Heuristics and other "weak" methods may be, as Glaser (1980) and others suggest, more typical of novice rather than expert problem-solving performance. Heuristic search, the painstaking selection and planning of solution procedures, may be a poor substitute for an extensive knowledge of the domain in which the problem is set, a knowledge that enables experts to rapidly, almost automatically, evoke well-organized, approximate solutions when confronted with problems. The bottom line is the efficiency of teaching general problem-solving methods, particularly when these skills appear constrained by the domain in which they were acquired and generalize outside that domain only with considerable instruction. The results of this study bolster the argument in favor of teaching domain-specific problem solving methods. Nevertheless, a worthy subject of further research is the efficacy of teaching domain-specific methods as special cases or applications of general problem-solving methods.

Method of instruction. The second goal of this study was to determine the effect of various instructional

methods and materials on the specific problem-solving operations of program comprehension, composition and debugging that define computer programming. Contrary to expectations, it was not instructional methods but practice time and gender that had the largest and most consistent effect upon increasing solutions and reducing errors. Method of instruction did not result in significant differences in group problem-solving performance.

At least four explanations were adduced for the lack of differences among treatment group solutions. The computer programming course, for example, may not have been of sufficient duration to produce discernable differences among group performances. This conclusion certainly was supported by the powerful effect that additional programming practice had on increasing solutions of programming problems. Extending the length of the experimental treatments might have produced different results, but it also would have increased the likelihood of contamination by intervening variables.

Another compelling explanation was that the students learned equally well whether taught by modeling or inspection methods or whether with simple or complex instructional programs. This was a reasonable inference considering the lack of differences among the average solutions of each treatment group. However, differences among groups were evident in the post hoc analyses of

problem relatedness, problem complexity, and error patterns. Furthermore, if all treatments were independent and had equally additive effects, one would find normal distributions of solutions, not the relatively large positive skewness that characterized the distributions of the inspection method groups.

An equally plausible explanation for the lack of differences among solution means was "noise" that might have been generated by varying amounts of programming practice. The quantity of practice time was equally distributed among the groups, but there was no guarantee that quality of this practice was equally distributed. For example, the four top-performing students in the inspection method groups programmed elaborate projects and spent far more time at the keyboard than their counterparts in the modeling groups. Equally significant were the low practice rates for sixth grade female students. Practice and gender effects probably reflected the influence of motivation and social reinforcement. Practice time, therefore, might have interacted with other factors, such as gender, to create random and multiplicative effects which may be reflected in the positively skewed solution distributions and in the lack of differences among treatment groups.

The absence of significant differences among the solution means may also indicate that method of

instruction has a greater effect at certain phases of learning sequence than others. Learning a conceptually complex domain such as computer programming may be similar to learning complex motor skills. Two phases of stimulus control seem to be operating in this kind of learning. The first phase is a noisy, inaccurate set of processes in which the learner acquires a conceptual understanding of the domain, but very few efficient problem-solving procedures. Differences among instructional methods would be very evident during this skill acquisition period. The second phase is a refinement process in which the learner practices and applies his skills in actual problem solving. Large variances attributable to comparative effectiveness of instructional methods are diminished during the refinement stage, but negatively skewed distributions may still result because of the willingness of a few individuals to spend the time and work necessary for exceptional performance.

Differences attributable to method of instruction would be difficult to detect during the refinement stage, and this may have been the very time during which the programming test was administered to the students. Perhaps, very clear differences would have been detected had the students been tested at regular intervals during the experimental treatments. However, individuals may pass from the acquisition to the refinement stages at very

different times. If this is the case, a single-subjects, rather than a group, research design would have been a more appropriate choice for this study.

Post hoc analysis of error patterns. The instructional treatments did not result in differences among group solutions, but they made a difference in debugging performance. Students whose instruction employed relatively simple programs made significantly fewer syntax errors than students whose instruction used relatively complex programs, particularly those in the complex inspection methods group. With the exception of data errors involving assignment to incorrectly labelled variables, syntax errors are the most perceptually obvious and readily detectable error type. Avoidance of syntax errors, therefore, may be the most apt measure of debugging skill in a paper-and-pencil test since data and control flow errors are best and most often detected when the program is actually run rather than when simply read. Debugging itself may be the most important skill in taking a multiple choice, programming test such as that used in this study. Choosing the best alternative answer often can be best accomplished by eliminating those programs or program descriptions with the most obvious and readily detectable errors.

Post hoc analysis of complexity. The contrast between group differences in debugging performance but no differences in solutions held throughout all three dimensions of the programming test. Equally pervasive was the contrast between the strong effects of program complexity and the relatively weak effects of instructional methods. When the results of the programming test were supplemented by an analysis of the protocols, interviews, and controlled observations of the students involved in this study, a number of tentative explanations could be made for the pervasive influence of program complexity on the test results.

Program complexity, rather than span of transfer, proved to be the strongest predictor of problem difficulty for all groups. The four complexity categories were found to comprise a "difficulty gradient" in which problems involving simple statements were easier to solve than those involving block structures which in turn were less difficult than those involving forward branching and looping structures.

Though the difficulty gradient existed, it did not perfectly distinguish between each complexity category and varied with method of instruction. The performances of students in the modeling groups tended to reflect the separate degree of difficulty within each complexity category. For inspection method students, the difficulty

gradient tended to be a simple dichotomy between simple statements-linear blocks and forward branching-looping. Forward branching and looping structures did not function as two separate degees of difficulty for either modeling or inspection method students, perhaps because both are special cases of branching.

No experimental treatment resulted in any group of students demonstrating an advantage or disadvantage in solving problems involving simple statements, block or branching structures. This applied whether students were interpreting or generating programs. There were differences in debugging performances. Students in the modeling groups committed fewer syntax errors when attempting to solve problems involving simple statements, but complexity of instructional materials rather than method of instruction resulted in the greatest effect on debugging complex programs. Paradoxically, students trained with relatively less complex programs tended to make fewer syntax and data errors when attempting to solve problems involving relatively more complex programs.

The superiority of presenting key programming concepts in the context of minimally complex programs or program fragments probably is related to the "chunking" phenomenon that occurs in the interpretation, generation and debugging of programs. The interpretation dimension of the programming test measures comprehension, the problem-

solving skill involving the induction of a meaningful pattern of relationships among the syntactical elements of a program. These patterns tend to take the form of functional units known as chunks.

Researchers have suggested that novices' chunks tend to be smaller, more unstable, and less well organized than those of expert programmers. Beginning programmers, therefore, will decode more complex programs in smaller chunks, often in a command-by-command, "bottom-up" fashion similar to the word-by-word decoding of beginning readers. This strategy would hinder comprehension by increasing the number of chunks well beyond the limit (5 to 9) that can be held in short-term memory. Programs containing fewer and less complex structures would be easier to chunk into units of larger size, thereby increasing decoding efficiency and reducing the number of chunks that have to be held in short-term memory. Training with relatively simple programs, paradoxically, might have resulted in students developing a repertoire of larger, better organized chunks and the habit of reading and debugging programs in a more efficient, "top-down" manner.

Top-down debugging may take the form of a "weak" strategy known as "subproblem decomposition." It involves dividing a defective or incomplete program into subproblems or subtasks and attacking each subproblem or subtask separately. In reviewing student protocols, it was

observed that students who fell in the last two quartiles of the programming test employed a cursory, unsystematic, command-by-command debugging technique. Higher performing students were observed using a subproblem decomposition strategies even when debugging or completing very simple programs. Furthermore, they divided programs into subproblems based on chunks of functionally related code, varying in size from two or three simple statements to small, well-organized block or branching procedures.

The debugging protocols were too few and unrepresentative to determine if students trained with low-complexity programs used these top-down debugging strategies more often than other students. However, it may be inferred from the results of the programming test that students trained with less complex programs did employ more powerful debugging strategies than students trained with more complex programs. Students in the low-complexity pooled groups made significantly fewer syntax and control flow errors when solving interpretation problems and fewer data errors when solving generation problems. They were particularly successful in avoiding syntax and data errors when attempting to solve problems involving block and branching structures. This comparative advantage might be explained by the chunking effect, but also by the ease with which subproblem decomposition techniques could be taught using relatively simple programs. Small, simple programs

can be quickly divided into subprocedures, thereby greatly reducing the amount of code that has to be inspected.

Analysis of a small number of protocols gathered on students composing programs provided some suggestive, but very tentative, evidence that students trained with simple programs displayed greater fluency in generating chunks of program statements and greater flexibility in arranging them into working programs. Results of the programming test were somewhat contradictory and inconclusive. Scattered differences in debugging performance were noted, but differences among treatment group solutions were not significant.

Post hoc analysis of program task type data. The hazy results from the generation portion of the programming test probably were due to the difficulty of attempting to estimate program composition skills using a paper-and-pencil, multiple-choice test. Solving the test's program generation problems required only a portion of the skills demanded by actual program composition. Actual program composition requires the ability to interpret problems statements, skill in generating program code fluently, and the planning abilities involved arranging and shaping the code into a possible solution. Because of the test's multiple-choice format, students were not required to produce or arrange the code in working form as

some were when composition protocols were taken. Success in solving generation problems had more to do with the interpretation of problem statements and understanding the possible solutions presented as multiple choices. Consequently, results of the programming test provided little help in answering questions about fluency and planning in program composition. An alternative would have been to require students to actually write programs in response to problem statements, but this would introduce the additional difficulty of scoring programs in ways that would be consistent with the data generated by the multiple-choice portions of the test.

More interesting results came from the problem-solving strategies revealed by a series of think-aloud protocols conducted by this researcher with twelve students. The students were asked to "think-aloud" while solving equivalent problems alternately with paper and pencil and at the computer keyboard. It was inferred from these protocols that the students employed somewhat "stronger" composition and debugging strategies when writing programs in a paper-and-pencil test than when actually composing programs at a computer keyboard.

When solving problems interactively, the students used a weak form of the "generate and test" strategy, generating very specific solution candidates and then relying upon the BASIC interpreter's error messages to provide corrective

feedback. When composing with paper and pencil, the students were observed to rely largely on a "hypothesis and match" strategy in which general, schematic solutions were generated and then systematically debugged and shaped into a final solution. The "hypothesis and match" strategy resulted in more correct solutions, fewer syntax errors, and fewer control flow errors. Although numbers of students involved in these episodes were too small to test the significance of differences between mean solutions and errors, the results suggest that removing students from the keyboard when composing programs might encourage more sophisticated and efficient planning of solutions.

Post hoc analysis of gender effects. Female students solved significantly fewer programming problems and made significantly more syntax, control flow, and data errors than did their male counterparts. Likewise, female students spent significantly less of their time outside of the instructional sessions working on their projects or on other programs. Nearly three-quarters of the sixth grade girls spent less than two hours in these activities while the same proportion of male students spent between three and twenty hours programming outside of class. Since there was a consistently strong effect of practice time resulting in increased problem solutions and decreased programming errors, male students probably performed better than female

students because the boys spent more time practicing their programming skills.

The fifth grade girls expressed greater interest in computing and spent more time programming outside of the instructional sessions than did the sixth grade girls, but their participation did not offset the very low practice rates of the sixth grade girls. This contrast might have indicated a shift of attitudes toward computing from one grade to the next. More deeply, this contrast may have reflected certain social pressures placed on girls of this age. The teachers and teachers' aides, who monitored the computer lab during the school day, reported that many girls displayed self-conscious behaviors when groups of boys were in the lab. Often, the girls were the objects of a great deal of unsolicited programming advice and authoritative judgements of their programs, most of which came from their male counterparts. It was observed that the girls, who spent time working on their projects, did so after the instructional sessions when most of the boys had gone home.

Recommendations for Further Research

Cognitive psychologists have tried to explain human problem solving by constructing taxonomies and models of human problem-solving behavior. These typologies and models are largely, but not exclusively, based on evidence

obtained from response times, think-aloud protocols, clinical interviews, and free recall activities. The implicit goal of these efforts is descriptive, to describe how people solve problems. The educator's agenda is prescriptive, to teach people how to improve their problem-solving skills. The goal of this study, therefore, was not to construct a model of programming behavior, but rather to build and evaluate methods of teaching computer programming as a set of problem-solving skills. This researcher did not feel obligated to evaluate his teaching methods with the observational techniques used by cognitive psychology. Although protocols were taken and clinical interviews were conducted, these techniques were used to clarify and extend the analyses of the programming test.

Nevertheless, some changes in problem-solving behavior could not be captured using the paper-and-pencil tests used in this study. This was evident in the kinds of subtle shifts in student problem-solving strategies revealed by the think-aloud protocols. As Schoen and Ohemke (1980) pointed out, multiple-choice, paper-and-pencil tests cannot reveal the kind of strategies employed by the problem solver. They can neither illuminate the kinds of fluency and planning demanded by such forms of constructive search as program composition, nor assess the efficiency of these strategies. On the other hand, protocol analysis and the other tools of the cognitive psychologist involve a

detailed analysis of each subject's problem-solving behavior. Not only are protocol gathering and analysis time-consuming, but the data captured do not always have the same meaning across subjects. Further progress in designing and evaluating methods of teaching problem-solving skills depends upon the development of instruments that combine the administrative ease of paper-and-pencil tests and the individual emphasis of think-aloud protocols with statistical techniques for determining individual differences in cognition. This is a promising area for further research.

Further investigations should also be made of the relationships between domain-specific and domain-independent problem-solving strategies. Despite the failure of programming instruction to enhance mathematical problem-solving skills, there may have been common, or parallel, skills responsible for superior performances on the IPST pretest and the programming posttest. Perhaps, exceptional performance in both domains is due to heuristics of a humbler kind. Hunt (1983) reported research that contrasted the ways good and poor problem solvers took multiple-choice tests. Poor problem solvers quickly searched for the most plausible alternative; good problem solvers spent their time trying to understand the problem and frame the best answer irrespective of the alternatives. Those wishing to design problem-solving

instruction need to investigate these simple, task-specific heuristics in an effort to understand the more complex, domain-independent strategies.

Educational Implications

A number of general, instructional recommendations are supported by the results of this study. In any method of teaching computer languages, three salient features of the programming environment need to be dealt with: the unique demands of programming's different phases, the transfer of programming knowledge from instruction to actual problem solving, and the "psychological difficulty" of programs and program specifications used in instruction and evaluation. In this study, instruction and evaluation were based upon a taxonomy of programming tasks (comprehension, composition, and debugging), a set of procedures for teaching and evaluating transfer to sets of related problems, and a theory of problem difficulty predicated upon the computational complexity of different programming structures.

Three specific recommendations are supported by the results of this study. If the goal is to train students to recognize, write, or debug key programming structures, set them in the least complex code possible. If the goal is to enhance a method of programming instruction or compensate for its shortcomings, provide for as much structured

programming practice as possible. Finally, there may be invidious social attitudes at work, particularly among preadolescent programmers, that may confound the effectiveness of instructional methods. Allowing young females a place and time to program computers in absence of young male collegues might increase female enthusiasm for computing, their confidence in their skills, and their concentration upon the task at hand.

Few effects could be attributed to instructional methods. This does not mean that any method will do. In the first place, instructional methods must be evaluated by other criteria such as student attitude and ease of teaching. Furthermore, some kinds of instruction may make great differences during the initial, acquisition stages of learning, but not later when freshly learned skills are being practiced. Finally, different methods developed and tested during this study might be more appropriate for certain kinds of teaching tasks and might be more effective at certain points in the instructional sequence. Both simple modeling and simple inspection methods can be used most appropriately in computer programming courses for upper elementary or junior high school students and might be most effective when used together. The reciprocal advantages of both methods can be illustrated in terms of Adelson's (1984) useful distinction between representations for computer programs that involve what the program does

and those that deal with how a program does it. Her distinction was between understanding based on a procedural representation of programs and understanding based on a declarative representation of programs.

Modeling computer programs with manipulative devices gave students a detailed, procedural representation of how a program functions. The simple models method seemed most useful for introducing basic programming concepts and structures and for teaching students how to read programs, particularly programs with multiple transfers of control. Simple modeling techniques, therefore, were more appropriate in the introductory phases of the teaching sequence. Modeling large, complex programs should not be used to introduce new programming concepts, but such modeling was particularly helpful in presenting procedures with many transfers of control, nested loops, or numerous data operations. Complex modeling techniques, therefore, were most effectively used in teaching sorting, searching, and animation routines.

Simple inspection methods, on the other hand, were useful for giving students a declarative representation of what a program does, its output specification. It was a particularly effective method of training students to quickly spot program bugs, to identify, name, or explain stereotypical program structures or problem statements, and to write programs. Since these are somewhat more advanced

skills, inspection methods were most effectively used somewhat later in the teaching sequence. Complex inspection methods, however, proved so unwieldy to teach and produced such consistently low results that it can not be recommended as an instructional strategy.

Final Summary

Reading, composing, and debugging computer programs entail problem solving in a conceptually complex domain. Many educators have viewed computer programming not only as a worthy subject of study by grade school children but also as a promising means of teaching general problem-solving skills. Cognitive psychologists and workers in the field of artificial intelligence have constructed theories of human problem solving which could become a rich source of instructional techniques for teaching both general and domain-specific problem-solving skills to young children.

One goal of this study was to determine the effectiveness of computer programming as a vehicle for teaching general, domain-independent, problem-solving skills to grade school children. The experiment used to accomplish this goal consisted of a five group, pretest-posttest design with one control group and four treatment groups. It was found that learning techniques for solving computer programming problems had no relationship to the ability of 113 fifth and sixth graders

to solve nonroutine, mathematical word problems. However, significant associations were found between performance on the computer programming test and performance on the Iowa Problem Solving Test.

The second goal of the investigator was to assess the effect of two instructional strategies, both derived from the fields of cognitive psychology and artificial intelligence, upon children's solutions to computer programming problems. These strategies were called modeling and inspection methods. Modeling and inspection methods are instructional strategies for promoting coherent and efficient representations of programming problems. Two levels of modeling and inspection methods, each based on the computational complexity of the programs used in instruction, were devised. The effectiveness of these four methods was assessed by means of a four group, posttest experiment. The criterion variable for this experiment was a researcher-designed test that measured three dimensions of computer programming: problem relatedness, program complexity, and programming task type.

Neither instructional method (modeling or inspection methods) nor the complexity of programs used in instruction (simple or complex) resulted in significant differences in the solution of programming problems. Students in each treatment group solved as many problems as students in any other group. However, post hoc analysis of error patterns

revealed that students trained with comparatively less complex programs made fewer syntax errors than students trained with more complex programs. It also was observed that instruction using models resulted in students taking advantage of the corrective feedback from the target test, but this solution information did not result in their solving more problems than students in the inspection method groups. Computational complexity was found to result in diminished probability of solving programming problems, but none of the instructional methods resulted in a comparative advantage for students in solving problems involving complex programs. Nevertheless, students trained with less complex programs were better able to detect certain classes of errors embedded in the more complex programs. Practice, gender, and grade level were the factors most related to problem-solving performance. Students who practiced their programming skills outside class solved significantly more problems than did other students, and six grade students were more adept than fifth graders at recognizing and avoiding syntax errors.

The results of this study bolster the argument in favor of teaching domain specific, rather than general, problem-solving strategies. The absence of general transfer effects in this experiment was ascribed to the abstract character of general problem-solving strategies, which made them difficult to apply to specific problems.

Teaching for transfer of strategies from one domain to another is necessary, but there is no evidence that this effort would be any more efficient or effective than teaching domain-specific skills. Local transfer of solution information from the target to the main portion of the programming test did occur, but transfer resulted in no special advantages in solving programming problems. Other transfer effects may have taken place, but were undetected because the programming test neither measured solution times nor revealed solution strategies.

The lack of difference among the average number of problems solved by each group was explained in several ways. The instructional treatments may not have been of sufficient duration. Students may have learned equally well, whether taught by modeling or inspection methods or whether with simple or complex programs. Potential differences also may have been drowned by "noise" produced by the interaction of factors such as practice and gender. Subtle social pressures, for example, may have inhibited female students from programming outside class. Finally, methods of instruction may have had their greatest effect during the initial stages of instruction before other factors such as practice came into play.

The use of minimally complex programs in instruction did make a difference in debugging performance. Debugging skills may be the greatest contributor to success in

multiple choice, programming tests in which correct choices often depend on a negative strategy of eliminating obviously bugged programs. Because of these and other limitations in the programming test, think-aloud protocols were conducted with a small number of students. Analysis of these protocols suggested that training with relatively simple programs may have developed in students repertoires of larger, better organized chunks and the habit of reading, composing, and debugging programs with more efficient, "top-down" strategies. However, the limitations of think-aloud protocols rendered these conclusions somewhat tentative. Further progress in developing methods for teaching problem solving may require measures that combine the statistical integrity of paper-and-pencil tests with the detailed analysis of individual problem-solving behavior that is possible with think-aloud protocols and clinical interviews.

Because of the pervasive effects of computer program complexity and programming practice on learning and testing behaviors, it was suggested that computer programming instructors should employ the least complex code possible to illustrate key programming concepts and should provide many opportunities for structured programming practice outside of class. However, precautions must be taken that young female students are not discouraged from availing themselves of these opportunities because of invidious

social attitudes and practices. Although program complexity and programming practice were dominant influences upon test performance, selection of appropriate method of instruction is important. Simple and complex modeling techniques and simple inspection methods can be profitably used together in instruction, provided they are employed at a proper time in the teaching sequence and in the teaching task for which they are best suited.

BIBLIOGRAPHY

BIBLIOGRAPHY

Adelson, B. Problem-solving and the development of abstract categories in programming languages. Memory and Cognition, 1981, 9(4), 422-433.

Adelson, B. When novices surpass experts: the difficulty of a task may increase with expertise. Journal of Experimental Psychology, 1984, 10(3), 483-495.

Anderson, J. R. Verbatim and propositional representation of sentences in immediate and long-term memory. Journal of Verbal Learning and Verbal Behavior, 1974, 13, 149-162.

Anderson, J. R. Retrieval of information from long-term memory. Science, 1983, 220, 25-30.

Anderson, R. C. Schema-directed processes in language comprehension. In Alan M. Lesgold, James W. Pellegrino, Sipke Fokkema & Robert Glaser (Eds.), Cognitive psychology and instruction. New York: Plenum Press, 1978.

Arblaster, A. Human factors in the design and use of computing languages. International Journal of Man-Machine Studies, 1982, 17, 211-224.

Atwood, M. E. & Ramsey, R. H. Cognitive structures in the comprehension and memory of computer programs: an investigation of computer program debugging. TR-78-A21, U.S. Army Research Institute for the Behavioral and Social Sciences, Alexandria, Virginia, August, 1978.

Bayman, P. & Mayer, R. E. Instructional manipulation of users' mental models for electronic calculators. International Journal of Man-Machine Studies, 1984, 20(2), 189-199.

Barnett, J. The study of syntax variables. In Gerald Goldin & C. Edwin McClintock (Eds.), Task variables in mathematical problem-solving. Columbus, Ohio: The Ohio State University, 1979. (ERIC Clearinghouse for Science, Mathematics, and Environmental Education).

Bell, D. A. An empirical study of formal procedure specification. Contemporary Educational Psychology, 3, 1978, 163-168.

Bhaskar, R. & Simon, H. Problem solving in semantically rich domains: an example from engineering thermodynamics, Cognitive Science, 1977, 1, 193-215.

Bower, G. H. A selective review of organizational factors in memory. In Endel Tulving & Wayne Donaldson (Eds.), Organization of memory. New York: Academic Press, 1972.

Bower, G. H. Experiments in story understanding and recall. Quarterly Journal of Experimental Psychology, 1976, 28, 511-534.

Brooks, L. & Dansereau, D. F. Effects of structural schema training and text organization on expository prose processing. Journal of Educational Psychology, 1983, 75(6), 811-820.

Brooks, R. Towards a theory of the cognitive processes in computer programming. International Journal of Man-Machine Studies, 1977, 9(6), 737-751.

Brooks, R. Studying programmer behavior experimentally: the problems of proper methodology. Communications of the ACM, 1980, 23(4), 207-213.

Brown, C. M. Computer vision and natural constraints, Science; 1984, 224(4655), 1299-1305.

Brownell, W. When is arithmetic meaningful? Journal of Educational Research, 1945, 38, 481-498.

Brownell, W. Meaning and skill: maintaining the balance. Arithmetic Teacher, 1956, 3, 27-43.

Brownell, W. & Sims, V. M. The nature of understanding. In The measurement of understanding. (Forty-fifth yearbook of the national society for the study of education, Part I, 1946).

Carnine, D. W. & Becker, W. C. Theory of instruction: generalization issues. Educational Psychology, 1982, 2)3-4), 249-262.

Catania, A. C. Learning. Englewood Cliffs, N.J.: Prentice-Hall, 1979.

Chapanis, A. & Kelly, M. J., Limited vocabulary and natural language dialogue. International Journal of Man-Machine Studies 1977, 9(4), 479-502.

Chase, W. G. & Simon, H. A. Perception in chess, Cognitive Psychology, 1973, 4, 55-81.

Chen, E. T. Program complexity and programmer productivity. IEEE Transactions on Software Engineering, 1978, SE-4(3), 187-194.

Clark, H. H. Inferring what is meant. In W. J. M. Levelt & G. B. Flores d'Arfais (Eds.), Studies in the perception of language. New York: John Wiley & Sons, 1978.

Collins, A., Warnock, E. H., Aiello, N., & Miller, M. L. Reasoning from incomplete knowledge. In D. G. Bobrow & A. Collins (Eds.), Representation and understanding: studies in cognitive science. New York: Academic Press, 1975.

Coombs, M. J. & Alty, J. L. Acquiring a first computer language: a study of individual differences. In M. J. Coombs & J. L. Alty (Eds.), Computing skills and the user interface. London: Academic Press, 1980.

Coulter, N. S. Software science and cognitive psychology, IEEE Transactions on Software Engineering, 1983, SE-9(2), 166-171.

Curtis, B., Sheppard, S. B., Milliman, P., Borst, M. A., & Love T. Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics. IEEE Transactions on Software Engineering, 1979, SE-5(2), 96-104.

Curtis, B., Sheppard, S. B., & Millimam, P. Third time charm: stronger prediction of programmer performance by software complexity metrics. Proceedings of the 4th International IEEE Conference of Software Engineering, 1979, 356-360.

Davis, R. B. Complex mathematical cognition. In H. P. Ginsburg (Ed.), The development of mathematical thinking. New York: Academic Press, 1983.

DeLaurenties, E. Learning by interactive programming: Microcomputer applications. Educational Technology, 1980, 20, 10-14.

Di Persio, T., Isbister, D., & Schneiderman, B. An experiment using memorization/reconstruction as a measure of programmer ability. International Journal of Man-Machine Studies, 1980, 13(3), 339-354.

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, 1981, 14(3), 237-249.

Egan, D. E. & Schwartz, B. J. Chunking in recall of symbolic drawings. Memory and Cognition, 1979, 14(3), 149-158.

Englert, C. S. & Hiebert, E. H. Children's developing awareness of text structures in expository materials. Journal of Educational Psychology, 1984, 76(1), 65-74.

Estes, W. K. The information processing approach to cognition: a confluence of metaphors and methods. In W. K. Estes (Ed.), Handbook of learning and cognitive processes (Vol. 5, Human information processing). Hillsdale, N. J.: Laurence Erlbaum Ass., 1978.

Ferguson, G. A. Statistical analysis in psychology and education (5th ed.). New York: McGraw-Hill, 1981.

Feurzig, W. & Nickerson, R. S. Microcomputers in education. Department of Health, Education, and Welfare, National Institute of Health, October 1981 (ERIC ED 208-901).

Fillenbaum, S. Or: some uses. Journal of experimental psychology, 1974, 103(5), 913-921.

Fillenbaum, S. If: some uses. Psychological research, 1975, 37(3), 245-260.

Fredrickson, C. H. Structure and process in discourse production and comprehension. In M. A. Just & P. A. Carpenter, Cognitive processes in comprehension. Hillsdale N. J.: Laurence Erlbaum Ass., 1977.

Galambos, J. A. & Rips, L. J. Memory for routines. Journal of Verbal Learning and Verbal Behavior, 1982, 21, 260-281.

Gick, M. L. & Holyoak, K. J. Analogical problem-solving. Cognitive Psychology, 1980, 12(2), 306-355.

Ginsburg, H. P., Kossan, N. E., Schwartz, R., & Swanson, D. Protocol methods in research on mathematical thinking. In H. P. Ginsburg (Ed.), The development of mathematical thinking. New York: Academic Press, 1983.

Glaser, R. Relationships between aptitude, learning, and instruction. In R. Snow, P. Federico, & W. E. Montague (Eds.), Cognitive process analyses of learning and problem solving (Vol. 2, Aptitude, Learning & Instruction). Hillsdale, N. J.: Laurence Erlbaum, 1980.

Goldin, G. A. Structure variables in problem-solving. In G. A. Goldin & C. E. McClintock (Eds.), Task variables in mathematical problem-solving. Columbus, Ohio: The Ohio State University, 1979. (ERIC Clearinghouse for Science, Mathematics, and Environmental Education).

Goldin, G. A. & Caldwell, J. H. Syntax, content, and context variables examined in a research study. In G. A. Goldin & C. E. McClintock (Eds.), Task variables in mathematical problem-solving. Columbus, Ohio: The Ohio State University, 1979. (ERIC Clearinghouse for Science, Mathematics, and Environmental Education).

Goldstein, I. P. & Miller, M. L. Structured planning and debugging: a linguistic theory of design (AI Memo 387, Artificial Intelligence Laboratory, Massachusetts Institute of Technology). Cambridge, Mass., 1976.

Green, T. R. G. Conditional program statements and their comprehensibility to professional programmers. Journal of Occupational Pscyhology, 1977, 93-109.

Green, T. R. G. If and thens: is nesting just for the birds? Software - Practice and Experience, 1980, 10, 373-381.

Green, T. R. G., Sime M. E., & Fitter, M. J. The art of notation. In M. J. Coombs & J. L. Alty (Eds.), Computing Skills and the User Interface. London: Academic Press, 1980.

Greeno, J. G. Indefinite goals in well-structured problems. Psychological Review, 1976, 83(6), 479-491.

Greeno, J. G. Process of understanding in problem-solving. In N. J. Catellan, D. B. Pisoni, & George Potts (Eds.), Cognitive theory (Vol. 1), Hillsdale, N. J.: Laurence Erlbaum Ass., 1977.

Greeno, J. G. Understanding and procedural knowledge in mathematics instruction. Educational Psychologist, 1978, 12, 262-283.

Greeno, J. G. Natures of problem-solving abilities. In W. K. Estes (Ed.), Handbook of learning and cognitive processes (Vol. 5, Human information processing), Hillsdale, N. J.: Laurence Erlbaum Ass., 1978.

Greeno, J. G. Some examples of cognitive task analysis with instructional implications. In R. Snow, P. Federico, W. E. Montague (Eds.), Cognitive process analysis of learning and problem-solving (Vol. 2, Aptitude, learning and instruction). Hillsdale, N.J.: Laurence Erlbaum Ass., 1980.

Greeno, J. G. Trends in the theory of knowledge for problem solving. in D. T. Tuma & F. Reif (Eds.), Problem solving and education: issues in teaching and research. Hillsdale, N. J.: Laurence Erlbaum Ass., 1980.

Greeno, J. G. & Magone, Maria E. Theory of constructions and set in problem-solving. Memory and Cognition, 1979, 7(6), 445-461.

Greeno, J. G. & Riley M. S. Conceptual competence and children's counting. Cognitive Psychology, 1984, 16(1), 94-143.

Grogono, P. & Nelson, S. Problem solving and computer programming. Reading, Mass.: Addison-Wesley, 1982.

Hansen, W. J., Measurement of program complexity by the pair, cyclomatic number, operator count. ACM SIGPLAN Notices, 1978, 13(3), 29-33.

Harrison, W., Magel, k., Kluczny, R., & DeKock, A. Applying software complexity metrics to program maintenance. IEEE Computer, 1982, 15(9), 65-79.

Hayes, J. R. & Simon, H. A. Psychological differences among problem isomorphs. In J. Castellan, D. B. Pisoni, & G. R. Potts (Eds.), Cognitive learning theory (Vol. 2). Hillsdale, N. J.: Laurence Erlbaum Ass., 1977.

Hayes-Roth, B. Implications of human pattern processing for the design of artificial knowledge systems. In D. A. Waterman & F. Hayes-Roth (Eds.), Pattern-directed inference systems. New York: Academic Press, 1978.

Hinsley, D. A., Hayes, J. R., & Simon, H. A. From words to equations: meaning and representation in algebra word problems. In M. A. Just & P. A. Carpenter (Eds.), Cognitive processes in comprehension. Hillsdale, N. J.: Laurence Erlbaum Ass., 1977.

Hunt, Earl. On the nature of intelligence. Science, 1983, 219(481), 141-146.

Kilpatrick, J. Reaction paper 2 (Task ariables in mathematical problem-solving). In G. A. Goldin & E. McClintock (Eds.), Task variables in mathematical problem-solving. Columbus, Ohio: The Ohio State University, 1979. (ERIC Clearinghouse for Science, Mathematics, and Environmental Education).

Kintsch, W. Comprehension and memory of text. In W. K. Estes (Ed.), Handbook of learning and cognitive processes (Vol. 5). Hillsdale, N. J.: Laurence Erlbaum Ass., 1977.

Kintsch, W. On comprehending stories. In M. A. Just & P. A. Carpenter (Eds.), Cognitive processes in comprehension. Hillsdale, N. J.: Laurence Erlbaum Ass., 1977.

Kintsch, W. & Young, S. R. Selective recall of decision-relevant information from texts. Memory and Cognition, 1984, 12(2), 112-117.

Kulm, G. The classification of problem-solving research variables. In G. A. Goldin & C. E. McClintock (Eds.), Task variables in mathematical problem-solving. Columbus, Ohio: The Ohio State University, 1979. (ERIC Clearinghouse for Science, Mathematics, and Environmental Education).

Larkin, J. H. Teaching problem solving in physics: the psychological laboratory and the practical classroom. In D. T. Tuma & F. Reif (Eds.), Problem solving and education: issues in teaching and research. Hillsdale, N. J.: Laurence Erlbaum Ass., 1980.

Larkin, J. H., McDermott, J., Simon, D., & Simon, H. A. Expert and novice performance in solving physics problems. Science, 1980, 20(4450), 1335-1342.

Larkin, J. H. & Rainard, B. A research methodology for studying how people think. Journal of Research in Science Teaching, 1984, 21(3), 235-254.

Lemos, R. S. Fortran programming: an analysis of pedagogical alternatives. Journal of Educational Data Processing, 1975, 12(3), 12-19.

Lemos, R. S. Methods, styles, and attitudes in the programming language classroom. Computer, 1980, 13(4), 58-65.

Lochhead, J. Research synthesis on problem-solving. Educational leadership, 1981, 39(1), 68-70.

Love, T. An experimental investigation of the effect of program structure on program understanding. ACM SIGPLAN Notices, 1977, 12(3), 105-113.

Lugar, G. F. State-space representation in problem solving behavior. In G. A. Goldin & C. E. McClintock (Eds.), Task variables in mathematical problem solving. Columbus, Ohio: The Ohio State University, 1979 (ERIC Clearinghouse for Science, Mathematics, and Environmental Education).

Lukey, F. J. Understanding and debugging programs. International Journal of Man-Machine Studies, 1980, 12, 189-202.

Lukey, F. J. Comprehending and debugging programs. In M. J. Coombs & J. L. Alty (Eds.) Computing skills and the user interface. London: Academic Press, 1980.

Mandler, J. M. & Johnson, N. Remembrance of things parsed: story structure and recall. Cognitive Psychology, 1977, 9(1), 111-151.

Mandler, J. M. & DeForest, M. Is there more than one way to recall a story? Child Development, 1979, 50(3), 886-889.

Mayer, R. E. Different problem solving competencies established in learning computer programming with and without meaningful models. Journal of Educational Psychology, 1975, 67, 725-734.

Mayer, R. E. Some conditions of meaningful learning for computer programming: advanced organizers and subject control of frame order. Journal of Educational Psychology, 1976, 68(2), 143-150.

Mayer, R. E. Analysis of a simple computer programming language: transactions, prestatements and chunks. Series in Learning and Cognition, Tech. Report No. 79-2, University of California, Santa Barbara, Cal., 1979.

Mayer, R. E. A psychology of learning BASIC. Communications of the ACM, 1979, 22(11), 589-594.

Mayer, R. E. The psychology of how novices learn computer programming. ACM Computing Surveys, 1981, 13(1), 121-141.

Mayer, R. E. & Bayman, P. Psychology of calculator languages: a framework for describing differences in users' knowledge. Communications of the ACM, 1981, 24(8), 511-520.

McCabe, T. J. A complexity measure. IEEE Transactions on Software Engineering, 1976, SE-2(4), 308-320.

McEntyre, D. S. Structures and strategies for teaching computer programming in an introductory course. (Doctoral dissertation, University of California, Berkeley, 1977). Dissertation Abstracts International, 1978, 38, 6585a, (University Microfilms No. 7805629).

McClintock, C. E. Hueristic processes as task variables. In G. A. Goldin & C. E. McClintock (Eds.), Task variables in mathematical problem solving. Columbus, Ohio: The Ohio State University, 1979. (ERIC Clearinghouse for Science, Mathematics, and Environmental Education).

McKeithen, K. B. Assessing knowledge structures in novice and expert programmers. (Doctoral dissertation, University of Michigan, 1979). Dissertation Abstracts Internation, 1979, (University Microfilms No. 7916770).

Miller, L. A. Programming by non-programmers. International Journal of Man-Machine Studies, 1974, 6(2), 237-260.

Miller, L. A. Behavioral studies of the programming process (Report RC 7637). Yorktown Heights, New York: IBM Thomas Watson Research Center, 1978.

Miller, L. A. Natural language programming: style, strategies, and contrasts. IBM Systems Journal, 1980, 20(2), 184-215.

Miller, L. A. Natural language programming: style, strategies, and contrasts. Perspectives in computing, 1981, 1(4), 22-33.

Moses, B. Individual differences in problem solving. Arithmetic teacher, 1982, 30(4), 10-14.

Myers, G. J. An extension of the cyclometric measure of program complexity. ACM SIGPLAN Notices, 1977, 12(10), 61-64.

Neches, R. & Hayes, J. R. Progress toward a taxonomy of strategy transformations. In A. M. Lesgold, J. W. Pellegrino, S. Fokkema, & R. Glaser (Eds.), Cognitive psychology and instruction. New York: Plenum Press, 1978.

Newell, A. One final word. In D. T. Tuma & F. Reif (Eds.), Problem solving and education: issues in teaching and research. Hillsdale, N. J.: Lawrence Erlbaum Ass., 1980.

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

Nie, N. H., Hull, C., Jenkins, J. G., Steinbrenner, K., & Bent, D. H. Statistical Package for the Social Sciences. (2nd ed.) New York: McGraw-Hill, 1975.

Oviedo, E. I. Control flow, data flow, and program complexity. Proceedings COMPSAC-80: IEEE 4th International Computer Software and Applications Conference, 1980, 146-152.

Papert, S., Weir, S., Watt, D., Bamberger, J., Abelson, H., diSessa, A., Hein, G. & Dunning, S. Interim report of the LOGO Project in the Brookline Public Schools: an assessment and documentation of a children's computer laboratory. Cambridge, Mass.: MIT Artificial Intelligence Laboratory - LOGO Group, 1979.

Papert, S., Watt, D, diSessa, A. & Weir, S. Final report of the Brookline LOGO Project, part II: project summary and data analysis. Cambridge, Mass.: MIT Artificial Intelligence Laboratory - LOGO Group, 1979.

Papert, S. Mindstorms: children, computers, and powerful ideas. New York: Basic Books, 1980.

Plum, T. Fooling the user of a programming language. Software - Practice and Experience, 1977, 7, 215-221.

Polya, G. How to solve it: a new aspect of mathematical method (2nd Ed.). Princeton, N. J.: Princeton University Press, 1974.

Reed, S. K., Ernst, G. W., & Banerji, R. The role of analogy in transfer between similar problem states. Cognitive Psychology, 1974, 6(3), 436-450.

Reisbeck, C. & Schank, R. C. Comprehension by computer: Expectation-based analysis of sentences in context. In W. J. M. Levelt & G. B. Flores d'Arcais (Eds.) Studies in the perception of language. New York: John Wiley, 1978.

Reitman, J. S. Skilled perception in Gō: deducing memory structures from inter-response times, Cognitive Psychology, 1976, 8, 336-356.

Rich, C. Inspection methods in programming (Tech. Report AI-TR-604). Cambridge, Mass.: Massachusetts Institute of Technology, Artificial Intelligence Laboratory, June 1981.

Rich, C., Shrobe, H. E., Waters, R. C., Sussman, G. J., & Hewitt, C. E. Programming viewed as an engineering activity (AIM 459). Massachusetts Institute of Technology, Artificial Intelligence Laboratory, January 1978 (NTIS No. AD-A052 307).

Rich, C. & Schrobe, H. E. Initial report on a LISP programmers apprentice. IEEE Transactions on Software Engineering, 1978, SE-4(6), 456-467.

Rich, C. & Waters, R. The disciplined use of simplifying assumptions. ACM SIGSOFT Engineering Notes, 1982, 7(5), 150-154.

Richgels, D. J. Schema theory, linguistic theory, and representations of reading comprehension. Journal of Educational Research, 1982, 76(1), 54-62.

Riley, M. S., Greeno, J. G., & Heller, J. I. Development of children's problem solving ability in arithmetic. In H. P. Ginsburg (Ed.), The development of mathematical thinking. New York: Academic Press, 1983.

Rips, L. J. & Marcus, S. L. Suppositions and the analysis of conditional sentences. In M. A. Just & P. A. Carpenter. Cognitive processes in comprehension. Hillsdale, N. J.: Laurence Erlbaum, 1977.

Royer, J. M. & Cable, G. W. Illustrations, analogies and facilitative transfer in prose learning. Journal of Educational Psychology, 1976, 68(2), 205-209.

Sacerdotti, E. D. A structure for plans and behavior. New York: Elsevier North-Holland, 1977.

Sachs, J. S. Recognition memory for syntactical and semantic aspects of connected discourse. Perception and Psychophysics, 1967, 2(9), 437-442.

Scapin, D. L. Computer commands in restricted natural language: some aspects of memory and experience. Human Factors, 1981, 23(3), 365-375.

Shank, R. C. & Abelson, R. Scripts, plans, goals, and understanding. Hillsdale, N. J.: Laurence Erlbaum Ass., 1977.

Schneiderman, B. Exploratory experiments in programmer behavior. International Journal of Computer and Information Sciences, 1976, 5(2), 123-143.

Schneiderman, B. Measuring computer program quality and comprehension. International Journal of Man-Machine Studies, 1977, 9, 465-478.

Schneiderman, B. Software psychology. Cambridge, Mass.: Winthrop Publishers, 1980.

Schneiderman, B. & Mayer, R. Syntactic/semantic interactions in programmer behavior: a model and experimental results. International Journal of Computer and Information Sciences, 1979, 8(3), 219-238.

Schoen, H. L. & Oehmke, T. A new approach to the measure Of problem-solving skills. In Krulik, S. & Reys, R. E. (Eds.), Problem Solving in School Mathematics. Reston, Vir.: National Council of Teachers of Mathematics, 980.

Schoenfeld, A. H. Explicit heuristic training as a variable in problem solving performance. Journal for Research in Mathematics Education, 1979, 10(3), 173-187.

Schoenfeld, A. H. Measures of problem solving performance and of problem solving instruction. Journal for Research in Mathematics Education, 1982, 13(1), 31-49.

Schoenfeld, A. H. Teaching problem-solving skills. American Mathematical Monthly, 1980, 87(10), 1980.

Sheil, B. A. The psychological study of programming. ACM Computing Surveys, 1981, 13(1), 101-120.

Sheppard, S. B., Curtis, B., Milliman, P., & Love T. Modern coding practices and programmer performance. Computer, 1979, 12(12), 41-49.

Shimp, Charles P. Organization in memory and behavior. Journal of the Experimental Analysis of Behavior, 1976, 26, 113-130.

Sime, M. E., Green, T. R. G., & Guest, D. J. Psychological evaluation of two conditional constructions used in computer languages. International Journal of Man-Machine Studies, 1973, 5, 123-142.

Sime, M. E., Green, T. R. G., & Guest, D. J. Scope-marking in computer conditionals - A psychological evaluation. International Journal of Man-Machine Studies, 1977, 9, 107-118.

Sime, M. E., Arblaster, A. T., & Green, T. R. G. Structuring the programmer's task. Journal of Occupational Psychology, 1977, 50, 205-216.

Simon, H. A. The structure of ill structured problems. Artificial Intelligence, 1973, 4, 145-180.

Simon, H. A. & Hayes, J. R. The understanding process: Problem isomorphs. Cognitive Psychology, 1976, 8(2), 165-190.

Simon, H. A. Problem solving and education. In D. T. Tuma & F. Reif (Eds.), Problem solving and education: Issues in teaching and research. Hillsdale, N. J.: Laurence Erlbaum Ass., 1980.

Simon, H. A. The sciences of the artificial (2nd ed.) Cambridge, Mass.: The MIT Press, 1981.

Skinner, B. F. Notebooks. Englewood Cliffs, N. J.: Prentice-Hall, 1980.

Solomon, C. Language in the LOGO computer culture. Proceedings of the 1979 National Educational Computing Conference, Norfolk, Va., 23-25 Sept. 1979 (ERIC ED 178-068), 25-254.

Stefik, M., Aikins, J., Balzer, R., Benoit, J., Birnbaum, L., Hayes-Roth, F., & Sacerdoti, E. Basic concepts for building expert systems. In F. Hayes-Roth, D. A. Waterman, & D. Lenat (Eds.), Building expert systems. Reading, Mass.: Addison-Wesley, 1983.

Stevens, A. L. & Collis, A. Multiple conceptual models of a complex system. In R. Snow, P. A. Federico, & W. E. Montague (Eds.), Cognitive process analyses of learning and problem solving (Vol. 2: Aptitude, learning and instruction). Hillsdale, N. J.: Laurence Erlbaum Ass., 1980.

Sussman, G. J. A computational model of skill acquisition (AI TR-297). Artificial Intelligence Laboratory: Massachusetts Institute of Technology, August 1973. (NTIS No. AD-773 472).

Taylor, B. M. Text structure and children's comprehension and memory for expository material. Journal of Educational Psychology, 1982, 74(3), 323-340.

Thorndyke, P. W. Knowledge transfer in learning from texts. In A. M. Lesgold, J. W. Pellegrino, S. Fokkema, & R. Glaser (Eds.), Cognitive psychology and instruction. New York: Plenum Press, 1978.

van Dijk, T. A. Semantic macrostructures and knowledge frames in discourse comprehension. In M. A. Just & P. A. Carpenter (Eds.), Cognitive processes in comprehension. Hillsdale, N. J.: Laurence Erlbaum Ass., 1978.

Voss, J. F. Cognition and instruction: toward a cognitive theory of learning. In A. M. Lesgold, J. W. Pellegrino, S. Fokkema, & R. Glaser (Eds.), Cognitive psychology and instruction. New York: Plenum Press, 1978.

Waters, R. C. Automatic analysis of the logical structure of programs (TR-492), Artificial Intelligence Laboratory: Massachusetts Institute of Technology, Dec. 1978, (NTIS No. AD-A084 818).

Waters, R. C. A method for analyzing loop programs. IEEE Transactions on Software Engineering, 1979, SE-5(3), 237-247.

Waters, R. C. The programmer's apprentice: knowledge based program editing. IEEE Transactions on Software Engineering, 1982, SE-8(1), 1-12.

Watt, D. A comparison of the problem-solving styles of two students learning LOGO: a computer language for children. Proceedings of the 1979 National Educational Computing conference, Norfolk, Va.: 23-25 Sept. 1979 (ERIC ED 178-068), 255-260.

Watt, D. Final report of the Brookline LOGO Project, part III: profile of individual student's work. Cambridge, Mass.: Massachusetts Institute of Technology, Artificial Intelligence Laboratory - LOGO Group, 1979.

Webb, N. Content and context variables in problem tasks. In G. A. Goldin & C. E. McClintock (Eds.), Task variables in mathematical problem-solving. Columbus, Ohio: The Ohio State University, 1979. (ERIC Clearinghouse for Science, Mathematics, and Environmental Education).

Weiser, M. Programmers use slices when debugging. Communications of the ACM, 1982, 25(7), 446-452.

Weiser, M. & Shertz, J. Programming problem representation in novice and expert programmers. International Journal of Man-Machine Studies, 1983, 19(4), 391-398.

Wells, G. W. The relationship between the processes involved in problem-solving and the processes involved in computer programming. (Doctoral dissertation, University of Cincinatti, 1981). Dissertation Abstracts International, 42, 2009a. (University Microfilms No. 8123791).

Wertheimer, M. Productive thinking (Enlarged Ed.). New York: Harper-Row, 1959.

Wickelgren, W. A. How to solve problems. San Francisco: W. H. Freeman and Co., 1974.

Winogard, T. A framework for understanding discourse. In M. A. Just & P. A. Carpenter (Eds.), Cognitive processes in comprehension. Hillsdale, N. J.: Laurence Erlbaum Ass., 1977.

APPENDICES

APPENDIX A

GLOSSARY OF TERMS

ALGORITHM. An unique sequence of steps, or moves, for solving a class of equivalent problems.

APPROXIMATION TRAINING. An instructional technique in the inspection methods treatments by which students were taught how to modify chunks that are approximate solutions of programming problems into more satisfactory solutions.

ARRANGEMENT PROBLEM. A task in which elements must be arranged in some combination that satisfies criteria set out in the problem statement.

CHUNK. In this study, the term is used in one of three ways: (1) In a psychological sense, a chunk is a theoretical formalism used to describe a pattern of stimuli or responses that have become consolidated into a single behavioral unit (see FUNCTIONAL UNIT OF BEHAVIOR); (2) In a computational sense, a chunk is a collection of propositions in a computer program that interact by controlling or by passing data to one another, thereby performing a clearly defined function; (3) In an instructional sense, chunks are sets of propositions in a programming language that are frequently used together by computer programmers to solve programming problems and, therefore, can be made the basis for teaching that programming language.

CHUNKING SCHEMA. The representation of computer programs as nested hierarchies of chunks, chunks bound together by control and data flow.

COMPLEXITY. The number of paths, or "transfers of control," within a program. Complexity is usually associated with the number of loops or branches within a program. Increasing the number of program paths is said to increase the difficulty of understanding, testing, and modifying programs.

CONCEPTUALLY-COMPLEX DOMAIN. A body of knowledge, or subject area, that is characterized by a relatively large number of functional units of information and by a relatively large number of connections, or relations, among those units.

CONSTRUCTIVE SEARCH. A problem-solving process in which an ill-structured problem space is "enriched" by adding elements or partial solutions and then testing these elements or solutions against the requirements of goals or subgoals.

CONTEXTUAL REFERENCING. Attributing meaning to a proposition based on the shared assumptions and knowledge possessed by reader and author or by speaker and listener.

DOMAIN-INDEPENDENT KNOWLEDGE. Perceptual, procedural, or propositional information that is found in two or more, often related, subject areas, or domains.

DOMAIN-SPECIFIC KNOWLEDGE. Perceptual, procedural, or propositional information that is found within a given subject area, or domain, but is found only rarely in other subject areas.

ENDLESS LOOP. See LOOP, ENDLESS.

EXTERNAL REFERENCING. Attributing meaning to a proposition by referring, either explicitly or implicitly, to external events or situations.

FUNCTIONAL UNITS OF BEHAVIOR. Organized patterns of stimuli and response having a significant degree of internal structure. This internal structure most often is inferred from experimental observations of the temporal or the ordinal relationships among the constituent behaviors.

HEURISTIC. A less precise, rule-of-thumb procedure that may suggest a strategy or algorithm for solving a problem. Also known as a GENERAL STRATEGY or a DOMAIN-INDEPENDENT STRATEGY.

Ill-STRUCTURED PROBLEM. Any problem not meeting the three criteria for well-structured problems. See WELL-STRUCTURED PROBLEMS.

INTERNAL COHERENCE. The organization of propositions within a text, an organization based on repetition, conditioning, causation, or temporal ordering.

LEARNING THROUGH INTERACTIVE PROGRAMMING. A term applied to instruction or a curriculum that uses computer programming as a means of teaching logic, mathematics, or problem-solving skills.

LOGO. A computer programming language developed by Seymour Papert and the LOGO Group of the Artificial Intelligence Laboratory, Massachusetts Institute of Technology.

LOOP, ENDLESS. See ENDLESS LOOP.

MODEL. A procedure or device to represent in a concrete and familiar way that which is abstract, unfamiliar, or hidden.

PERCEPTUAL TRAINING. A teaching technique in the inspection methods treatments in which a tachistoscope was used to train students to identify and name chunks whether alone or embedded within other chunks. Also referred to as RECOGNITION TRAINING.

PROBLEM. A proposition or a situation requiring the performance of an operation or the making of a construction.

PROBLEM SOLVING. The process of searching the set of all possible solutions to a problem, a search constrained by a perceptual, procedural, or propositional understanding of the domain in which the problem is set.

PROBLEM OF INDUCING STRUCTURE. A problem task whose solution requires inferring specific patterns of relations among the elements within a prblem space.

PROCEDURAL KNOWLEDGE. Knowledge of how to do things.

PROCEDURAL SPECIFICATION. A class of problem-solving tasks involving the specification of a "sequence of actions such that, when executed by a designated agent, a particular goal can be accomplished" (Miller, 1981, p. 22).

PROBLEM SPECIFICATION. A description of what a completed program should do.

PRODUCTION SYSTEM. (1) In artificial intelligence programs, production systems are structures that make the execution of specific actions contingent upon the presence of certain conditions. (2) In cognitive psychology, production systems are mental representations, consisting of actions paired with conditions ("condition-action pairs"), specifying when or how the action is to be taken.

PROPOSITION. A unit of verbal behavior consisting of a relational term, or operator, and one or more arguments, or details.

PROPOSITIONAL KNOWLEDGE. Verbal behavior, consisting of declarations about concepts, word meanings, events or episodes. In computer programming, statements about the purpose of commands or groups of commands is propositional knowledge. Also known as DECLARATIVE KNOWLEDGE.

PROPOSITIONAL MACROSTRUCTURE. The overall organization of propositions in texts or spoken language. See SCHEMA.

REPRESENTATION. Internal, mental structures that result from the encoding or mental reorganization of the external environment. The notion of representation is the basis of certain theoretic formalisms such as "production systems," "schemas," "scripts," and "procedural networks."

SCHEMA. In the literature of cognitive psychology, the term has been used in two ways: (1) Mental representations which give a reader or listener a reliable prediction of the content or structure of certain classes of written or spoken language; (2) The underlying organization (internal referencing) of the form or content of written or spoken language. Also known as SCHEMATA.

SCHEMATIC TRAINING. A technique in the inspection methods treatments by which students were taught to match chunks to written descriptions of their behavior and to match problem statements to chunks that are approximate solutions to those problems.

SEGMENTATION TRAINING. A technique in the inspection methods treatments by which the students were taught how to decompose a program into its component chunks and subchunks.

STRATEGY. A well-defined procedure that narrows down the set of possible, alternate moves without specifying a single, successful path to a solution.

SYNTAX. The arrangement of words, or elements, in a sentence to show their relationships.

TEMPORAL-SEQUENTIAL SCHEMA. The representation of computer programs as lists of commands, organized according to the temporal order by which computers execute a sequence of individual commands.

TEXTUAL COHERENCE. The degree and kind of organization that binds together propositions within a text.

TRANSFER OF LEARNING. The effect of learning one task upon the later learning or performance of another task.

UNDERSTANDING. In the literature of cognitive psychology, understanding involves the encoding, or mental reorganization, of the external environment, which results in a coherent, internal network of mental representations

corresponding to the salient features of the external environment. In the behaviorist account, an organism is said to understand if it behaves in an appropriate fashion.

WELL-STRUCTURED PROBLEM. According to Simon (1973) and Greeno (1976), a well-structured problem must meet three criteria: (1) The problem occurs in an environment containing a specified set of elements and a set of rules for combining those elements; (2) The problem statement must define a set of operators for making transitions from one problem state to another until the goal state is achieved; (3) The problem statement must specify a "definite goal" state, thereby excluding certain problems with multiple solutions from the "well-structured" category.

APPENDIX B

ADDITIONAL TABLES

Table 40. t Tests of Differences between Solution Means within the Problem Relatedness Categories of the Simple Modeling Group

	Identical (13.37)	Equivalent (13.37)	Isomorphic (12.26)	Similar (10.74)
Target (11.19)	3.50[a] ($p<.002$)[b]	3.35 ($p<.002$)	1.47 ($p<.15$)	.62 ($p<.53$)
Identical (13.37)		.80 ($p<.43$)	2.52 ($p<.02$)	5.02 ($p<.001$)
Equivalent (13.37)			2.28 ($p<.03$)	4.62 ($p<.001$)
Isomorphic (12.26)				2.50 ($p<.02$)

[a] t value. (The means were calculated on subjects, who attended nine or more programming classes.)

[b] 2-Tail probability.

Table 41. t Tests of Differences between Solution Means within the Problem Relatedness Categories of the Simple Inspection Methods Group

	Identical (13.15)	Equivalent (13.10)	Isomorphic (11.20)	Similar (12.45)
Target (12.75)	.45[a] ($p<.65$)[b]	.49 ($p<.63$)	1.64 ($p<.11$)	.34 ($p<.73$)
Identical (13.15)		.22 ($p<.82$)	4.32 ($p<.002$)	1.16 ($p<.26$)
Equivalent (13.10)			2.57 ($p<.02$)	.62 ($p<.54$)
Isomorphic (11.20)				2.34 ($p<.03$)

[a] t value. (The means were calculated on those subjects, who attended at least nine programming classes.)

[b] 2-Tailed probability.

Table 42. t Tests of Differences between Solution Means within the Problem Relatedness Categories of the Complex Modeling Group

	Identical (13.17)	Equivalent (13.13)	Isomorphic (12.54)	Similar (11.13)
Target (11.33)	2.58[a] ($p<.02$)[b]	3.21 ($p<.004$)	1.77 ($p<.09$)	.38 ($p<.71$)
Identical (13.17)		.23 ($p<.82$)	.77 ($p<.45$)	2.72 ($p<.02$)
Equivalent (13.13)			.54 ($p<.60$)	3.08 ($p<.005$)
Isomorphic (12.54)				2.64 ($p<.01$)

[a]t value. (The means were calculated on subjects, who attended nine or more programming classes.)

[b]2-Tail probability.

Table 43. t Tests of Differences between Solution Means within the Problem Relatedness Categories of the Complex Inspection Methods Group

	Identical (11.19)	Equivalent (10.26)	Isomorphic (10.63)	Similar (10.48)
Target (10.15)	1.43[a] ($p<.16$)[b]	.15 ($p<.88$)	.67 ($p<.50$)	.54 ($p<.60$)
Identical (11.19)		1.25 ($p<.22$)	1.13 ($p<.27$)	1.15 ($p<.26$)
Equivalent (10.26)			.38 ($p<.70$)	.27 ($p<.79$)
Isomorphic (10.63)				.11 ($p<.91$)

[a]t value. (The means were calculated on subjects, who attended nine or more programming classes.)

[b]2-Tail probability.

Table 44. MANOVA: Syntax Errors by Instructional Method and Gender

(Main Effects)

Target Items Included			Target Items Excluded		
Modeling	Inspection M.		Modeling	Inspection M.	
19.33 (N=51)	21.54 (N=47)	(p<.22)	18.08 (N=51)	20.79 (N=47)	(p<.12)
Males	Females		Males	Females	
17.92 (N=42)	22.88 (N=56)	(p<.005)	17.00 (N=42)	22.07 (N=56)	(p<.004)

(No Interactions)

Note. Means calculated on the performances of subjects attending nine or more programming classes.

Table 45. MANOVA: Control Flow Errors by Instructional Method and Gender

(Main Effects)

Target Items Included			Target Items Excluded		
Modeling	Inspection M.		Modeling	Inspection M.	
19.73 (N=51)	20.83 (N=47)	(p<.44)	19.57 (N=51)	20.62 (N=47)	(p<.46)
Males	Females		Males	Females	
18.35 (N=42)	22.41 (N=56)	(p<.005)	18.12 (N=42)	22.28 (N=56)	(p<.004)

(No Interactions)

Note. Means calculated on the performances of subjects attending nine or more programming classes.

Table 46. MANOVA: Data Errors by Instructional Method and Gender

(Main Effects)

Target Items Included		Target Items Excluded	
Modeling	Inspection M.	Modeling	Inspection M.
20.29 (N=51)	20.79 (N=47)	19.88 (N=51)	20.49 (N=47)
(p<.72)		(p<.65)	
Males	Females	Males	Females
9.15 (N=42)	22.09 (N=56)	18.79 (N=42)	1.74 (N=56)
(p<.04)		(p<.04)	

(No Interactions)

Note. Means calculated on the performances of subjects attending nine or more programming classes.

Table 47. MANOVA: Solutions to Program Interpretation Problems by Instructional Method and Gender

(Main Effects)

Target Items Included		Target Items Excluded	
Modeling	Inspection M.	Modeling	Inspection M.
30.41 (N=51)	29.09 (N=47)	24.88 (N=51)	23.69 (N=47)
(p<.50)		(p<.46)	
Males	Females	Males	Females
32.21 (N=42)	27.02 (N=56)	26.60 (N=42)	21.88 (N=56)
(p<.001)		(p<.005)	

(No Interactions)

Note. Means calculated on the performances of subjects attending nine or more programming classes.

Table 48. MANOVA: Solutions of Related Problems by Instructional Method and Gender

	(Main Effects)				
	Target	Identical	Equivalent	Isomorphic	Similar
Method:					
Modeling	11.25[a]	13.19	13.29	12.37	10.92
Insp. M.	11.26	12.04 b	11.53	10.92	11.41
	($p<.99$)	($p<.20$)	($p<.06$)	($p<.11$)	($p<.56$)
Gender:					
Male	12.12	13.94	13.83	13.04	11.94
Female	10.27	11.24	10.96	10.20	10.33
	($p<.02$)	($p<.003$)	($p<.003$)	($p<.002$)	($p<.05$)
	(No Interactions)				

[a]Means. (The means were calculated on the performances of subjects, who attended nine or more programming classes.)

[b]Significance of the F ratio.

Table 49. MANOVA: Solutions within the Problem Complexity Categories by Instructional Method and Gender

	(Main Effects)			
	S. State.	L. Blocks	F. Branching	Looping
Method:				
Modeling	15.79[a]	13.23	10.65	10.10
Insp. M.	14.90	12.31	9.37	9.33
	($p<.36$)[b]	($p<.36$)	($p<.16$)	($p<.34$)
Gender:				
Male	16.52	13.62	11.48	11.13
Female	14.12	11.90	8.49	8.22
	($p<.02$)	($p<.09$)	($p<.002$)	($p<.001$)
	(No Interactions)			

[a]Means. (The means were calculated on the performances of subjects, who attended nine or more programming classes. Target problems were excluded from these calculations.)

[b]Significance of the F ratio.

Table 50. MANOVA: Solutions of Problems: Complexity Categories by Materials Complexity and Gender

	S. State.	L. Blocks	F. Branch.	Looping
(Main Effects)				
Instructional Materials:				
Low Complex.	15.85[a]	13.58	10.73	9.98
High Complex.	14.91	12.06	9.40	9.49
	($p<.32$)[b]	($p<.13$)	($p<.15$)	($p<.54$)
Gender:				
Male	16.52	13.62	11.48	11.13
Female	14.12	11.90	8.49	8.22
	($p<.02$)	($p<.08$)	($p<.001$)	($p<.001$)
(No Interactions)				

[a]Means. (The means were calculated on the performances of subjects, who attended nine or more programming classes. Target problems were excluded from these calculations.)

[b] Significance of the F ratio.

Table 51. MANOVA: Solutions of Related Problems by Instructional Materials and Gender

	Target	Identical	Equivalent	Isomorphic	Similar
(Main Effects)					
Materials:					
Lo. Comp.	11.85[a]	13.28	13.49	11.81	11.47
Hi. Comp.	10.71	11.61	11.61	11.53	10.78
	($p<.17$)[b]	($p<.20$)	($p<.05$)	($p<.77$)	($p<.42$)
Gender:					
Male	12.21	13.94	13.83	13.04	11.94
Female	10.17	11.24	11.02	10.11	10.17
	($p<.01$)	($p<.003$)	($p<.003$)	($p<.002$)	($p<.04$)
(No Interactions)					

[a]Means. (The means were calculated on the performances of subjects, who attended nine or more programming classes.)

[b]Significance of the F ratio.

Table 52. MANOVA: Total Syntax Errors by Instructional Method and Grade

(Main Effects)

Target Items Included		Target Items Excluded	
Modeling	Inspection M.	Modeling	Inspection M.
19.19 (N=51)	18.37 (N=47)	18.24 (N=51)	17.97 (N=47)
(p<.14)		(p<.13)	
5th Grade	6th Grade	5th Grade	6th Grade
22.15 (N=42)	18.58 (N=56)	21.23 (N=42)	18.34 (N=56)
(p<.05)		(p<.08)	

(No Interactions)

Note. Means calculated on the performances of subjects attending nine or more programming classes.

Table 53. MANOVA: Syntax Errors within the Relatedness Categories by Instructional Methods and Grade

(Main Effects)

	Target	Identical	Equivalent	Isomorphic	Similar
Methods:					
Modeling	4.14[a]	3.29	3.43	3.69	4.16
Insp. M.	4.28	4.06	4.30	4.47	4.30
	($p<.74$)[b]	($p<.07$)	($p<.07$)	($p<.20$)	($p<.80$)
Grade					
Fifth	4.74	4.33	4.23	4.41	4.44
Sixth	3.85	3.22	3.59	3.83	4.08
	($p<.04$)	($p<.01$)	($p<.07$)	($p<.20$)	($p<.52$)

Interactions

#1. Isomorphic Problems

	Modeling	Insp. M.
5th	3.69	5.44
6th	3.80	3.86

($p<.05$)[b]

#2. Similar Problems

	Modeling	Insp. M.
5th	3.71	5.28
6th	4.47	3.69

($p<.04$)

[a]Means. (The means were calculated on the performances of subjects, who attended nine or more programming classes.)

[b]Significance of the F ratio.

Table 54. MANOVA: Syntax Errors within the Relatedness Categories by Instructional Materials and Grade

(Main Effects)

	Target	Identical	Equivalent	Isomorphic	Similar
Materials:					
Lo. Comp.	3.79[a]	3.30	3.36	3.72	3.94
Hi. Comp.	4.59	4.00	4.29	4.37	4.49
	($p<.05$)[b]	($p<.10$)	($p<.05$)	($p<.17$)	($p<.32$)
Grade					
Fifth	4.72	4.33	4.23	4.41	4.44
Sixth	3.85	3.22	3.59	3.83	4.08
	($p<.03$)	($p<.01$)	($p<.17$)	($p<.22$)	($p<.53$)

Interactions

#1. Equivalent Problems

	Lo. Comp.	Hi. Comp.
5th	4.47	4.00
6th	2.61	4.48

($p<.02$)

[a]Means. (The means were calculated on the performances of subjects, who attended nine or more programming classes.)

[b]Significance of the F ratio.

MONTANA STATE UNIVERSITY LIBRARIES
3 1762 10005046 5

D378
B833
cop.2