Order Number 9004923

Problem representation and achievement in computer programming: The differential effects of inductive reasoning skills and computer programming experience

Langstaff, Janis Jacobsen, Ph.D.

The University of Iowa, 1989

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

PROBLEM REPRESENTATION AND ACHIEVEMENT IN COMPUTER PROGRAMMING: THE DIFFERENTIAL EFFECTS OF INDUCTIVE REASONING SKILLS AND COMPUTER PROGRAMMING EXPERIENCE

by

Janis Jacobsen Langstaff

A thesis submitted in partial fulfillment
of the requirements for the Doctor of
Philosophy degree in Education
(Instructional Design and Technology)
in the Graduate College of
The University of Iowa

May 1989

Thesis supervisor: Associate Professor David F. Lohman

Graduate College
The University of Iowa
Iowa City, Iowa

CERTIFICATE OF APPROVAL

PH. D. THESIS

This is to certify that the Ph.D. thesis of

Janis Jacobsen Langstaff

has been approved by the Examining Committee
for the thesis requirement for the Doctor of Philosophy
degree in Education (Instructional Design and
Technology) at the May 1989 graduation.

Thesis committee: _____
Thesis supervisor

_____
Member

_____
Member

_____
Member

_____
Member

To my advisor, Dr. David F. Lohman,
and to his advisor, Dr. Richard E. Snow,
in appreciation for their
inspiration and comprehensive thinking.

The concept of transfer occupies a crucial position in any theory attempting to relate learning to human ability.

On Transfer and the Abilities of Man
George A. Ferguson

ACKNOWLEDGMENTS

I am particularly grateful to the chairman of my doctoral committee, Dr. David F. Lohman, for his inspiration, guidance and encouragement throughout my dissertation. It's a great privilege to work with a person who deeply understands cognitive processes and who is committed to improving education.

Special thanks to the members of my doctoral committee, Dr. Stephen B. Dunbar, Dr. Donald L. Epley, Dr. Harold L. Schoen, and Dr. Lowell A. Schoer, Acting Dean of the College of Education, for their helpful suggestions and encouragement. The contributions of Dr. Glendon W. Blume, a former member of my committee, are deeply appreciated.

Sincere appreciation to the former Chairperson of the Computer Science Department, Dr. Theodore J. Sjoerdsma, for help designing the survey instrument and for permission to do pilot research in his department. Thanks also to the current Chairperson, Dr. Arthur C. Fleck, for computer time and for his cooperation. Special thanks to Dr. R. Christiansen and his teaching assistants for their support. This research would not have been possible without their efforts and the cooperation of the students who served as subjects.

I wish to thank Dr. Nancy Pennington, at University of Chicago, and Dr. Reif, at University of California at Berkeley, for their assistance during the pilot phase of this dissertation. I am grateful for guidance from Dr. Michelene T. Chi from the Learning Research Development Center, Pittsburgh.

For help designing the tasks, I wish to thank Dr. Don Epley, Hayden Huntley, Mark Kelley, and Tony Wilson. Sincere appreciation to Bob Blankenship, Mark Kelley, and Charles Maxon for serving as raters for this research. For data collection assistance, I wish to thank Crystal Kohl. For word processing assistance, I am grateful to Janet Emard, Rose Higgins, Jennifer and Joel Roney, Mary Ann Stone, Martha Morrow and Eva Norlyck. For help with coding, I wish to thank Carol Inman. Special thanks to Dara Llewellynn and Linda Bender for data entry assistance. The support received from the user consultants at the Weeg Computer Center was extremely helpful. Also, the large amount of computer time provided by the University of Iowa made the project possible.

I wish to express my gratitude to my friends and relatives who provided emotional support during this time. Special thanks to my parents, Mr. and Mrs. Bernold M. Jacobsen, Sr..

Finally, a very special thank you is extended to my husband, Don, without whose love, support, and encouragement this endeavor would not have been possible.

ABSTRACT

Research and business firms need more programmers who have not only a strong knowledge base in computer science but who also can solve novel problems. The roles of both novel problem skills (inductive reasoning skills) and domain specific knowledge were examined in a series of four studies. More specifically, this research investigated the effects of inductive reasoning skills and computer programming experience on problem representation and achievement in computer programming.

Study 1

The goal of the first study was to determine whether individual differences in inductive reasoning skills and prior computer programming experience make independent contributions to the prediction of first examination scores and course grades in an introductory computer programming course. For predicting Exam 1 success, a model was created containing two components: inductive reasoning and computer-related knowledge base. Computer-related knowledge base included prior programming experience and ACT Math scores. A regression analysis revealed that each variable in the model made a significant and approximately equal contribution to the model. The model yielded an r-square of .31 (n = 52). Predicting Exam 1 success was of particular interest because many students drop the course as soon as they receive their Exam 1 scores. The same model was used for predicting course grades. A regression analysis revealed that the model was not a good predictor of course grades.

Study 2

The purpose of Study 2 was to ascertain if novice and intermediate level programmers of average and high inductive reasoning skills have different organizational categories for their new knowledge. A sorting task consisting of 30 computer programming problems was administered to students enrolled in the same introductory programming course. The results for 17 novices and 16 intermediate level programmers of average and high inductive reasoning skills (Gf) were analyzed using multidimensional scaling and cluster analysis procedures. Results from the quantitative and qualitative analyses suggested that categorization behavior is influenced by the subject's level of inductive reasoning as well as by prior computer programming experience. As programming experience and inductive reasoning (Gf) increase, categories are based less on superficial features in the problem statements and more on underlying solution features or strategies.

Study 3

The purpose of Study 3 was to uncover the knowledge contained in the schemata of novice and intermediate level programmers of average and high inductive reasoning skills. Six novices of average and high inductive reasoning skills (Gf) and six intermediate level programmers of average and high inductive reasoning skills (Gf) completed a free association task based on category labels generated by subjects in Study 2. Results indicated that both natural and computer associations are contained in the schemata of novice and intermediate level programmers. Novice and experienced programmers of high inductive reasoning skills (Gf) generated a greater number of

computer concepts and more complex computer definitions and examples than the novice and intermediate level subjects of average inductive reasoning skills (Gf).

Study 4

The purpose of Study 4 was to examine in greater detail the solution plans and programming achievements of novice and intermediate level programmers of average and high inductive reasoning skills (Gf). Eleven of the twelve subjects who participated in Study 3 participated in this study. The subjects were asked to think out loud while generating their basic approaches to eleven computer programming problems. For the last four problems, subjects wrote programs. Ratings of the quality of program plans did not vary markedly across the four groups of subjects. For easier problems, Gf was the best predictor of the rated quality of plans; for more difficult problems, both Gf and programming experience were important, perhaps both being required for the generation of a good plan. For the three problems requiring programming, a correct basic approach was a good predictor of programming success. The novices of high inductive reasoning skills (Gf) as a group showed the greatest improvement in scores from the basic approach to actual programming. On the other hand, the novice group with average inductive reasoning skills (Gf) did worse translating basic approaches into solution strategies. Differences between high and average Gf experienced programmers were in the same direction, but much smaller.

Results from Studies 1-4 were discussed in terms of a process theory of aptitude for learning from instruction (Snow & Lohman, 1984). Prior experience in computer programming is important, although clearly not as important as inductive reasoning skills during the early stages of learning a new computer language. Inductive

reasoning skills develop with education and experience (Snow & Lohman, 1988). It was concluded that the computer science curriculum can encourage the development of inductive reasoning skills by teaching novel problem skills in the context of teaching computer programming.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Need For Computer Programmers

Improvements in computer hardware have increased both the computational power and the availability of all sizes of computers. Unlike software costs, the cost per unit of new hardware has decreased substantially during the past thirty years. In the mid-1950's, 90% of application costs were devoted to hardware; by the 1980's, however, 90% of application costs were for software (Shneiderman, 1980). Computer software continues to be expensive to produce and to maintain. Conventional applications of the computer continue to grow and the variety of new uses of the computer is increasing. The software industry has become a multibillion dollar international business that employs millions of people. Therefore, substantial resources will continue to be devoted to training people in computing skills (Cetron & O'Toole, 1982).

Professional programmers in both business and research firms today must continuously learn new operating systems, new programming languages, new software tools, and new hardware systems. In the context of this dynamic environment, new knowledge is needed to solve application problems in innovative ways. Typically, only a small percent of the programmers in a given organization actually do the truly innovative programming, while the majority of the programmers do lower level design and coding. Some claim that this is because many programmers have difficulty solving novel problems. Whatever the reason, a dire need exists for

more programmers who have not only a strong knowledge base in computer science but who can also solve novel problems.

Purpose of the Study

Filling the need for highly competent programmers who can solve novel problems is one of the training challenges of this decade. Unfortunately, little is known about the nature of computer programming skills, and even less about the kinds of cognitive skills people already have that might serve as a foundation for learning to program computers. Consequently, educators lack the knowledge necessary to provide optimal learning conditions for teaching computer programming. In the absence of this vital knowledge, training continues to be guided "by the tacit 'folk theories' of programming development that until now have served as the underpinnings of program instruction" (Pea & Kurland, 1983, p.1). The success of current and future training programs depends on not only an adequate understanding of the psychology of computer programming, but also on a comprehensive understanding of the knowledge and skills necessary for learning computer programming.

The purpose of this study is to investigate the differential effects of individual differences in computer programming experience and inductive reasoning abilities on achievement in an introductory programming course. Since other investigators have examined the role of experience in learning computer programming, the particular focus of this study will be on the contributions of inductive reasoning, that is, those inferential processes that expand knowledge in the face of uncertainty (Holland, Holyoak, Nisbett & Thagard, 1986). Inductive reasoning is ubiquitous in human thinking. It involves the ability to make predictions, generalizations, and projections from known instances to unknown, potential instances. In essence, inductive

reasoning involves the transfer of the familiar to the unfamiliar or novel situation. The question addressed in this study, then, is whether novice and experienced programmers who differ in inductive reasoning ability perform differently on computer programming tasks when learning a new programming language. The answer to this question may clarify problems in the selection and training of computer programmers.

Background Information and Definitions of Terms

Programming Defined

For the purposes of this study, computer programming is defined as "that set of activities involved in developing a reusable product consisting of a series of written instructions that make a computer accomplish some task" (Pea & Kurland, 1983, p.5).

Cognitive Subtasks Involved in Programming

From a psychological point of view, computer programming requires both general problem-solving skills and domain-specific knowledge. Problem-solving is typically decomposed into five stages: understanding the problem, devising a plan, implementing the plan, evaluating a potential solution, and improving the plan (Mayer, 1983: Polya, 1973). In computer programming, these cognitive subtasks roughly translate as: (a) understanding the programming problem, (b) designing or planning a programming solution, (c) writing a programming code that implements a plan, (d) comprehending and debugging the program. Some theorists assert that all the above subtasks are required for programming, whether the programmer is a novice or an expert (Pea & Kurland, 1983). Stages 1 and 2, understanding the programming problem and designing a program, are of critical importance in this study.

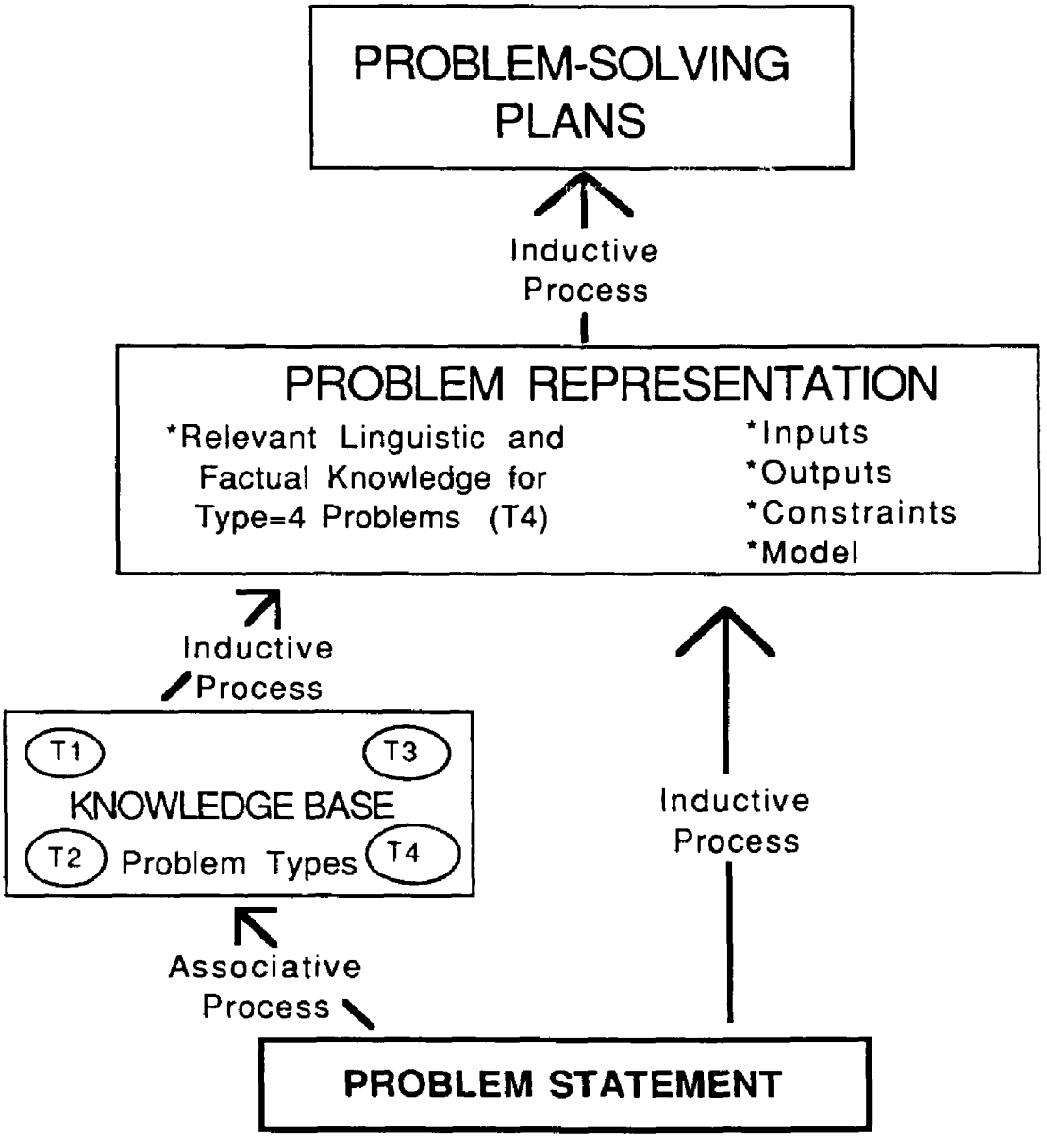
Problem Representation

The notion of problem representation is important in this investigation of computer programming and induction. Cognitive theorists assert that problem solvers construct representations of the problems they are attempting to solve. The problem representation is the problem solver's understanding of the problem. It includes the solver's linguistic, factual, strategic, and other knowledge about the problem. It also may contain a mental model of the to-be-solved problem. The elements of the model stand as tokens for aspects of the problem. The problem solver manipulates these tokens using some system of rules. The solution to this simplified problem is then mapped back on to the original problem. For example, the problem solver may solve a time-rate-distance problem by imagining two cars approaching each other on a highway. Figure 1 illustrates the process of representing a problem. The problem statement cues associations in the problem solver's knowledge base. For experienced problem solvers, this knowledge base is organized by problem types. Each problem type includes some information on solution strategy. So perhaps the problem solver decides that the problem in question is an instance of a time-rate-distance (or "T4") problem. This conjecture becomes an important aspect of the problem representation. The problem solver also reads the problem for information about inputs, outputs, and constraints and adds this information to the problem representation.

The Role of Inductive Reasoning in Representing A Problem

The quality of the problem representation the problem solver constructs greatly influences how easily a problem can be solved (Hayes & Simon, 1977; Newell & Simon, 1972). Holland, Holyoak, Nisbett and Thagard (1986) assert that induction plays a vital role in generating a problem representation. Inductive processes are

Figure 1. The Process of Representing A Problem



Note. It is assumed that other cognitive processes are involved in addition to inductive and associative processes.

involved in gathering inputs from the problem statement and knowledge base to create the problem representation. After the problem representation is created, inductive reasoning skills are vital in helping the problem solver design a plan for solving the problem. Figure 1 illustrates this process. Induction consists of generating and revising the units of the problem representation from which mental models are constructed. "Mental models are built by integrating knowledge in novel ways in order to achieve the system's goal" (p.14). If this is true, then different levels of inductive reasoning abilities should differentially effect the problem representations and thus the mental models students construct and hence their solutions to problems. This study attempts to explore the role of inductive reasoning on problem representation and achievement in computer programming.

Skill differences in problem representation and solution strategy have been investigated by many researchers (for details see Chapter II). However, previous researchers of programming-skill learning have not explored the effects of individual differences in inductive reasoning skill on problem representation and solution plans.

Research Problem

The purpose of this dissertation is to investigate the differential effects of computer programming experience and inductive reasoning abilities on problem representations, solution plans, and learning outcomes of students enrolled in an elementary computer programming course. Four studies are reported. The goal of the first study is to determine whether individual differences in inductive reasoning abilities and prior computer experience make independent contributions to the prediction of examination scores and course grades in an introductory Pascal course.

Problem representation is explored in Studies 2 and 3, whereas solution plans are investigated in Study 4. More specifically, Studies 2, 3, and 4 examine in greater detail the particular contributions of experience and inductive reasoning abilities to students' categorization of programming problems (Study 2), to the contents of their problem schemas (Study 3), and to the problem-solving plans they construct (Study 4).

CHAPTER II

REVIEW OF THE LITERATURE

Research on problem representations and solution plans is first presented, followed by research on the effects of prior computer experience on learning a new language. Next, the metatheoretical framework for this study is introduced including research on computer programming aptitude. The chapter concludes with an overview of the four studies in this dissertation.

Research on Problem Representation in Related Areas

Before reviewing the relevant literature, a distinction needs to be made concerning differences between problem-solving research in knowledge-rich domains such as physics or radiology and problem-solving research in knowledge-lean domains such as computer science. As mentioned previously, the problem representation is at least in part constructed from domain-related knowledge. In addition, the problem representation should include information about the inputs, outputs, and constraints in the problem statement, and a mental model of the problem. Research on problem representation in computer science is based more on this latter type of knowledge derived from the problem rather than on domain-related knowledge such as statistics or accounting. (However, for experienced programmers the problem representation also is based on computer programming knowledge.) This is an important distinction, especially when comparing studies of problem solving in physics and other fields with studies of problem solving in computer science. This issue does not pose an

ecological validity problem for this study because the target population is introductory computer language classes where domain knowledge (e.g., banking, etc.) is kept to a minimum.

Problem representation has been studied in the context of a variety of tasks--from puzzle-like problems (Wertheimer, 1945; Hayes & Simon, 1977) to physics problems and electronic trouble-shooting (Larkin, 1977, Egan & Schwartz, 1979, Chi, Feltovich, & Glaser, 1981). Researchers agree that the problem solver attempts to understand problems by constructing an initial problem representation. The efficiency and accuracy of the solver's further thinking is determined by the quality, completeness, and coherence of this initial representation. Furthermore, these three characteristics of the problem representation are greatly influenced by the knowledge available to the problem solver and how that knowledge is organized. The major findings on problem representation relevant to the proposed study can be summarized by examining the literature on skill differences in chess, Go, physics, electronics, and computer science. In chess, de Groot (1966) demonstrated that after a brief exposure to a chessboard display, master-level players were able to recall about twice as many chess plays as were beginners. DeGroot attributed the master players' performance to an ability to classify groups of pieces as instances of familiar playing categories. Chase and Simon (1973) replicated de Groot's findings and went on to characterize the chunks used by chess masters. They found that chunks frequently consisted of chess pieces that formed specific strategies, such as attack or defense configurations. Chase and Simon's work suggests that masters have identified the functional relationships that occur between the pieces during the game. Masters use their knowledge of

functional relationships to create internal representations of typical chess configurations (Adelson, 1981).

Like Chase and Simon (1973), Reitman (1976) found that master Go players encode game-board pieces as functional clusters. That is, pieces forming attack or defense configurations (Adelson, 1981) are encoded together.

An example from the field of electronics further extends these findings. Egan and Schwartz (1979) found that skilled electronic technicians recall the elements of a circuit diagram in functional chunks. Also, the technicians can rapidly identify a concept that serves to relate elements in a chunk. These experts systematically search circuit diagrams for elements that are conceptually related. Egan and Schwartz suggest that, through experience, technicians develop functionally-based schemata. Furthermore, expert technicians have developed knowledge about how to use these schemata.

Reif (1979) has proposed a problem-solving model in which the first step involves representing or redescribing the problem in terms of concepts provided by the knowledge base. This knowledge base is organized according to problem schemata, each of which, Reif hypothesizes, contains information necessary to solve a specific category of problem. In the process of identifying a problem as an instance of a particular type of problem, associations cue information in the knowledge base.

An exemplary study by Chi, Feltovich, and Glaser (1981) explored these and related hypotheses. Chi and her colleagues investigated both problem and plan representations in problem solving in the field of physics. They explored how experts and novices form an initial understanding of a problem, and how they use this information in generating a strategy for solving a problem. In a series of studies, these researchers attempted to determine: (a) the categories that experts and novices impose

on physics problems; (b) the knowledge which these categorical representations activate in the problem solver; and (c) the cues or features of problems which subjects use to choose among alternative categories.

Eight advanced Ph.D. students from the physics department (experts) and eight undergraduates (novices) participated. The novices had just completed a semester of mechanics. Twenty-four problems were selected from a textbook on the fundamentals of physics. The subjects were asked to sort the problems into groups based on similarities of solution. Subjects did not actually solve the problems in order to sort them. Results indicated that experts and novices categorized problems differently. Novices sorted physics textbook problems on the basis of the apparatus involved (lever, inclined plane, etc.), the words used in the problem statement, or the visual features of the diagram presented with the problem. In contrast, experts sorted the same problems on the basis of a solution strategy. More specifically, experts classified physics problems on the basis of the underlying physics principles that were needed to solve the problem (e.g., Newton's Second Law). For both experts and novices, Chi et al. (1981) claimed that the use of these categories elicits a knowledge structure (a schema) that functions in the representation of a problem. For experts, at least, this schema included potential solution methods.

In addition to a sorting task, a second problem-solving task was administered to a new group of novices and experts. Subjects were asked to read problems and to indicate their "basic approach" for solving the problem. Experts interpreted this basic approach as the identification of the major principles they would apply to solve the problems. This task elicited three physics principles even more consistently than the sorting task. The response of the novices to this task produced, by contrast, either the

most general kind of abstracted solution methods or the actual detailed sets of equations for solving the problem.

Research on the Problem Representations of Good and Poor Novice Problem Solvers

Differences in problem representations also exist between good and poor novice problem solvers. For example, Silver (1979) found that differences between novices rated as "good" or "poor" problem solvers were similiar to the sort of differences Chi et al. (1981) found between experts and novices. Studies of the problem-solving behavior of novices have been conducted in several domains, such as geology (Shavelson & Stanton, 1975), psychology (Fenker, 1975), and physics (deJong & Ferguson-Hessler, 1986; Thro, 1978). For example, deJong and Ferguson-Hessler (1986) investigated whether good novice problem solvers have their knowledge arranged around problem types to a greater degree than poor problem solvers.

Twelve problem types were identified on the topic of electricity and magnetism in the field of physics. These problem types were distinguished according to their underlying physics principles. For each problem type, a set of elements of knowledge containing characteristics of the problem situation was constructed. In addition, every problem type consisted of at least one element of declarative knowledge (i.e., facts, concepts, or principles) and one of procedural knowledge (i.e., knowledge of how to process or manipulate information to accomplish a task). The resulting 65 elements were printed on cards.

After completing an examination on electricity and magnetism, subjects were asked to sort the 65 cards into piles. Good problem solvers sorted the cards according to problem types, whereas poor solvers sorted to a greater extent by the surface

characteristics of the elements. DeJong and Ferguson-Hessler concluded that "an organization of knowledge around problem types might be highly conducive to good performance in problem solvers" (p.279).

Organization of knowledge changes importantly with instruction. After instruction, novices appear to organize their knowledge more like experts than before instruction, at least those novices who receive good grades in a course (Snow & Lohman, 1988).

Research on Problem Representation in Computer Programming

Unlike the excellent research on problem categorization in physics problem-solving, little empirical research exists on how computer programmers categorize problems. Several proposals regarding categorization by experts have been suggested and are summarized by Pea and Kurland (1983):

1. Function-oriented categorization. Problems would be categorized by different program-goals or functions in terms of what is to be accomplished, such as: "update inventory accounts and produce reports" (e.g., Balzer, Goldman & Wile, 1977; Shneiderman & Mayer, 1979).

2. Data/process-oriented categorization. Problems would be categorized by external object classes (e.g., updates, inventory accounts, status report) and operations (e.g., transform initial objects fo final ones) applied to specific classes of objects (Brooks, 1982; Miller & Goldstein, 1977).

3. Sequence-oriented categorization. Problems would be categorized by decomposing them into their basic components or procedures and the sequences for executing these components (e.g., Atwood, Jeffries, & Polson, 1980).

Some evidence exists on the organization of the programming knowledge that helps structure the plan representation. Similar to the Chase and Simon (1973) findings for chess experts, several studies have demonstrated that expert computer programmers perceive and remember larger chunks of information than do novices, and thus can recall more lines of a normal program-listing than can novices (Sheppard, Curtis, Milliman, & Love, 1979; Shneiderman, 1977). Consistent with findings in other domains, novices and experts show equally poor memory for scrambled listings. McKeithen, Reitman, Rueter, & Hirtle (1981) replicated and further extended these findings by investigating the details of programmers' chunks of key programming concepts. They found that experts cluster keyword commands according to programming knowledge (e.g., a cluster of words that are normally found together in a loop statement). Novices, on the other hand, associate the programming concepts with a rich variety of natural-language associations. Intermediate programmers make both programming-language and natural-language associations.

Similarly, Adelson (1981) found that recall clusters for experts were semantically based whereas those for novices were syntactically based. Furthermore, some evidence suggested that experts used a hierarchical organization based on procedural similarity. The novices, on the other hand, did not relate the items within a category in an organized way. In a subsequent study, Adelson (1984) further investigated the nature of the knowledge represention of experts and novices. Her data suggests that the more abstract problem representation of the expert contains more general knowledge about what the program does (i.e., output specifications). For example, an expert might indicate that a solution to a particular indexing problem consists of three main procedures: (a) read in the set of key terms, (b) compare the key terms in the

text, and (c) store the resulting index. According to Adelson, these elements that describe the operation of the problem are used in the representation of the problem because they are "easy to work with and easy to change" (Adelson, 1984, p.495). Consequently, they allow the expert to find an optimal solution to a problem.

Unlike the problem representations of experts, the representations of novices appear to be more concrete in that they contain information about the method the program uses to achieve a particular result (i.e., algorithms). For example, in a problem involving a sort routine, the novice would indicate a specific type of sort, such as a bubble sort.

Other than the reasearch of Adelson (1981, 1984) and of McKeithen et al. (1981), little evidence is available on the knowledge schemas of expert computer programmers. However, several proposals for schemas have been suggested, and are summarized by Pea and Kurland (1983):

1. An expert's programming knowledge schemas might include anything from transactions (less than a programming statement) to chunks (units that accomplish particular goals) to higher level chunks (familiar algorithms) (Mayer, 1981).

2. The expert's schemas might be organized in a hierarchy of patterns from operations (compare two numbers) to small algorithms (sum an array) to large algorithms (bubble sort) to program patterns (Shneiderman, 1980; Shneiderman & Mayer, 1979).

3. Schemas might include known solutions/plans/plan elements (Atwood, Jeffries & Polson, 1980; Balzer & Wile, 1977; Miller & Goldstein, 1979).

4. The expert's programming knowledge schemas might contain high-level programming units, such as loop and recursion structure (Rich, 1981; Rich & Shrobe,

1979; Soloway & Woolf, 1980; Soloway, Ehrlich, Bonar, and Greenspan, 1982).

5. Other theorists have proposed that the expert's schemas contain building block units such as basic loop, augmentation, and filter (Waters, 1979).

6. In addition, schemas might include categories with internal structure, such as rules for data structures, enumerations (looping constructs), mappings, etc. (Barstow, 1977).

Problem Solving Plans and Achievement

In the previous section, research on the problem representations of novice and expert computer programmers was summarized. In the present study, problem representation differences between novice and intermediate programmers in an introductory computer programming course in Pascal were investigated. In addition, this study explored problem-solving plans and student achievement in the context of the course.

One of the most relevant studies in the computer science literature is probably a study by Soloway, Ehrlich, Bonar, and Greenspan (1982) on learning Pascal. Soloway et al. (1982) attempted to identify the needs of novice programmers by understanding the source of their programming difficulties. One topic of investigation was bugs and misconceptions about loops. These researchers posed the following question: "Does choosing the appropriate loop construct (i.e., for, repeat, or while, depending on context) for a given problem facilitate the production of a correct program solution?" (p.43). Soloway and his colleagues reasoned that if a student chooses the correct loop construct, then the student might know more about programming, and thus be more likely to write a correct program; alternatively, the

loop construct itself might impose constraints that enable the student to identify the components of the loop.

Results indicated that the students (novices and intermediates) who chose the appropriate loop construct were only slightly more likely to write correct programs than were students who chose an incorrect loop construct. Therefore, choice of the appropriate loop construct (e.g., a "for" loop) was not a good predictor of program correctness. However, choice of the appropriate looping strategy was a predictor of correct programs. An example of a looping strategy is the Read/Process Strategy. It specifies that "the actions that 'read a value, then process it' are nesting in a repetition loop" (Soloway et al., 1982, p.34).

Research on problem representations and solution plans has been presented. In this dissertation, the influence of prior computer experience on learning a new computer language also was explored. In the following section, this topic will be discussed from both teaching and research perspectives.

Prior Programming Knowledge and Learning a New Language

Instructional Research

Instructors of programming languages agree that prior programming experience influences the learning of subsequent programming languages. This phenomenon is particularly obvious when teaching an introductory computer language course to a class composed of experienced programmers and students with no prior computer programming experience.

Shneiderman and Mayer (1979) maintain that it is unwieldy to teach non-programmers and programmers a new language in the same course. They explain

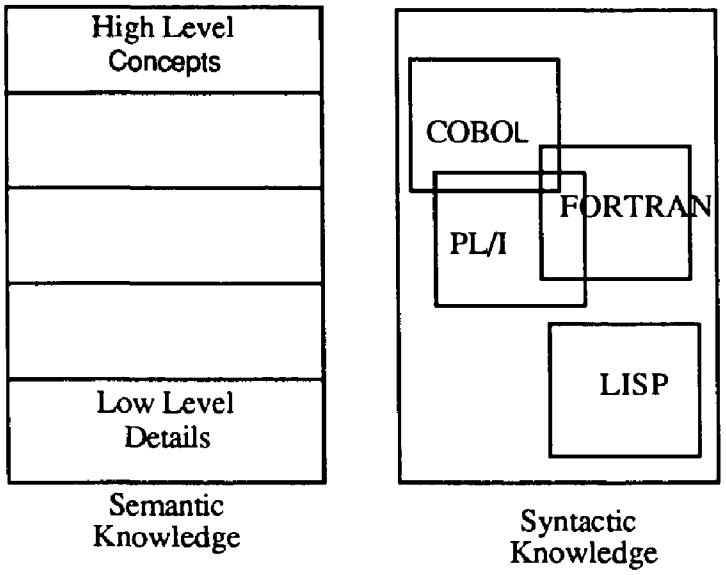
their theory in relation to semantic and syntactic knowledge in long-term memory.

Syntactic knowledge refers to a user's knowledge of the basic elements of language code including line numbers, key words, variable names, numeric values, arithmetic symbols, logical symbols, and punctuation. Syntactic knowledge also includes the knowledge of rules for ordering the basic elements of the language code within a line of code and rules for ordering lines of code (Dyck & Mayer, 1989). According to Shneiderman and Mayer, this knowledge is more precise and arbitrary. Consequently, syntactic knowledge is more easily forgotten than semantic knowledge, which is generalizable over different syntactic representations.

Semantic knowledge refers to the meaning of a statement or of a program module. This knowledge is more generic than syntactic knowledge. It has been gained through programming experience and instruction. Semantic knowledge has been abstracted from these experiences and stored as sets of general, meaningful information that are more or less independent of the syntactic knowledge of particular programming languages. For example, an experienced programmer knows that sequencing and iteration are relevant in all programming languages. This distinction between syntactic and semantic knowledge in long-term memory is summarized in Figure 2 (from Shneiderman & Mayer, 1979).

The results of several studies indicate that previous programming language experience influences the learning of subsequent programming languages (Lucas & Kaplan, 1974; Sheppard, Curtis, Millman, & Love, 1979; Shneiderman & Mayer, 1979; Ricardo, 1983; Taylor & Mounfield, 1989). For example, Ricardo reported a correlation of .16 (significant at .01 level) between prior programming experience and introductory course achievement.

Figure 2. Programmer's Knowledge in Long-term Memory



Note. From "Syntactic/semantic interactions in programmer behavior: A model and experimental results" by B. Shneiderman and R.E. Mayer, 1979, International Journal of Computer and Information Sciences, *8*, p. 223. Copyright 1979 by Plenum Publishing Corporation. Reproduced by permission.

In summary, there seems to be agreement between computer educators and researchers that prior computer programming influences the learning of subsequent programming languages. However, little empirical research exists on the cognitive structures and processes which may transfer when an additional programming language is learned.

Transfer

The subject of transfer in the programming literature has been approached mainly

from the perspective of the question, "What problem-solving skills from programming transfer to other subjects or tasks?". Although a few studies have reported positive transfer, most report no transfer between programming and other types of problem-solving. For reviews, see Blume (1984), Dalbey and Linn (1985), Pea and Kurland (1984), and Salomon and Perkins (1987).

As previously mentioned, one focus of this exploratory study is on transfer from one programming language to another. An informal assessment on this topic was reported by Linn (1985). From a group of 2400 high school students studying BASIC, twenty-four students comprising the top 1% were selected. These students were introduced to an elementary programming language called Spider World (Dalbey, 1983). After a brief introduction, students attempted to write three programs. According to Linn, many of these students handled the programming problems with obvious procedural skill. In a few cases, students used general plans seemingly generalized from BASIC.

A study by McKeithen (1979) also reports some preliminary evidence of cognitive structures that transfer when learning a new programming language. McKeithen studied the memory organization of computer programmers. The purpose of her study was to determine the content of a chunk or memory unit, formed from ALGOL W reserved words, thereby displaying their organization. Reserved words are names reserved for a specific purpose in the programming language. For example, the word "string" cannot be used as a user variable.

McKeithen hypothesized that chunk content would vary with experience level. Twenty-two subjects participated in this study: seven ALGOL experts, six novices (one course in ALGOL), eight naive subjects, and one general expert. The general

expert had no hours of programming in ALGOL W; however, he was familiar with the language. In addition, he met the programming experience criteria for expertise (2000 hours of programming experience).

The results from this study suggested that experts have similar memory structures. Memory structure was estimated by applying a clustering algorithm to each subject's recall data. The procedure outputs a tree-structure which summarizes the orgnization of concepts for the subject. McKeithen conjectures that the clustering or memory trees of experts look alike because the experts used the same basis of organization for the ALGOL W reserved words--the ALGOL W statements. Statements refer to algorithmic actions which are executable. The particularly relevant finding, however, is that the memory tree of the general expert (no prior ALGOL programming) was more like the memory trees of the experts than like the memory structures of the novices (one course in ALGOL). The general expert used statements from either other programming languages or ALGOL W to organize the ALGOL W words. McKeithen suggests that one effect of experience on programmers is the habit of thinking in terms of language structures. Since only one general expert participated in this study, further research is needed to confirm and extend these findings.

Is prior computer experience the main variable influencing achievement in an introductory programming course? Perhaps computer-programming aptitude is important as well.

Metatheoretical Framework: Process Theory of Aptitude for Learning from Instruction

A process theory of aptitude for learning from instruction (Snow & Lohman, 1984) provides a context for exploring the relationship between computer learning and

aptitude. Aptitude processes are defined as "those predictable, directed changes in psychological functioning by which some individual learners adapt to the short-term and long-term demands of instructional conditions while others do not" (Snow, 1980). The term "aptitude" also implies that the individual difference construct of interest is not merely correlated with success in the treatment but is needed as a preparation to achieve that success. In educational environments, then, aptitude refers to readiness to learn from a particular instructional method (Snow & Lohman, 1984). This theory is unique in that it attempts to account for changes in both stable and dynamic systems: stable cognitive changes reflect modifications in the information being processed, due to the sequence of operations performed; dynamic cognitive changes refer to changes that result from the processing of incoming information and the monitoring of that information. To account for learning, the cognitive system has to be dynamic.

Snow and Lohman propose that both ability test and learning task performance involve not only stable system change, but also adaptive or dynamic system change. Furthermore, they hypothesize that the most important aspects of the dynamic system are assembly and control processes--"higher order strategic processes involved in the organization, reorganization, and monitoring of component processes within a task" (p.11). Individual differences in these higher-order processes are predicted to be a principal source of ability-learning correlations in education. Snow and Lohman interpret these ability-learning correlations as signifying aptitude transfer--following old leads from Ferguson (1954, 1956) and Hunt (1961).

Higher-order strategic processes are of particular importance in complex ability tests and learning tasks. The work of Glaser (1980) supports this hypothesis:

> A reasonable prediction is that individual differences in the cognitive components of aptitude measures will be more effectively analyzed as the result of variations in higher level strategies than as a result of the more molecular aspects of elementary processes such as speed of retrieval from STM [short-term memory]. These higher-level strategies will interact with knowledge-based declarative and procedural information to yield the cognitive basis of individual differences in cognitive competence and style. (pp.313-314)

Aptitude processes need to be understood at both the micro and molar level. Micro level refers to the changes that occur during learning or information processing that are discernable in short time-frames (seconds, minutes, etc.), whereas molar level refers to changes during long time-frames (week-to-week, month-to-month, etc.) (Snow, 1981). These molar changes that occur over accumulated instruction are referred to as "accretion", "restructuring", and fine-tuning of organized knowledge and skill (Rumelhart & Norman, 1978). This study will deal with changes that occur at both micro and molar levels (e.g., from a four-minute free-association task in Study 3 to a final semester exam).

Computer Programming Aptitude

Since computer courses were first introduced into universities, computer educators and researchers have hypothesized about potential computer programming aptitudes. As previously mentioned, several researchers have noted that prior programming experience seems to influence learning a new programming language. What other variables appear to be strong predictors of programming achievement?

Empirical evidence suggests that math training and/or ability correlate with college computer science performance (e.g., Peterson & Howe, 1979; Butcher & Muth, 1985; Kovalina, Wileman & Stephans, 1983; Alspaugh, 1972). Other researchers argue that command of one's native language correlates with programming success (Kurtz, 1980;

Wills, 1982). Empirical evidence by Sauter (1986) suggests that both mathematical aptitude and language aptitude are associated with programming skill. Math aptitude was associated with the ability to learn rules of logic, whereas language aptitude was associated with an ability to learn syntactic rules.

A model for predicting success in an Introduction to Computers course was developed by Peterson and Howe (1979). Their research indicated that only college GPA and general intelligence (as measured by the General Aptitude Test Battery, GATB) contributed significantly to the model. Forty percent of the variance in course grade was explainable by their model.

Kurtz (1980) had difficulty confirming and extending the model proposed by Peterson and Howe. A substantial number of students in an introductory class are either freshmen or transfer students; therefore, college GPA data was not a good variable to use for prediction purposes. Also, Kurtz had little success with performance measures similar to IQ-type items. Consequently, Kurtz constructed an "intellectual development" measure for his prediction study on introductory computer programming achievement. His test contained fifteen items in ten areas of formal reasoning. Students were classified as one of the following: late concrete reasoners, early formal reasoners, or late formal reasoners. Kurtz found that late concrete reasoners received poor test grades, whereas late formal reasoners received outstanding grades.

Ricardo (1983) validated a new measure of computer-programming aptitude, the Programming Readiness Test. This test consists of three parts: deductive reasoning, persistence, and inductive reasoning. Each of the three parts correlated with an exam score ($\underline{r}$ = .34 to .44) and final grade ($\underline{r}$ = .29 to .39) for an introductory computer

programming course. (For a comprehensive review of the history of computer-aptitude tests used in business and academic environments, refer to Ricardo, 1983.)

In summary, researchers have found that computer programming aptitude is related to some degree to math training and/or ability, language aptitude and native language achievement, grade-point average, general intelligence, formal reasoning, persistence, deductive and inductive reasoning.

The underlying assumption of traditional prediction studies is that undifferenciated "faculties" or "powers", such as general intelligence or math aptitude, transfer to the new computer language environment and are instrumental in computer programming achievement. This approach yields useful information for screening purposes. It addresses the student's aptitudes and the results of the student's behavior but does not shed light on the processes involved in the acquisition of computer programming skill. That is, the traditional approach does not inform us about the possible trainable information processing skills and strategies essential for computer programming achievement. This dissertation research includes the traditional regression approach to the study of computer programming aptitude (Study 1) as well as an information processing approach (Studies 2-4). (For a discussion of the information processing approach to the study of abilities, refer to Snow and Yalow (1982).)

Computer-Programming Aptitude: Inductive Reasoning

The focus of this study is aptitude for computer programming. More specifically, inductive reasoning will be the aptitude to be investigated, since transfer of knowledge

and strategies depends to some extent on inductive thinking skills. An individual may have relevant background knowledge yet may not be able to connect such knowledge to the learning of a computer programming language. Tasks and tests designed for measuring inductive thinking may serve as a predictor of computer programming development and of the quality and degree of transfer outcomes.

In this study, Raven's Advanced Progressive Matrices (Raven, 1977) is used to measure inductive reasoning. Success on this complex reasoning task depends upon the learner's ability to adapt flexibly what he or she has learned from previous problems in the test to new, more difficult problems.

Fluid and Crystallized Abilities

Another name for inductive reasoning is fluid ability (Gf). It is facility in reasoning, particularly where adaptation to new situations is required. Raven's Advanced Progressive Matrices (1977) is a standard task for measuring fluid ability (Gf). Gf generally predicts learning in novel instructional situations. For example, an art major who decides to become a business major will be required to develop new ways of thinking in order to solve business course-related problems. In this study, high Gf refers to people who had the highest scores on the Raven's task--a test of inductive reasoning aptitude. Average Gf refers to those students who scored the lowest on the Raven's task.

Crystallized ability (Gc) represents "a coalescence of organization of prior knowledge and educational experience into functional cognitive systems for retrieval and skilled application to aide further learning in future educational situations" (Snow, 1980, p.58). For example, biology graduate students build upon their knowledge

base from their undergraduate training. According to Snow and Yalow (1982), "the transfer producing this coalescence need not be only of specific knowledge but also of organized processing strategies we think of as academic learning skills" (p.519). In this study, crystallized ability (Gc) is represented as the experience or prior learning dimension. More specifically, high Gc refers to people who have programming experience and who are currently learning a new language. Low Gc refers to people who have no or little prior experience with programming.

This information processing approach to Gf and Gc is relatively new (Snow & Yalow, 1982). In the psychometric tradition, Gf and Gc were discussed as undifferentiated powers that transfer. The current approach is based on ideas from Hunt (1961) and from Ferguson's (1954, 1956) early view that abilities develop as a function of learning-to-learn and transfer. Locke and Dewey's themes are woven into the new approach as well. Snow and Yalow summarize this promising approach to abilities:

> Ability is attained through experience over time and consolidated through exercise. Skill in one kind of task performance transfers to performances on other tasks as a function of the similarity between tasks. Abilities thus develop as transfer relations within a class of tasks. Aptitude for learning, then, is readiness to transfer prior ability to new performances on similar tasks. In the new language, however, it is information processing skills and strategies rather than undifferentiated "powers" or "faculties" that transfer. (pp.516-517)

These information processing skills range from estimates of elementary cognitive processes (such as speed of retrieval of information from LTM) to general problem-solving strategies (such as means-ends analysis). Thus,

> . . . . there may be elementary information processes that are common to the performance programs for different tasks. These can account for some of the relations among tasks that we take as indicative of transferable ability. However, there may also be higher-order processes that learn to assemble

and transfer relations among performance tasks. (p.517)

Theory of Field Achievement

In addition to testing hypotheses related to a theory of aptitude, it is hoped that this study will contribute to a theory of field achievement. The purpose of a theory of field achievement is "to understand the facets of learning outcome, particularly those leading to retention and transfer, that constitute sought-after changes in content and process structures for the learner" (Snow & Lohman, 1984, p.353). For this study, a theory of field achievement would include an understanding of cognitive accomplishments from learning programming, particularly those leading to retention and transfer to subsequent computer language courses and other problem-solving areas. A model developed by Linn (1985) and her colleagues includes the major components of a theory of field achievement for computer programming: (a) learning the computer language features or syntax; (b) learning how to design programs to solve problems; (c) learning problem-solving skills applicable to other formal systems. Table 1 gives greater detail on the main features of the model.

This study investigates many of the outcomes outlined in Linn's model on cognitive accomplishments from computer learning. More specifically, this study investigates whether students with previous computer language experience use generalized templates in learning a new computer language. Furthermore, it investigates whether the degree of transfer is influenced by individual differences in inductive reasoning ability (Gf) as measured by Raven's Advanced Progressive Matrices.

Table 1. Potential Chain of Cognitive Accomplishments from Learning Programming

1. Learn the language features (syntax).
2. Learn to design programs to solve problems.
-- Develop a repertoire of templates.
-- Develop procedural skills to combine templates or language features to solve problems.
3. Learn problem-solving skills applicable to other formal systems (e.g., a new programming language).
-- Develop a repertoire of generalized templates suitable for adaptation to new formal systems (e.g., an efficient sort template adapted to another programming language).
-- Explicitly identify generalized procedural skills for planning, testing, and reformulating problems in a variety of formal systems.

Note. From "The Cognitive Consequences of Programming Instruction in Classrooms" by M.C. Linn, 1985, Educational Researcher, 14 (5), p.16, Washington, D.C.: American Educational Research Association. Copyright 1987 by the American Educational Research Association. Adapted by permission.

Overview of Research Purposes

The purpose of this dissertation is to investigate the differential effects of computer programming experience and inductive reasoning abilities on problem

representations, solution plans, and achievement of students enrolled in an elementary computer programming course. More specifically, the goals of this research are as follows:

1. To determine whether individual differences in inductive reasoning abilities and prior computer experience make independent contributions to the prediction of examination scores and course grades in an introductory Pascal course (Study 1). The results from Study 1 set the stage for a more in-depth examination of the structures and processes involved in solving problems in computer programming.

2. To determine in greater detail the particular contributions of experience and inductive reasoning abilities to students' categorization of programming problems (Study 2). Do experienced programmers use generalized templates when categorizing problems? Is degree of transfer influenced by individual differences in inductive reasoning skills?

3. To investigate the particular contributions of experience and inductive reasoning to the contents of students' problem schemas (Study 3).

4. To examine the contributions of experience and inductive reasoning to the problem-solving plans and solutions they construct (Study 4).

5. To use the above findings to help build a process theory of computer programming aptitude and achievement.

CHAPTER III

STUDY 1

The purpose of this study was to determine if inductive reasoning abilities and prior computer programming experience make significant independent contributions to the prediction of examination scores and course grades in an introductory programming course.

Method

Subjects

Subjects were 55 male and 27 female paid volunteers who were recruited from a class of 263 students enrolled in an introductory Pascal course. This course included both students with no prior computer language experience and students who had previous experience with other computer languages, but who were beginners in learning the Pascal computer language. Further descriptive information on this sample is presented in Tables 3 and 4 below.

A Learner Characteristics Questionnaire was administered to all 263 student volunteers. The 222 students who indicated that the researcher could have access to their grades were invited to complete two additional tests at $3.50 per hour. However, only 82 of the 222 subjects were interested in participating. These subjects completed the Word Problem Translation Test (Mayer & Dyck, 1984) and the Advanced Raven's Progressive Matrices Sets 1 and 2 (Raven, 1977).

Instruments

Learner Characteristics Questionnaire. This survey instrument was used to gather information about entry characteristics of the students, including access to standardized test scores (e.g., ACT). A copy of the Learner Characteristics Questionnaire is presented in Appendix A. Data from the Learner Characteristics Questionnaire pertaining to ACT assessment scores were obtained from the registrar's office for students who gave permission. Grade-Point Average (GPA) for senior high school and college students were self-reported on the Learner Characteristics Questionnaire. Course grades were obtained from the teaching assistants for the course.

Word Problem Translation Test (Mayer & Dyck, 1984). This three-minute, multiple-choice test consists of six word problems. The subject's task is to select the correct equation corresponding to each problem. In a series of four studies involving 196 subjects, Mayer, Dyck, and Vilberg (1985) found that performance on the Word Problem Translation Test correlated $\underline{r}$ =.54 with a BASIC programming language post test. (Four correlations were computed; $\underline{r}$ = .54 is a mean correlation weighted for number of subjects.)

Raven's Advanced Progressive Matrices Sets 1 and 2 (Raven, 1977). This test is designed to assess inductive reasoning ability or fluid ability (Gf). The examinee is required to solve problems presented in abstract figures and designs.

Procedure

Students enrolled in the introductory computer course in Pascal were told: "The purpose of this research is to study how differences in previous learning and aptitude

affect learning a new computer programming language. We are researching this topic for the purpose of improving computer language instruction."

The Learner Characteristics Questionnaire was completed by 263 student volunteers. This process took approximately ten minutes for students with no prior computer programming experience and fifteen minutes for students with prior computer programming experience. The experienced programmers needed an additional five minutes to complete eleven questions concerning their programming experience. Subjects were told: (a) that they could omit any questions, (b) that they could discontinue their participation at any time, (c) that the data obtained would be used only for research purposes, (d) that names would be used only to match data with class-related findings, and (e) that course grades would not be influenced by any findings in the questionnaire.

Eighty-two students, who gave the researcher permission to have access to their grades, completed two additional tests. They were paid $3.50 per hour. Both the Word Problem Translation Test and the Raven's Advanced Progressive Matrices Sets 1 and 2 were administered using standard procedures. The Word Problem Translation test took approximately 5 minutes to administer. The Raven's Advanced Progressive Matrices Sets 1 and 2 took approximately 50 minutes. Of the 82 subjects, seven students were not included in Study 1 because they had prior Pascal experience. Therefore the number of students who participated in Study 1 was 75.

How representative are the subjects in Study 1? The participants in Study 1, as a group, appear to be more able than the population of students enrolled in the course. The dropout rate was 44% for the class as a whole as contrasted with 29% for the participants in Study 1. A comparison of exam scores between the subjects in Study 1

and the course participants as a whole is shown in Table 2. The mean grade for each group on all exams was C. The Study 1 students, on the average, had slightly higher scores (4-9 points) than the course average scores.

Results and Discussion

Descriptive information about the sample is first presented, followed by a discussion of the main dependent variable, Exam 1. Next, the intercorrelations among dependent variables are presented, followed by the correlations among subject variables, and then correlations between Exam 1 and the subject variables. Lastly, the regression analyses for Exam 1 and Course Grades are presented.

In an attempt to replicate and extend the findings of the researchers previously discussed (e.g., Butcher & Muth, 1985; Kurtz, 1980; Mayer, Dyck & Vilberg, 1985; Peterson & Howe, 1979; Ricardo, 1983; and Sauter, 1986), measures of math, English, reasoning, programming experience, grade-point average, student status, college major and gender were included in this study. Additional variables hypothesized to be related to success in introductory programming were included. For example, responses related to motivation on the Learner Characteristics Questionnaire (items 9, 10, and 11) were included in the analysis.

Table 3 shows the mean scores and standard deviations for the major subject classification variables. The typical student in Study 1 had an ACT Math score of 26.37, an English score of 22.51, Progressive Matrices (inductive reasoning) score of 26.64, Word Problem Translation score of 3.58 (out of a possible 6 points), College Grade-Point Average of B-, High School Grade-Point Average of B, reported 2.5 semesters of college math, 6.4 semesters of high school math, and was 20 years old.

Since ACT Math and inductive reasoning are two of the main independent variables in Study 1, some additional data are useful. The mean ACT Math achievement score of the subjects in Study 1 was approximately one standard deviation above the 1988 national mean of 17.2 points (SD = 7.8). Similarly, the inductive reasoning skills of this group were also above the national average, mean = 21 points (SD = 4). Although, as expected, the sample was above average in ability, scores on both ACT Math achievement and the Progressive Matrices test are within an acceptable range.

Table 4 shows the frequency counts for the major categorical subject classification variables. For the variables related to computer experience, 76% of the students

Table 2. Mean Scores and Standard Deviations for Course Exams

Variable	N	Mean	Standard Deviation
Exam 1 Scores			
Study 1 Subjects	63	101.65	31.80
Course Participants	216	95.70	31.89
Exam 2 Scores			
Study 1 Subjects	54	105.09	27.18
Course Participants	159	102.08	28.64
Final Scores			
Study 1 Subjects	54	151.09	47.60
Course Participants	153	152.40	38.96

Note: Exam 1 Scores 75-109 = C, Exam 2 Scores 84-110 = C, Final 120-159 = C

participating in Study 1 had more than one semester of high school BASIC programming experience; 45% had prior top-down design experience; 75% had prior personal computer experience. The motivation level of the group was high: 80% of the students surveyed planned to take Advanced Pascal, 95% expectedto use programming in their future jobs, and 80% indicated that the Introductory Course in Pascal was a high priority for them. English was the native language for 87% of the

Table 3. Mean Scores and Standard Deviations for Relevant Independent Variables

Variable	N	Mean	Standard Deviation
ACT Math	63	26.37	4.08
ACT Verbal	63	22.51	4.52
Inductive Reasoning (Advanced Raven Progressive Matrices Set 2)	75	26.64	3.67
Word Problem Translation Test (6=highest possible score)	74	3.58	1.30
College Grade-Point Average (Self-reported)	60	2.99	0.57
High School Grade-Point Average (Self-reported)	68	3.32	0.44
No. of Semesters College Math Courses	56	2.50	2.31
No. of Semesters High School Math Courses	72	6.40	1.94
Age 69		20.62	5.54

students. Two-thirds of the students were men. There were more freshmen than any other category, with approximately the same number of sophomores, juniors, and seniors.

Data from the Learner Characteristics Questionnaire pertaining to prior computer experience was used to determine the computer experience group. The computer experience groups were defined as follows:

Group 1 = Novice--prior computer programming experience or only 1 semester of high school BASIC or COBOL.

Group 2 = Advanced Novice--two semesters of high school computer programming or one semester of college-level programming.

Group 3 = Intermediate--two semesters of college-level programming (two different languages) or three or four semesters of high school programming in one language plus additional programming experience.

Group 4 = Advanced Intermediate--two to six semesters of college-level programming in at least one language. Most people had studied two or three languages. Many had work experience.

An independent rater (an experienced programmer) chose the subjects for the intermediate and advanced intermediate groups. For these intermediate groups, the rater was told to give more weight to languages related to Pascal (e.g., FORTRAN) than to languages unrelated to Pascal (e.g., LISP).

Of the 82 subjects, seven students were not included in Study 1 because they had prior Pascal experience. The remaining 75 students were divided into four groups based on level of computer experience as defined above.

Table 5 shows the number of subjects in each experience group at the time the

Table 4. Frequency Counts and Percents for Categorical Subject Variables

Variable	Frequency	Percent
Prior Computer Programming Experience		
Novice	18	24.0
Advanced Novice	24	32.0
Intermediate	20	26.7
Advanced Intermediate	13	17.3
Prior Top-Down Design Experience		
Yes	28	37.3
No	33	44.0
No Response	14	18.7
Prior Personal Computer Use		
Never	18	24.0
Sometimes	37	49.3
Frequently	20	26.7
Prior Word Processing Experience		
Yes	53	70.7
No	22	29.3
Plan To Take Advanced Pascal		
Yes	59	78.7
No	15	20.0
No Response	1	1.3
Future Job Using Programming		
Yes	71	95.0
No	4	5.0
Pascal Course -- High Priority		
Low	0	
Medium	15	20.0
High	60	80.0
English Native Language		
Yes	65	86.7
No	10	13.3

Table 4--continued

Sex			
	Female	25	34.0
	Male	50	66.0
Status			
	Freshman	27	36.0
	Sophomore	17	22.7
	Junior	15	20.0
	Senior	14	18.7
	Special Students	2	2.7

study was administered and the number of subjects who completed Exam 1. A few subjects in each experience category dropped the course prior to Exam 1. That is, of the 75 students participating in this study, 63 took Exam 1.

Analysis of the Dropout Group

Of the 75 subjects who participated in Study 1, 21 subjects or 28% dropped the course. To determine whether any of the predictive measures could be used to explain this, T-tests for independent groups were performed for differences between the means on all variables in the drop and non-drop groups.

Significant differences were found for only two variables. The drop group had significantly fewer college math courses ($T = 7.49$, $p > = 0.0001$). In addition, completing the Pascal course was a higher priority for the drop group than for the non-drop group ($T = 2.88$, $p > = 0.01$). So it cannot be asserted that the drops were not interested in completing the course. They were very interested in pursuing computer science when they filled out the Learner Characteristics Questionnaire at the beginning of the semester.

Table 5. Number of Subjects in Each Group Prior to Exam 1 and Number Who Completed Exam 1

Experience Group	No. Prior to Exam 1	Percent of Total	No. Completed Exam 1	Percent of Total
1. Novice	18	24	14	22
2. Advanced Novice	24	32	21	33
3. Intermediate	20	26.7	17	27
4. Advanced Intermediate	13	17.3	11	18
	75		63	

Since the dependent variable Exam 1 was the main outcome variable in the study, a one-way analysis of variance test was performed on the differences between the drops and non-drops on Exam 1. As expected, the non-drops performed significantly better on Exam 1 than the drops ($T = 24.46$, $p> = .0001$).

Analysis of College Major

Table 6 shows a breakdown of college major for the subjects who took Exam 1. The Pascal course was required for the 29 computer science majors. Since college major is an unstable variable during the first year or two of college, no additional analyses were made.

Main Dependent Variable: Exam 1

Exam 1 is emphasized here because it is the main dependent variable, not only for Study 1 but also for Studies 2 through 4. The period between enrollment in the

introductory computer programming course in Pascal and Exam 1 is of particular importance because this study was investigating the early stages of transfer from prior computer programming experience to learning an additional programming language. The role of inductive reasoning skills during this stage of the semester was also of interest. As previously mentioned, 46% of the students initially enrolled in the course dropped out either before Exam 1 or just after they received their Exam 1 results.

A better understanding of the trainable aspects of successful performance during the early stages of the course could lead to changes in the course or in assistance offered to students such that more students might successfully complete the course.

The purpose of Study 1 was to estimate the independent contributions of inductive reasoning and prior computer programming experience to the prediction of scores on the first examination in the course. Study 1 thus provides the context for understanding the results from Studies 2 through 4. Correlation data for Exam 2 Scores, Final Exam Scores and Course Grades are also presented. The main focus of discussion, however, in Study 1 is on Exam 1.

Intercorrelations Among Dependent Variables

The intercorrelations among the dependent variables are shown in Table 7. In part, the magnitude of the correlation between grade and each exam can be explained by the part-whole relationship (i.e., grade includes each exam score).

Intercorrelations Among Independent Variables

The intercorrelations among subject variables are reported in Table 8. In preparation for the regression study, these correlations were examined for collinearity. Strong correlations were not found between the main subject classification variables

Table 6. The College Majors of Subjects Taking Exam 1

Major	N	Percent of Total Group
Computer Science	29	38.7
Math	5	6.7
Social Science	2	2.7
Business	9	12.0
Art	1	1.3
Health Science	2	2.7
Other	14	18.7
No Response	13	17.3

Table 7. Intercorrelations for Course Exams and Grade

	EXAM 1	EXAM 2	FINAL	GRADE
EXAM 1				
EXAM 2	.70			
FINAL	.62	.64		
GRADE	.66	.64	.81	

Note. N = 54. Exam 1 was administered after 5 weeks, Exam 2 after 9 weeks, and the Final Exam after 15 weeks.

that were later used as independent variables in the regression analysis.

The correlation and regression analyses related to the first exam are of particular interest and will be presented first and discussed. This will be followed by a briefer presentation and discussion of the analyses on Course Grade.

Correlations: Exam 1 and Independent Variables

Major Variables and Exam 1. Previous research findings focused on predicting final examination scores and final grades, not on earlier examination scores in the semester. As expected, many of the variables used in previous studies were found to be correlated with first examination scores in this study (i.e., math achievement, word problem translation, inductive reasoning, computer experience and grade-point average.)

Table 9 reports correlations between all subject classification variables and the four dependent variables. For the correlations between Exam 1 and the independent variables, see column 1. The major variables include ACT English, ACT Math, Progressive Matrices (inductive reasoning), programming experience group, Word Problem Translation Test, grade-point average, number of semesters of college math, and personal computer use. Correlations between Exam 1 and these variables are significant at least at the .05 level, except ACT Verbal. ACT Verbal was included for comparison purposes even though the correlation was small.

Inductive reasoning (Progressive Matrices) and computer programming experience were the two major subject variables of interest in this research. A comparison between these subject variables and dependent variables is shown in Figure 3.

Table 8. Correlations Among All Subject Variables

	1	2	3	4	5	6	7	8	9
1 ACT Verbal									
2 ACT Math	.49 (63)								
3 Progressive Matrices	.21 (63)	.37 (63)							
4 Programming Experience	-.02 (63)	-.03 (63)	.11 (75)						
5 Word Problem Translation	.32 (63)	.39 (63)	.35 (74)	-.12 (74)					
6 Grade Point Average	.45 (51)	.46 (51)	.09 (60)	.03 (60)	.18 (59)				
7 No. College Math Courses	-.01 (47)	.11 (47)	.00 (56)	.25 (56)	.16 (55)	.01 (53)			
8 Personal Computer Usage	-.16 (63)	-.12 (63)	.24 (75)	.51 (75)	-.06 (74)	-.19 (60)	-.05 (56)		
9 High School Grade Point	.33 (63)	.28 (63)	-.02 (68)	.07 (68)	-.09 (68)	.55 (55)	.14 (51)	-.09 (68)	
10 No. High School Math Courses	-.10 (62)	.20 (62)	.17 (72)	-.11 (72)	.09 (71)	.19 (58)	.06 (54)	-.03 (72)	.15 (67)
11 English Native Language[a]	.24 (63)	.29 (63)	.19 (75)	-.05 (75)	.17 (74)	-.14 (60)	-.13 (56)	.18 (75)	.1 (68)
12 Course High Priority	.16 (63)	-.01 (63)	-.15 (75)	-.10 (75)	-.15 (74)	-.06 (60)	-.27 (560	-.10 (75)	-.10 (68)
13 Advanced Pascal Plans[a]	-.20 (62)	-.35 (74)	.11 (74)	.21 (73)	-.13 (59)	-.14 (55)	-.32 (74)	.30 (67)	-.29 (72)
14 Future Job using Programming[a]	-.12 (63)	-.26 (63)	-.27 (75)	.09 (75)	-.17 (75)	-.22 (60)	-.09 (56)	.09 (75)	-.14 (68)
15 Prior Word Processing[a]	-.12 (63)	-.02 (63)	.20 (75)	.38 (75)	.21 (74)	.07 (60)	.26 (56)	.48 (75)	-.12 (68)
16 Prior Top-down Design[a]	-.08 (57)	.16 (57)	.08 (69)	.08 (69)	.02 (68)	.32 (54)	-.25 (51)	.17 (69)	.35 (62)
17 Age	.09 (60)	-.09 (60)	.01 (71)	-.05 (71)	.04 (70)	-.26 (58)	.22 (53)	-.16 (71)	-.31 (64)
18 Sex[b]	-.03 (53)	-.10 (53)	-.16 (61)	-.24 (61)	.20 (60)	-.04 (51)	-.08 (45)	-.31 (61)	.02 (56)

Table 8--continued

	10	11	12	13	14	15	16	17
10 No. High School Math Courses								
11 English Native Language[a]	-.10 (72)							
12 Course High Priority	-.01 (72)	.14 (75)						
13 Advanced Pascal Plans[a]	.14 (74)	-.10 (74)	.09 (74)					
14 Future Job using Programming[a]	-.14 (72)	.08 (75)	.06 (75)	.03 (74)				
15 Prior Word Processing[a]	-.02 (72)	.18 (75)	-.25 (75)	-.03 (74)	-.02 (75)			
16 Prior Top-down Design[a]	-.27 (58)	.13 (61)	.11 (61)	-.07 (60)	-.02 (61)	.32 (61)		
17 Age	-.17 (66)	.06 (69)	.09 (69)	.01 (68)	.04 (69)	.00 (69)	.18 (55)	
18 Sex[b]	-.12 (68)	.00 (71)	-.08 (71)	.14 (70)	-.05 (71)	.08 (71)	.02 (62]	-.02 (65)

Note. Sample sizes are in parentheses.
[a] Coded No=0, Yes=1.
[b] Coded Female=0, Male=1.

Level of inductive reasoning (Gf) was the stronger predictor of achievement for the first two-thirds of the course. By the final exam, inductive reasoning was no longer a significant correlate for predicting computer programming achievement, whereas prior computer programming was significant. The contribution of prior experience was similar across all three exams. Neither inductive reasoning nor computer programming experience correlated significantly with course grade. Consistent with prior research, college grade-point average (self reported) was a

Table 9 . Correlations Between Dependent and Subject Variables

	EXAM 1	EXAM 2	FINAL	GRADE
ACT Verbal	.24 (52)	.37 (46)	.16 (46)	.19 (46)
ACT Math	.39 (52)	.48 (46)	.20 (46)	.22 (46)
Progressive Matrices	.39 (63)	.45 (54)	.10 (54)	.04 (54)
Programming Experience Group	.30 (63)	.25 (54)	.31 (54)	.15 (54)
Word Problem Translation	.38 (62)	.43 (53)	.12 (53)	.20 (53)
Grade-Point Average	.28 (52)	.43 (45)	.20 (45)	.42 (45)
No. College Math Courses	.29 (48)	.25 (40)	.18 (40)	.20 (40)
Personal Computer Usage	.27 (63)	.10 (54)	.12 (54)	-.04 (54)
High School Grade-Point Average	.08 (57)	.14 (50)	.12 (50)	.13 (50)
No. High School Math Courses	.19 (60)	.20 (51)	.29 (51)	.26 (51)
English Native Language[a]	.25 (63)	.06 (54)	.06 (54)	-.12 (54)
Pascal Course A Priority This Semester	-.21 (63)	-.18 (54)	.00 (54)	-.03 (54)
Advanced Pascal Plans[a]	.00 (63)	-.21 (54)	-.16 (54)	-.26 (54)
Future Job using Programming[a]	.00 (63)	-.24 (54)	-.09 (54)	-.07 (54)

Table 9--continued

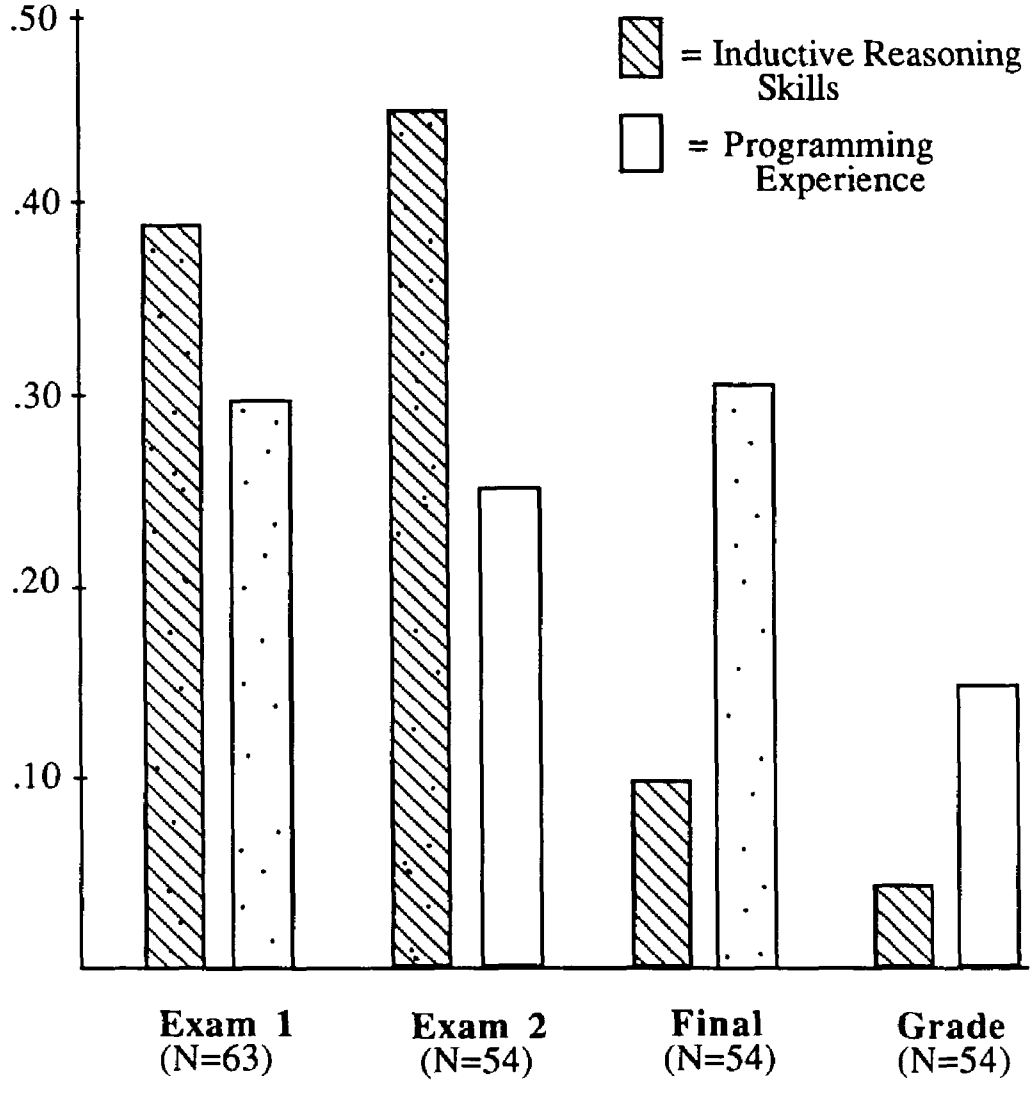
	EXAM 1	EXAM 2	FINAL	GRADE
Prior Word Processing[a]	.35 (63)	.18 (54)	.10 (54)	.09 (54)
Prior Top-down Design[a]	.19 (54)	.11 (47)	-.04 (47)	-.15 (47)
Age	.17 (58)	.07 (50)	.09 (50)	.10 (50)
Sex[b]	.18 (61)	.17 (52)	.05 (52)	.01 (52)

Note. Sample sizes are in parentheses.
[a] Coded No=0, Yes=1.
[b] Coded Female=0, Male=1.

good predictor of course grades. This finding is misleading, however, because 27 of the 75 students in Study 1 were first-semester freshmen. However, for students in this study who had college experience, GPA was a good predictor of introductory computer programming achievement. It is also interesting to note that college grade-point average was the only variable that correlated with course grade at the .05 or below significance level ($p<.004$, $n = 45$).

Other Variables and Exam 1. Correlations between Exam 1 and the remaining variables in this study are also reported in Table 9. Prior word processing experience was found to be correlated with Exam 1 ($r = .35$, $p < .005$). However, since it was measured imprecisely in this study and since it was not reported as strong correlate in previous research, it was not included in the subsequent regression analysis. Further

Figure 3. A Comparison of the Correlations Between the Dependent Variables and Inductive Reasoning and Computer Programming Experience



Note: Dots indicate at least the $p<.05$ level of significance.

research is needed to ascertain the relative importance of prior experience with a word processor for programming course achievement. It may be that familiarity with the computer system used in the course is of particular importance and not word processing per se.

The 65 students who responded affirmatively to the question, "Is English your native language?", tended to do better on Exam 1 than the ten students for whom English is not their native tongue. This variable was also not included in the subsequent regression analysis for the same reasons as mentioned above for the word processing variable.

Small or low correlations with Exam 1 were found for high school grade-point average, number of high school math courses, age, sex, and the motivation-related variables (Pascal Course A Priority This Semester, Plans To Take Advanced Pascal and Future Job In Programming). It is often the case that correlations between variables decline as the time interval between them increases. This may help explain the small correlations between Exam 1 and high school grade-point average, number of high school math courses, and motivational variables. Further, all of these variables are measured imprecisely here.

In addition, prior experience with "top-down design", as reflected in response to Question 16 on the Learner Characteristics Questionnaire was not an important Exam 1 correlate. This later finding is not surprising in that the novices were just beginning to learn about top-down design. According to anecdotal information from several intermediate level programmers in the study, the programming problems presented early in the course were too simple to require layers of decomposition and, hence, they did not spontaneously use a structured or top-down approach to the problems, even

though they were encouraged to do so by their instructor.

Based on an examination of the correlation matrix and previous research, three independent variables were chosen for the regression analyses: Progressive Matrices (inductive reasoning), programming experience group, and ACT Math.

Regression Analyses

For Study 1, a model was created predicting exam and course grade achievement. The model contained two components: inductive reasoning and computer-related knowledge base. Computer programming experience and ACT Math comprised the knowledge base component. Figure 4 is a visual representation of the model.

The Statistical Analysis System (SAS) was used for the multiple regression analysis. The regression procedure chosen was General Linear Models (GLM) procedure. This program provides statistics on the predictive power of the model as well as the individual contributions of each independent variable to the model. More specifically, the GLM procedure gives the sum of squares (Type III) that would be obtained for each variable if it were entered last into the model. That is, the effect of each variable is evaluated after all other factors have been accounted for. For purposes of this study, only the regression analyses for Exam 1 and Course Grades are discussed. The summaries for the regression analyses for Exam 2 and Final Exam are presented in Appendix B.

Regression Analysis for Exam 1. Scores for Progressive Matrices (inductive reasoning), Computer Programming Experience, and ACT Math were regressed on Exam 1. The results from the regression analysis indicated that 31% of the variation in

Figure 4. Model for Predicting Exam and Course Grade Success

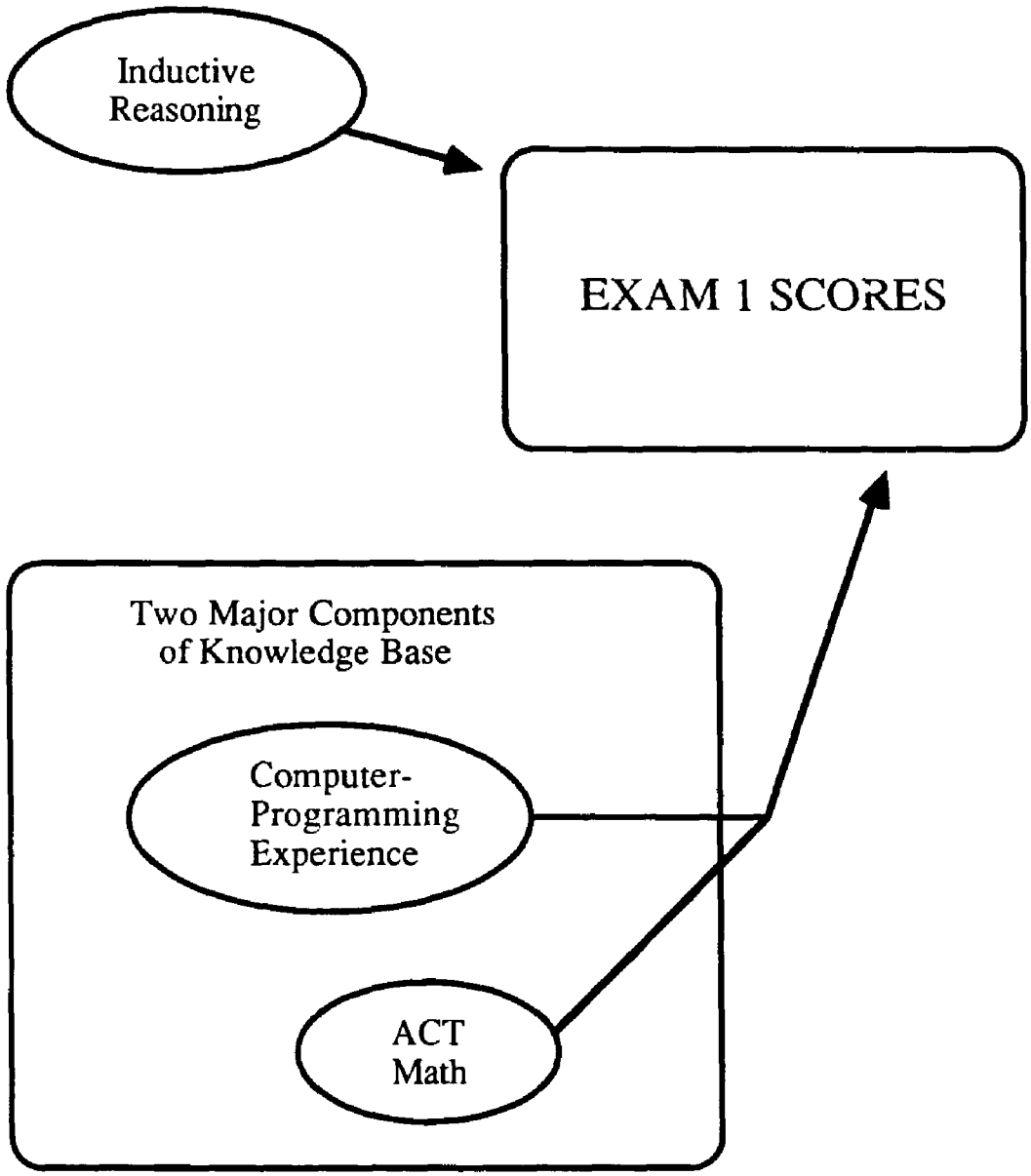
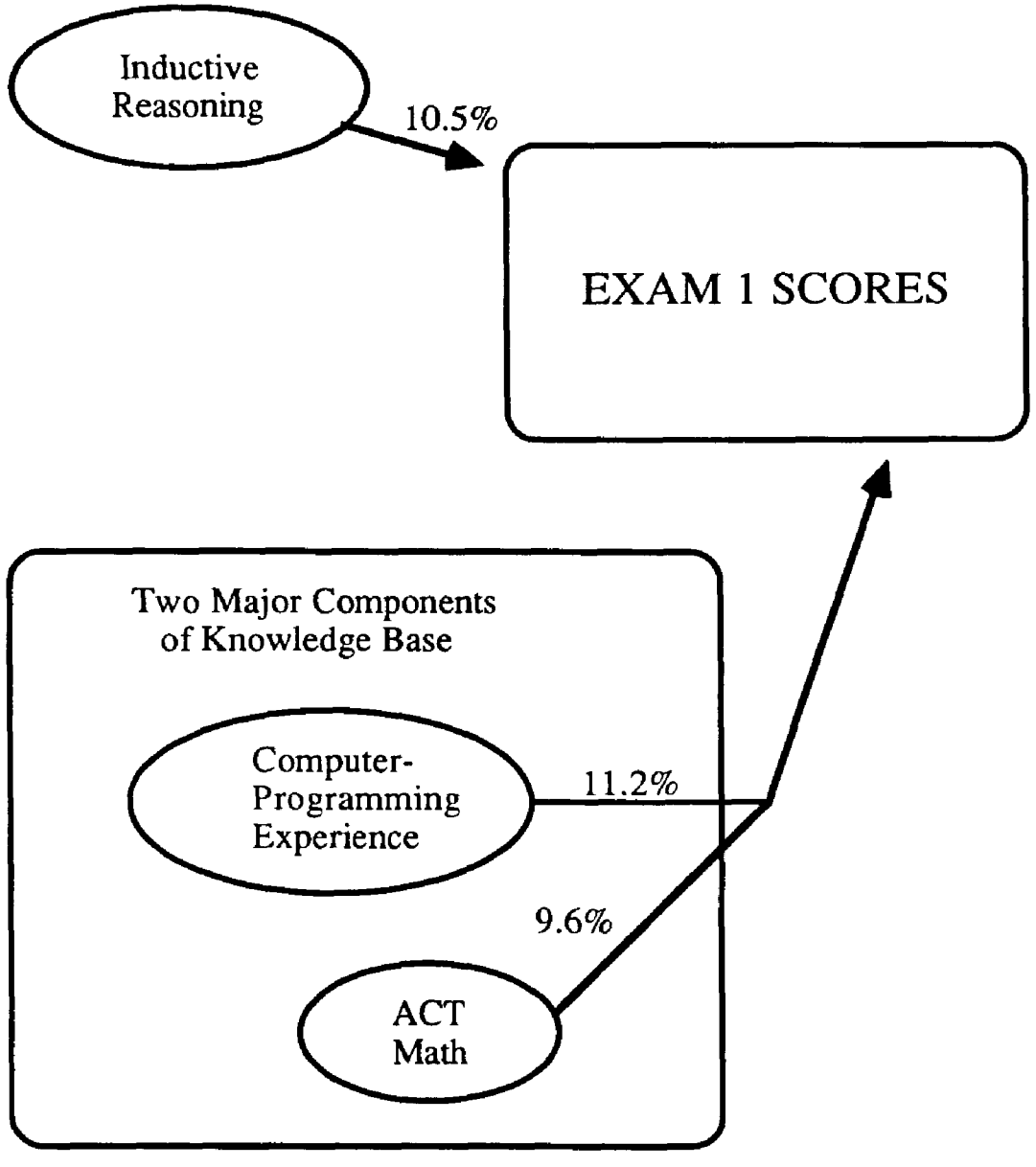


Figure 5. Contributions of Each Independent Variable in the Model for Exam 1 Achievement



Exam 1 could be accounted for by the linear combination of the three independent variables. Of the 63 people who took Exam 1, eleven did not grant permission to use their ACT Math scores. Consequently, the number of subjects for the Exam 1 regression study was reduced to 52.

All three variables in the model made significant independent contributions to the prediction of Exam 1 achievement. Figure 5 shows the contributions of each variable to the model. The percent of total variation contributed by each variable was similar, ranging from .31 to .36. Table 10 gives more statistical information about the contributions of each variable in the model for predicting Exam 1 scores.

Table 10. Contributions of Each Variable in the Model for Predicting Exam 1 Scores

Variable	ß	Type III SS	F Value	PR>F
Inductive Reasoning	2.21	3078.76	5.05	.03
Programming Experience	8.25	3244.68	5.32	.03
ACT Math	1.91	2807.43	4.60	.04
Intercept	-27.25			

Note. N = 52.

Regression Analysis for Course Grades. Although Exam 1 is the focus for this dissertation, a regression analyis was performed for course grades as well. The results were not surprising considering the low correlations between each independent variable and Course Grade (Table 9). It is important to note that Course Grades include Exam 1, Exam 2, Final Exam, nine quiz grades, and ten homework grades.

Table 11 gives a summary of the regression analysis for Course Grades.

Table 11. Contributions of Each Variable in the Model for Predicting Course Grades

Variable	ß	Type III SS	F Value	PR>F
Inductive Reasoning	-.02	.239	.16	.6929
Programming Experience	.08	.286	.19	.6657
ACT Math	.07	3.5	2.31	.1358
Intercept	.95			

Note. N = 46.

Nine of the 63 people who took Exam 1 dropped the course following Exam 1. Of the 54 subjects receiving course grades, eight people did not grant permission to use their ACT Math scores in this study. Therefore, the number participating in the regression study for Course Grades was reduced to 46.

Only 6% of the variance in Course Grade can be said to be attributed to the model. None of the independent variables made significant contributions to the model. Table 11 gives details on the contributions of each variable in the model for predicting Course Grades.

The summaries for the regression analyses for Exam 2 and Final Exam are presented in Appendix B.

Summary

In summary, the goal of the first study was to determine whether individual differences in inductive reasoning abilities, prior computer programming experience,

and ACT Math make independent contributions to the prediction of examination scores and Course Grades in an introductory Pascal course. Results indicated that all three variables made significant independent contributions to the prediction of Exam 1 achievement. The model as a whole accounted for 31% of the variance in Exam 1 scores. In contrast, none of the three variables made independent contibutions to the prediction of Course Grades, and together only accounted for 6% of the variance in Course Grades.

Problem representation is explored in Study 2. More specifically, Study 2 examines in greater detail the particular contributions of prior programming experience and inductive reasoning abilities to students' categorization of programming problems prior to Exam 1.

CHAPTER IV

STUDY 2

Study 1 indicated that prior computer programming and inductive reasoning skills made independent contributions to Exam 1 achievement. In Study 2, the nature of this contribution is examined in the context of how students represent problems in computer programming. As previously discussed, a problem representation is constructed using the knowledge for a particular type of problem (Chi, Feltovich, & Glaser, 1981). Figure 1 (see Chapter 1) illustrates this process. Chi and her colleagues hypothesized that differences between novices and experts may be related to "poorly formed, qualitatively different, or nonexistent categories in the novice representation" (p.122). Other researchers (Holland, Holyoak, Nisbett, & Thagard, 1986) assert that induction plays a vital role in generating a problem representation. This study examines the roles of both prior computer programming knowledge and inductive reasoning skills in generating a problem representation.

Overview of Research Design

The designs of Studies 2-4 closely resemble the Chi, Feltovich, and Glaser (1981) research designs. The main differences are (a) computer science, not physics, is the domain of interest, (b) more subjects are used in this study, and (c) students do not just state their basic approaches but actually solve programming problems. The methodologies are also consistent with suggestions from Linn (1985) and Shneiderman & Mayer (1979) on measuring specific cognitive accomplishments from

learning programming. In Studies 2-4, the knowledge dimension (novice versus experienced computer programmers) is crossed with inductive reasoning dimension (Average Gf versus High Gf).

Research Questions: Study 2

1. Do the novice and experienced programmers, who are all learning a new language, have different organizational categories for their new knowledge?

2. Do individual differences in inductive reasoning/fluid ability (Gf) influence categorization?

Method

Subjects

The 82 subjects from Study 1 participated in this study. Since Study 2 did not involve any programming, the seven subjects with some prior Pascal experience who were excluded from Study 1 were included in this study. They were paid $3.50 per hour. The data were collected during the first few weeks of the same introductory computer language course in Pascal described in Study 1. The subjects ranged from novices who had no prior programming experience to advanced intermediate programmers who had both academic and work experience in one or two programming languages. Although all 84 subjects completed the categorization task, the middle range of subjects, the advanced novice and intermediate groups, were removed from the sample because of time and financial constraints. For a more detailed description of the experience groups, see Chapter 3. Data for the advanced novice and intermediate groups are available upon request from the author. More

specifically, the subjects for both the quantitative and qualitative analyses were 17 novice programmers (8 with high inductive reasoning abilities, 9 with average inductive reasoning abilities) and 16 of the most experienced programmers (11 with high inductive reasoning, 5 with average inductive reasoning). The experienced programmers will be called "intermediate level" in this study. Subjects who scored 25 or below on the Raven's Advanced Progressive Matrices were labeled average inductive reasoners, whereas subjects who scored 26 or above were labeled high inductive reasoners. According to the norms manual for the 1962 version of the Raven's Advanced Progressive Matrices, the mean score for university students is 21 (S.D.=4) (Raven, Court, & Raven, 1977). Thus, the subjects labeled high inductive reasoning ability scored more than one standard deviation above the mean for university students on this test.

Materials

A set of 30 typical introductory computer programming problems was constructed in which surface features (e.g., letters, numbers, money) were roughly crossed with secondary features involved in solution strategies (e.g., sorts, single or multiple loops, state machine). For example, the following problem was categorized as a sort problem:

> A certain Swiss Bank has ten customers, each with a unique account number. Write a program to read in ten pairs of account numbers and balances. Then your program should print them out so that an account with a larger balance is always listed before any of the accounts with smaller balances.

This task, inspired by the sorting task in the Chi, Feltovich and Glaser (1981) study, was created by an individual with a B.S. in Computer Science and the author of this

study. See Appendix C for the 30 problems organized by both surface and secondary features.

Procedure

Subjects were asked to sort the computer-programming problems into five groups based on similarities of solution. They were given twenty-five minutes for the sorting aspect of the task and ten minutes to complete the answer sheet. They were then asked to label each problem group and also to give a reason for choosing the label. Appendix D contains the answer sheet, including detailed instructions.

Results and Discussion

The quantitative analyses of the sorting task will be presented first, including both multidimensional scaling and cluster analyses. This will be followed by a qualitative analysis of the types of category labels generated by novice and experienced programmers of average and high inductive reasoning abilities. Chapter four concludes with an analyis of the number of natural language and computer language category labels generated by each group.

Quantitative Analysis of Sorting Task Responses

Hypotheses.

1. Novices will generate categories based on superficial features in the problem statements.

2. Advanced intermediates will produce categories based on problem types (i.e., secondary features in the problem statements).

3. No specific hypotheses were made concerning inductive reasoning abilities.

A 30 x 30 matrix was created for each of the four groups by counting the frequency with which each of the 30 problems was paired with every other problem. The resulting four proximity matrices were used for both a multidimensional scaling analysis and a cluster analysis. These analyses were performed to determine the degree to which subjects in each group agreed that certain problems belonged to the same group.

The multidimensional scaling procedure called ALSCAL (SAS Institute, Inc., 1986) was used to create spatial representations of the proximity data in two through five dimensions. The untie option for ordinal data was chosen. Ties existing in the data were untied in such a way that goodness-of-fit was optimized. The similar option was used indicating that small numbers mean little similarity, whereas large numbers mean great similarity.

The spatial solutions for dimensions one through five were examined as well as the stress values for measuring goodness of fit. Since the two-dimensional solutions for each group were more interpretable and had relatively low stress values, they were used for this exploratory study. Table 12 gives stress values and the r-squares for all five solutions for each group.

A hierarchical cluster analysis (SAS Institute, Inc., 1985) was performed using a distance matrix in which larger numbers indicated greater dissimilarity between two problem statements. The original proximity matrices were therefore modified to accomodate this feature. In the SAS (1985) cluster procedure, each observation begins in a cluster by itself. Then the two closest clusters are merged to form a new cluster replacing the two old clusters. This process is repeated until only one cluster is left.

Both the average method and Ward's minimum variance method were used. Ward's method was chosen because it was more interpretable when used in conjunction with the multidimensional scaling plots. For the Ward's method, the distance between two clusters is the sum of squares between the two clusters summed over all variables. Clusters with few observations tend to be joined with the Ward's method. Also this method is biased toward producing clusters with roughly the same number of observations. In the context of this study, an observation or variable refers to paired problem statements. (See Appendix C for the list of problem statements used in this task.)

Table 12. Stress Values and Squared Correlations for the Solution Dimensions

		Novice		Intermediate	
Number of Dimensions		Inductive Reasoning Average	High	Inductive Reasoning Average	High
5	Stress	.04	.03	.01	.05
	RSQ	.98	.99	1.00	.97
4	Stress	.06	.05	.03	.07
	RSQ	.97	.98	1.00	.95
3	Stress	.07	.06	.05	.10
	RSQ	.97	.98	.98	.93
2	Stress	.11	.12	.10	.16
	RSQ	.94	.93	.96	.87
1	Stress	.27	.19	.18	.29
	RSQ	.78	.90	.92	.75

For the purposes of this exploratory study, the numerical details from the cluster analysis are less important than clearly identified clusters or groups as seen in the cluster tree plots. These clearly identified clusters were superimposed on the scaling representation for each group. The lack of ideal fit between the multidimensional scaling plots and the cluster analysis capitalizes on chance and, therefore, adds an element of uncertainty in the analysis.

Categorization Behavior: Novices Average Gf. The results for the two-dimensional solution for the novices of average Gf are shown in Figure 6. All clusters appear to be categories chosen on the basis of primary or secondary surface features in the problem statements. The clusters appear to represent: (a) money problems (E, O, U, 5, K); (b) geometry problems (1, 7, Q, R, S); (c) math problems (4, 6, J, L, N); (d) uninterpretable--three problems with numeric input with one uncertain (C, F, P, T); (e) alphabet or output oriented problems (8, 9, A, B); and (f) character or word input and/or character or word output problems (2, 3, D, G, H, I, M). Categorization according to surface features is typical novice behavior as reported by Chi, Feltovich, and Glaser (1981) and other researchers in a wide range of fields.

Categorization Behavior: Novices High Gf. The results for the two-dimensional solution for the novices of high inductive reasoning appear in Figure 7. From left to right, the clusters appear to represent: (a) read in characters or text, perform transformation and print out (2, G, M); (b) alphabet and/or outcome oriented problems (8, 9, A, B); (c) single and multiple loops (3, D, H); (d) outcome oriented or sort problems (K, R); (e) decision oriented problems (C, I, P, T); (f) money

Figure 6. Two-dimensional Solution for Novices of Average Inductive Reasoning

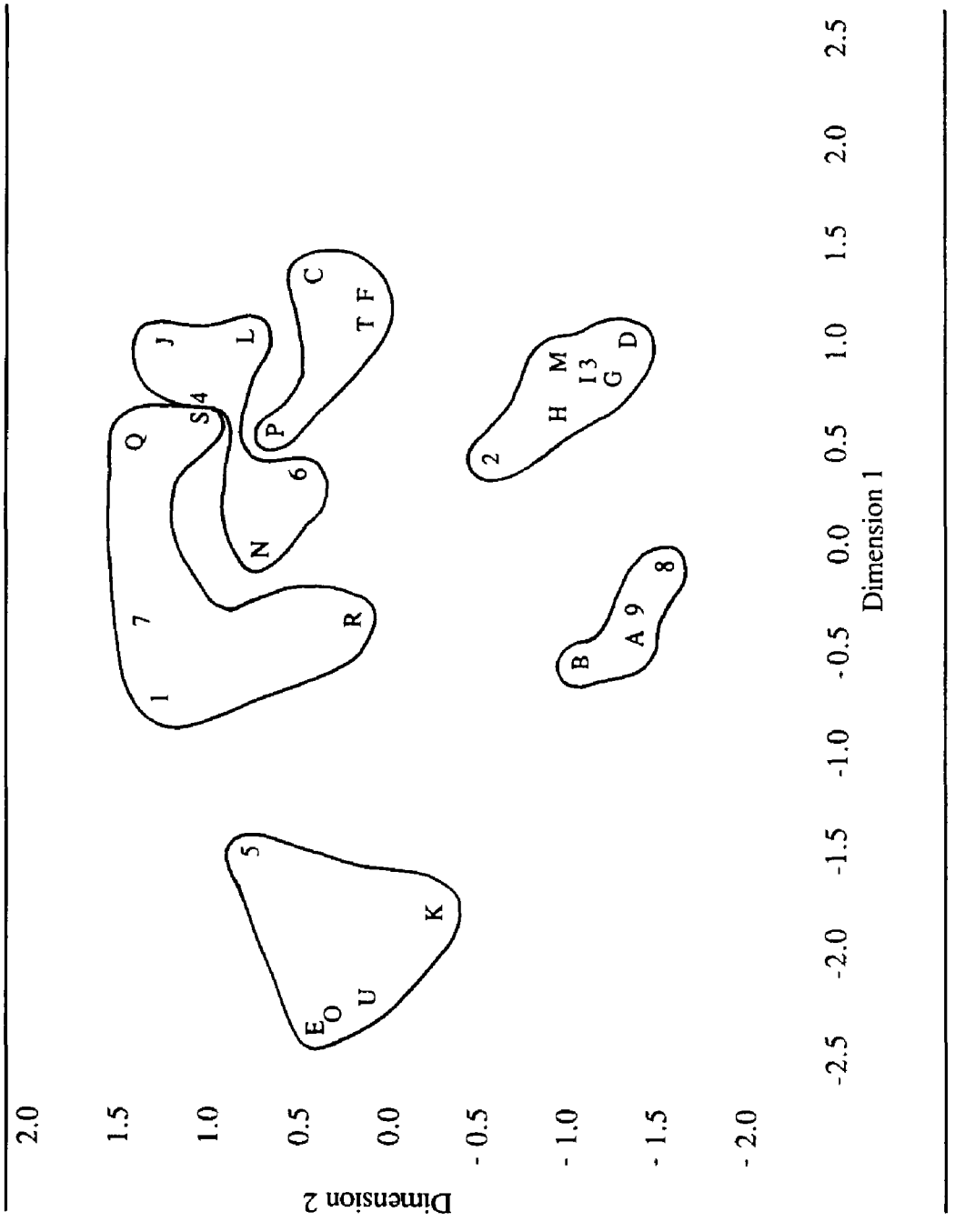
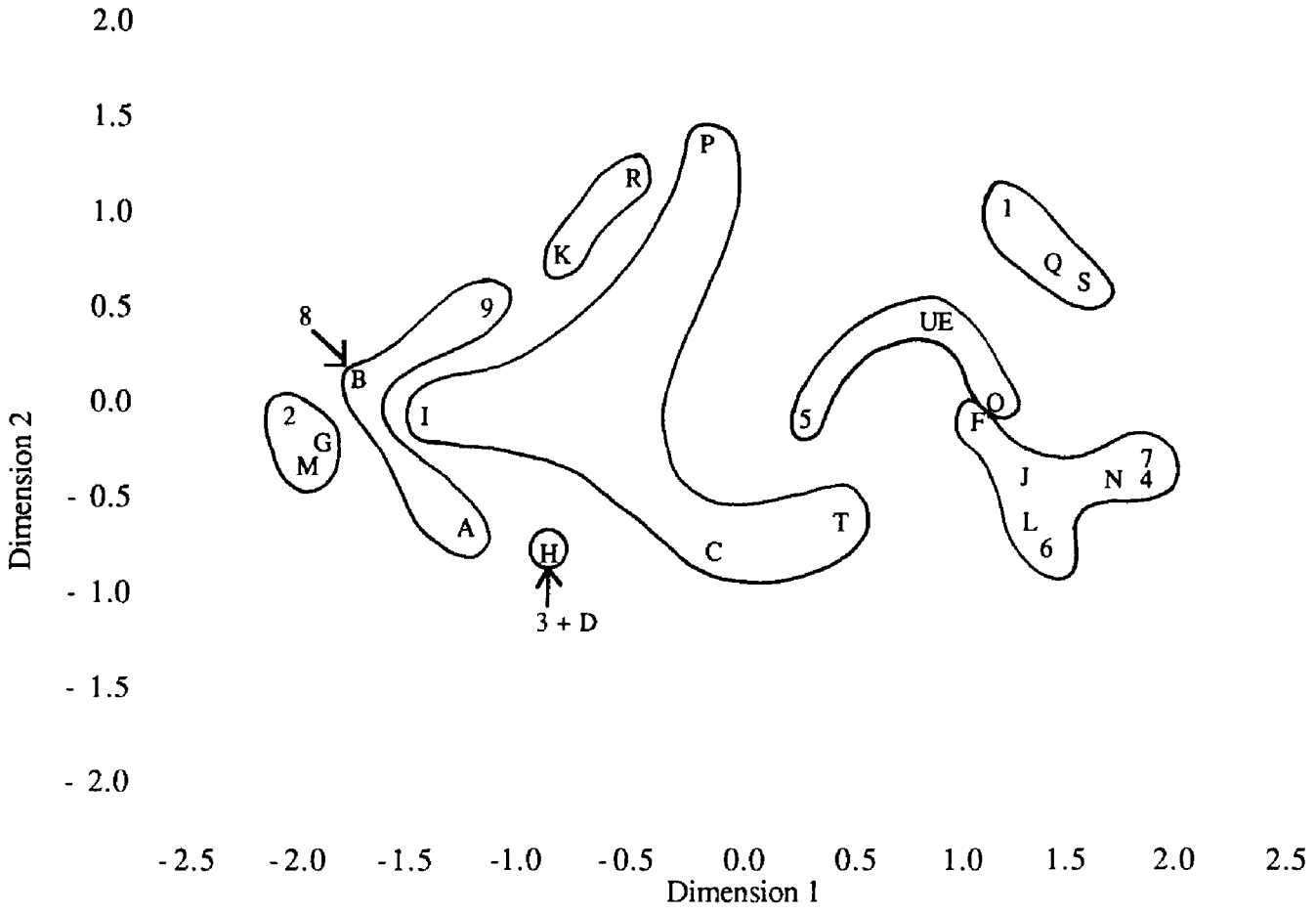


Figure 7. Two-dimensional Solution for Novices of High Inductive Reasoning



problems (5, E, O, U); (g) geometry problems (1, Q, S); and (h) math problems (4, 6, 7, F, J, L, N).

Like the novices of average Gf, these subjects used surface features for identifying some of their categories (geometry, money, and math). However, they transcended the boundaries of the surface features in several instances. For example, the novices of average Gf identified problem R as a geomety problem and problem K as a money problem. Problem K is the Swiss bank problem previously mentioned as a problem with money-related surface features and a sort solution strategy. Problem R is also a sort problem. The novices of high Gf paired problems R and K. In addition to using sorts as a solution strategy, problems R and K require precise output specifications. This was a design flaw in the task so it is difficult to ascertain if the novice subjects had read about sorts and recognized the conditions for using them or if they categorized the two problems by output specifications. In either case, the categorization behavior of the novices of high Gf appears to be on a deeper level.

Another case of differences between novices of average Gf and novices of high Gf can be seen by examining categorization behavior for problems 3, D, and H. The novices of average Gf clustered these problems with several other problems that had character input or character output, whereas novices of high Gf created a separate category for 3, D, and H. This category contains problems which use either loops or multiple loops for solving the problem.

Lastly, two dimensions appear to capture most of the data for the novice group with average inductive reasoning skills. The appropriate label for dimension one is not obvious. However, dimension two appears to represent categorization by numeric problem features. Problems in the upper half of the plot were mainly sorted into

categories based on numeric problem features (money, geometry, math, three numeric input problems plus one uncertain) whereas problems in the lower half of the plot were sorted based on letter features (alphabet, character, or word). In contrast, one dimension seems to fit better for the novices with high inductive reasoning skills. This dimension appears to represent categorization by problem features. The clusters on the right half of the plot are categories with common surface features (i.e., money, geometry, and math). The clusters on the left side of the plot are categories which reflect more inferred knowledge or at least more processing of the problem statement beyond superficial features (i.e., decision oriented, single or multiple loops, outcome oriented, and read in characters or text, perform transformation and print out). (The stress level, however, is better for two dimensions. Table 12 gives the stress levels for both dimensions.)

Researchers such as deJong and Ferguson-Hessler (1986) have reported that good novice problem solvers have their knowledge arranged around problem types (deep features) in the problem statement to a greater degree than poor novice problem solvers. These analyses lend some support to this hypothesis, although the distinction appears not to be as clear-cut as the deJong and Ferguson-Hessler (1986) findings. Novice problem solvers who were here classified as high Gf also classified problems primarily by the primary or secondary features in the problem statements. The fact that they organized some of their knowledge around problem types is striking considering they only had a few weeks of programming experience. In contrast, the subjects in the study by deJong and Ferguson-Hessler had a semester of college-level physics prior to participating in the problem-solving task in their study.

Categorization Behavior: Intermediate Level Programmers Average Gf. The results for the two-dimensional solution for the experienced programmers of average Gf are shown in Figure 8. From left to right the clusters appear to represent: (a) Rearrange or transform characters or Words (2, 3, D, G, I); (b) Uninterpretable--five sorts with two uncertain (8, 9, A, B, H, K, M); (c) Interest Problems (E, U); (d) Math Problems (4, F, J, L, N, O, T); (e) Geometry (1, R, 7, Q, S); and (f) Uninterpretable--two decision oriented of boolean data type with two uncertain (5, 6, C, P). Overall, the results for the intermediates of average Gf were more difficult to interpret than the results of the other three groups.

Categorization Behavior: Intermediate Level Programmers High Gf. The results for the two-dimensional solution for the intermediates of high Gf are shown in Figure 9. From left to right, the clusters appear to represent: (a) Rearranging or transforming characters or words (2, 3, D, G, H, I, M); (b) Sorts (Large Algorithm) (8, 9, A, B, K, R); (c) Money Related (5, E, O, U); (d) Math Calculations (4, 6, F, J, L, N, T); (e) Geometry (1, 7, Q, S); and (f) Decision oriented of boolean data type (C, P).

It is surprising to see how many superficial level categories are created by both computer programming experience groups. There is, however, some additional discrimination. For example, the intermediates of average Gf have a money-related category with only two members. Both problems are interest problems. Perhaps even experts would categorize problems together that require more application. Further research is needed. To create a sort algorithm category, the intermediate programmers of high Gf probed beneath the surface level of geometry, money, letters, etc., to a common solution strategy for six problems. Categorization by solution plan features

Figure 8. Two-dimensional Solution for Intermediates of Average Inductive Reasoning

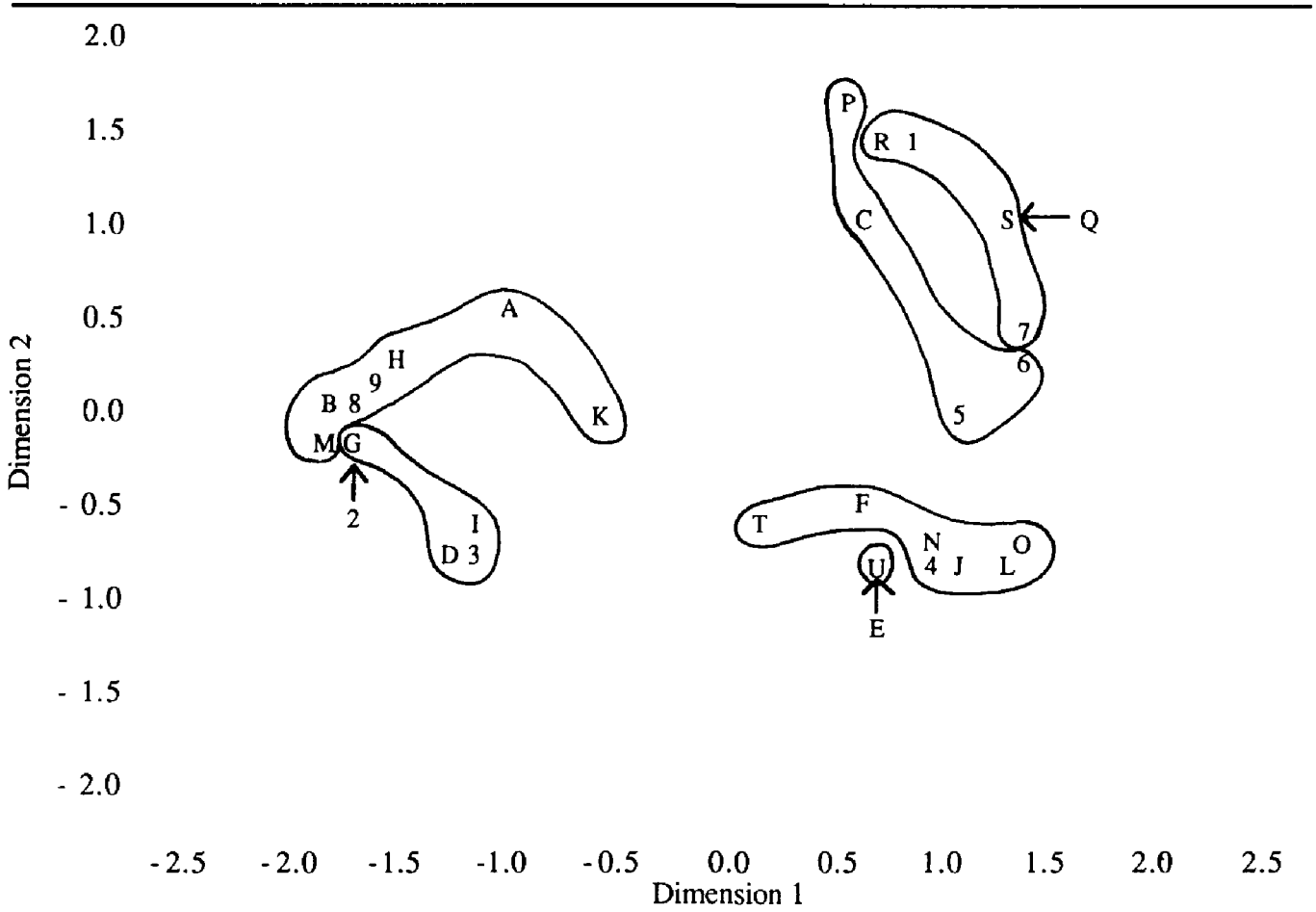
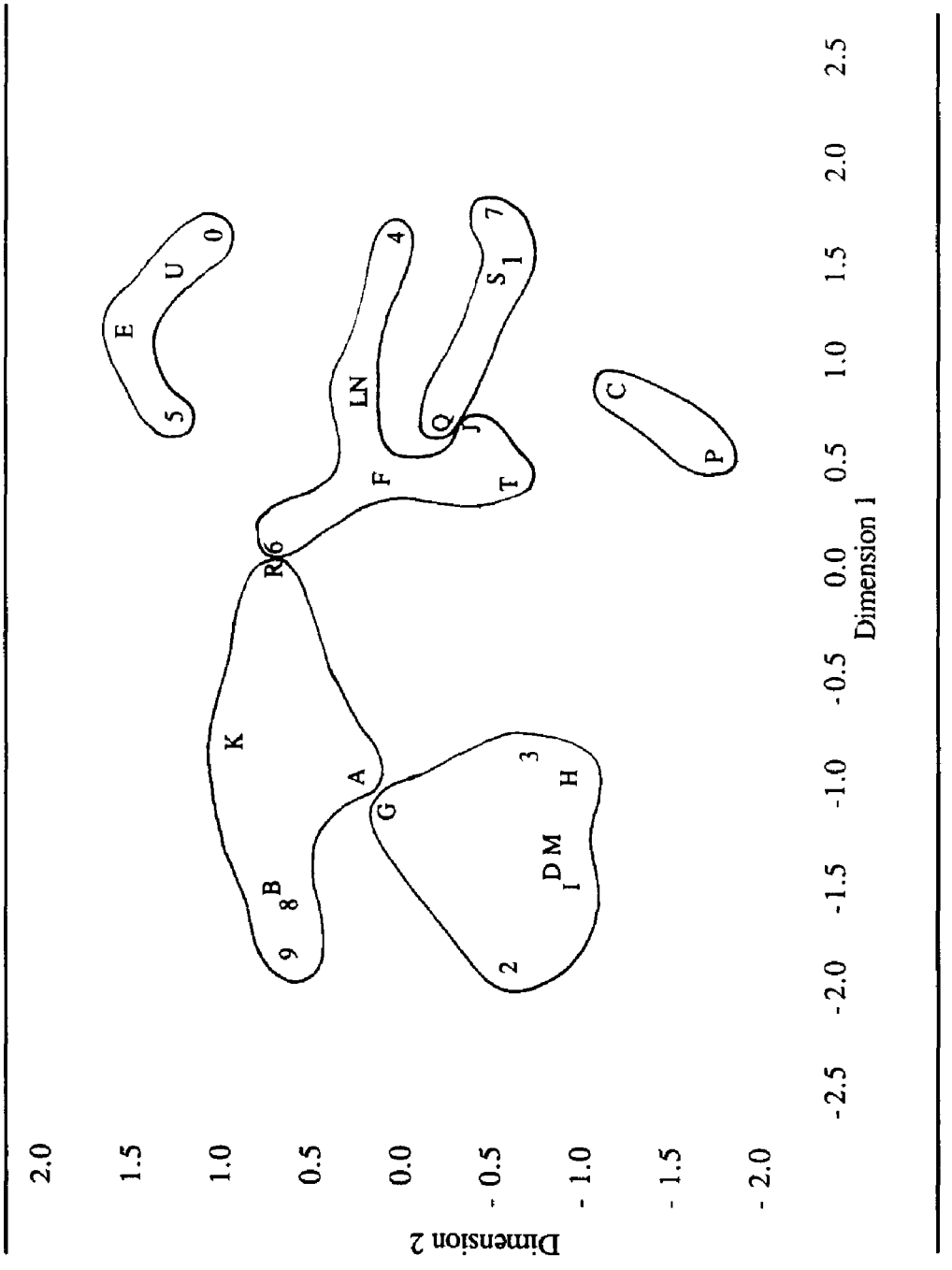


Figure 9. Two-dimensional Solution for Intermediates of High Inductive Reasoning



is typical behavior for experts in a variety of fields.

It is interesting to note that none of the groups created "reduction" or "state machine" categories. These categories were built into the underlying structure of the task. The reduction and state machine categories were expert-generated categories discovered during the task development phase prior to conducting this study. All other categories in the underlying model (Money, Geometry, Numbers, Letters/Words, etc.) were represented in the data. For a review of the underlying structure of the task, refer to Appendix C. Next, the qualitative analysis of the same sorting task will be presented.

Qualitative Analysis of Categorization Task Responses

Hypotheses.

1. Novices will generate more natural language categories than intermediate level programmers, with novices of average Gf producing the greatest number of natural language categories.

2. Intermediate level programmers will produce more computer language categories than novices, with intermediate subjects of high Gf generating the greatest number of computer categories.

For the quantitative analysis presented, the raw data were the number of times each problem was associated with the other problems. For the qualitative analysis reported in this section, the raw data were the five labels or names and the brief description given by each subject for all chosen categories. For example, problem numbers 76, 96, and 108 were put into a category labeled "interest." The subject indicated that "each of these requires some sort of comparison using interest."

Once again, the subjects for this analysis were: 17 novices (8 of high inductive reasoning, 9 of average analogical reasoning), and 16 intermediates (11 of high inductive reasoning, 5 of average inductive reasoning). (Data for all groups for this analysis, however, appears in Appendix E, Table 30.)

An independent rater who was an experienced programmer scored each category label for all subjects. The rater was unaware of data on the personal profiles of each subject. The rater consolidated all the categories of problem statements generated by the subjects into twenty-one categories presented in Table 13. These categories were in turn collapsed into the four superordinate categories shown in Table 13. The four superordinate categories were: (1) Natural language--problem features, (2) Natural language--problem process, (3) Computer language--problem features, and (4) Computer language--problem process. Table 13 shows how each of the twenty categories for problem labels were assigned to these superordinate categories. The first natural language category included problems grouped by labels which emphasized surface features such as money, geometry, letters, numbers, categories referring to editing or organizing, and input oriented statements. The second natural-language category, problem process, included labels which seem to require more thinking about how problems might be solved yet do not explicitly mention programming-related constructs. The categories include simple math, complex math, outcome oriented and decision makers.

The program knowledge categories also were divided into problem features and problem process. The problem features categories were data types (except Boolean) and Boolean. The problem process categories included operations with data types, output oriented, data structures, data structures plus manipulation, loops, nested

loops, small algorithms, large algorithms, outline approach, procedures, and complicated programs.

A no credit category was included (a) for irrelevant responses, (b) for two categories with same name and numbers, and (c) for associations to associations.

Table 13. Rater Consolidated Categories

Natural Language	**Computer Programming**
Problem Features	**Problem Process**
1. Surface Features	10. Operation with Data Types
2. Organizing/Editing	11. Output Oriented
3. Input Oriented	12. Data Structures
	13. Data Structures Plus Manipulation
Problem Process	14. Loop
4. Simple Math	15. Nested Loops
5. Complex Math	16. Small Algorithm
6. Outcome Oriented	17. Large Algorithm
7. Decision Makers	18. Outline Approach
	19. Procedures
Computer Programming	20. Complication Programs
Problem Features	**No Credit**
8. Data Types (Except Boolean)	
9. Boolean	

Number of Natural Language and Computer Language Categories Generated

For this analysis, then, the natural language category includes both the "features" and "process" categories. Similarly, the computer language category includes both the "features" and "process" programming categories. To test the hypothesis that no

differences exist in the categorization behavior of the four groups, a chi-square test was performed on the number of natural language and computer language categories produced by each group. The hypothesis was rejected ($x^2 = 19.01$, significant at .001 level). The groups are exhibiting significantly different categorizing behavior. Is there a pattern to these differences?

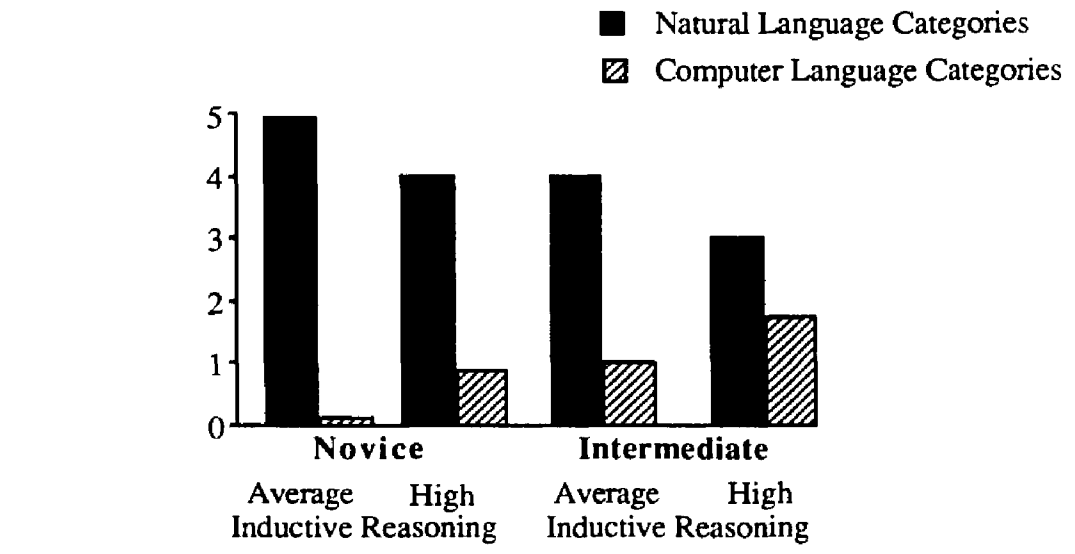
The first analysis sought to determine whether the four subject groups differed in the relative frequency with which they used natural language problem labels versus computer programming problem labels.

Figure 10 shows the mean number of natural language and computer language categories generated by each group. As experience and inductive reasoning skills increase, the number of natural language concepts decrease and the number of computer concepts increase.

The reader will recall that all subjects were asked to generate five categories for the thirty problems. As predicted, the novices of average Gf produced the greatest number of natural language categories. An average of 4.89 out of five were natural language categories. The novices of high Gf produced, on average, four natural language categories out of five. Likewise, intermediates of average Gf generated four out of five, while intermediates of high Gf generated 3.27 out of five natural language categories. The most striking result is the small number of computer language categories generated by the groups with computer experience. Although the intermediates generated more computer language categories than the novices, it was surprising to see how few they produced. Average Gf experienced programmers generated only one out of five, and the high Gf experienced programmers generated only 1.73 out of five.

Perhaps the time limit of 25 minutes was too short. An expert who completed the same sorting task indicated that he probably would have produced more refined sorting categories if he had more time to complete the task. This issue will be further explored in the final discussion in Chapter 7.

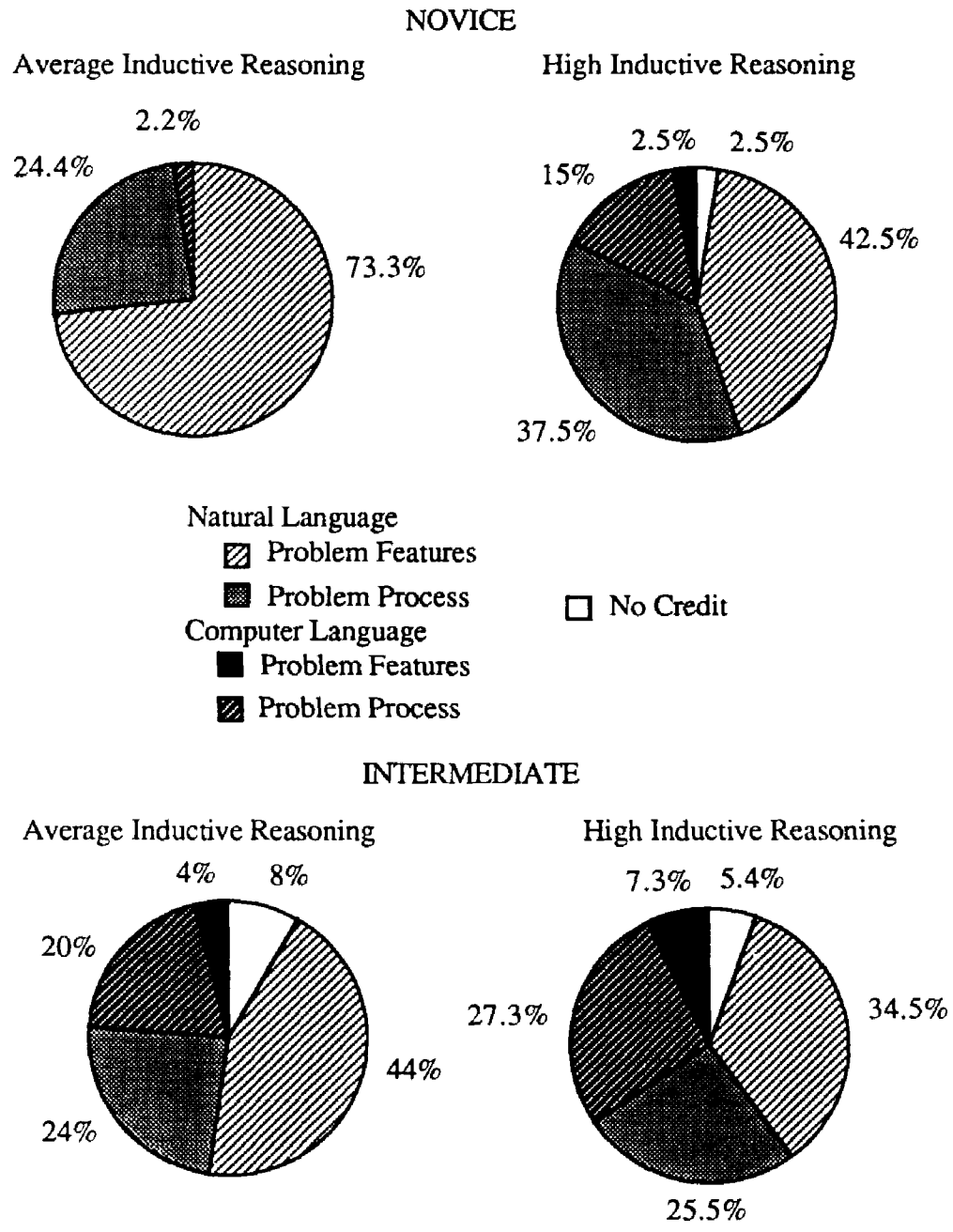
Figure 10. Mean Number of Natural Language and Computer Language Categories Generated by Novice and Intermediate Level Programmers of Average and High Inductive Reasoning (Gf)



Types of Category Responses

Additional differences between the four subject groups emerged when problem categories were further subdivided. For these analyses, the natural language and computer language categories were decomposed once again into two subgroups each (see Table 13). The percent of problem levels which were assigned to each of these

Figure 11. Types of Category Responses Made by Novice and Intermediate Level Programmers of Average and High Inductive Reasoning (Gf)



four categories (plus a no-credit category) is shown in Figure 11, separately for each of the four ability by experience groups.

The percent of problem labels assigned to the surface features category by the rater was much higher for the average Gf novices than for the other three groups. High Gf novices also relied primarily on natural language categories. However, in contrast with the intermediate Gf novices, high Gf novices used more problem process labels. This same trend is mirrored in the data for the students with some programming experience: low Gf intermediates used considerably fewer programming labels that focused on programming processes than did the high Gf intermediates. Thus, if there is a generalization here, it would appear that the experience dimension predicts the extent to which subjects use programming concepts to categorize problems whereas the inductive reasoning dimension predicts the extent to which they will focus on how problems are solved.

Final Discussion

Categorization and Learner Characteristics

It is important to note that the sorting task was administered a few weeks prior to the first examination in the introductory programming course in Pascal. The novice and experienced programmers who were all learning a new programming language were able to categorize programming problems in a meaningful way. What organizational framework did they use to categorize these problems? Do individual differences in inductive reasoning skills influence categorization?

The results from the multidimensional scaling and cluster analyses suggest that categorization behavior is influenced by both prior computer programming experience

and level of inductive reasoning skills. As programming experience and inductive reasoning skills increase, categories are based less on superficial features in the problem statements and more on underlying solution features or strategies.

More specifically, novices of average and high inductive reasoning skills appear to categorize problems according to primary or secondary surface features in the problem statements. In addition to surface characteristics, evidence suggests that novices of high inductive reasoning skills organize some of their knowledge around problem types (e.g., loops and decision-oriented categories).

The results of this study suggest that intermediate level programmers use both surface features and inferred constructs not found in the problem statements to categorize the programming problems. For example, like the novice groups, members of both intermediate groups put the geometry problems together without carefully considering whether the problems required similar solution algorithms. In addition, however, evidence suggests that a deeper categorization framework based more on problem types was also used.

For the experienced programmers of average inductive reasoning skills, it was difficult to determine the sorting categories from the multidimensional scaling and cluster anlayses. However, analysis of the labels given to the categories indicated that the experienced programmers of average Gf used several computer-related categories for organizing the problem statements (data types, operations with data types, output oriented, loops, and algorithms). This qualitative analysis revealed that these intermediate level programmers of average inductive reasoning skills generated fewer categories based on the programming procedure than did the intermediate level programmers of high analogical reasoning skills.

The multidimensional scaling and cluster analyses suggest that some experienced programmers of high inductive reasoning skills categorize problems according to a solution strategy--a sort algorithm. The more qualitative analysis of labels further supports this interpretation.

Are differences between the groups confounded with Math achievement? To answer this question, T-tests for independent means were performed for differences between group means on ACT Math achievement. No significant differences were found. The mean ACT scores for each group are shown in Table 14. (ACT Math scores were not available for seven subjects.) From the available data, it does not appear that differences between the groups are confounded with math achievement.

Table 14. Mean ACT Math Scores and Standard Deviations (S.D.) for Novice and Intermediate Programmers of Average and High Inductive Reasoning (Gf)

Group	Mean	S.D.	N
Novice Average Gf	25.29	4.72	7
Novice High Gf	28.14	2.61	7
Intermediate Average Gf	25.25	3.30	4
Intermediate High Gf	26.38	4.31	8

Categorization and Problem Representation

What is the relation between categorization responses and an individual's representation of problems? Chi, Feltovich and Glaser (1981) suggest two plausible

interpretations, one theory proposed by McDermott and Larkin (1978) and an alternative interpretation proposed by themselves.

According to McDermott and Larkin, researchers in the area of physics problem solving, the process of representing a problem proceeds in the following way: (a) the problem statement is read; (b) a representation is formed; and (c) based on the representation, the problem is categorized. While solving a problem, the problem solver progresses through four stages of representations.

1. Stage one -- a literal representation of the problem is made. It contains relevant key words.

2. Stage two -- a "naive" representation is created. This representation contains the literal objects and their spatial relationships as stated in the problem statement. An individual with little or no prior knowledge in a field could create a stage two representation.

3. Stage three -- a "scientific" representation is formed containing the idealized objects and relevant concepts necessary to generate the equations for stage four of the problem representation process.

4. Stage four -- equations are produced resulting in an algebraic representation.

In the Study 2 sorting task described in this chapter, subjects were asked to sort computer programming problems into categories based on similarities of solution. An interpretation of the results of Study 2 from the McDermott-Larkin framework, then, is that the novices of average inductive reasoning skills appear to base their categorization mainly on the construction of literal representations, or what are here called natural language concepts. The experienced programmers' categorization is based on both naive (or natural language) representations and scientific (or programming)

representations. However, the McDermott-Larkin framework does not accomodate the finding that high Gf subjects in both experience groups appeared to generate problem representations that emphasized solution process.

In this respect, then, the Study 2 results are consistent with Holland, Holyoak, Nisbett and Thagard's (1986) assertion that induction plays a vital role in generating a problem representation. They argue that induction consists of generating and revising the units of the problem representation from which mental models are constructed. It is not surprising, then, that novice and experienced programmers of different levels of inductive reasoning skills generate somewhat different categories and, hence, problem representations. The key difference attributable to inductive reasoning appears to be the extent to which problems are represented by how they are solved rather than by the terms or concepts embedded in them.

Chi, Feltovich and Glaser (1981) postulate more interaction among stages of representation than McDermott and Larkin (1978). An interesting feature of this alternative hypothesis is that a problem representation is not fully constructed until after the initial categorization process. More specifically, after a preliminary analysis of problem features, a problem is tentatively categorized. This process can be accomplished by a set of rules that specify problem features and the corresponding categories to be cued. Next, available knowledge associated with the category is used to construct a plan for solving the problem. Chi and her colleagues suggest that the knowledge available for a problem type constrains and guides the final form of the problem representation. A "schema" (Rumelhart, 1981) for a particular problem type is composed of a category and its associated knowledge in the knowledge base. They argue that the quality of the problem representation is ultimately determined by the

contents of these problem schemata.

Study 2 suggests that, in general, as programming experience and inductive reasoning skills increase, categories and, hence, representations are based less on superficial features in the problem statement and more on underlying solution features or strategies. Because differences are found in categorization responses, the problem schemata of novice and experienced programmers of average and high inductive reasoning skills should contain different knowledge. Study 3 is designed to give a more direct look at the knowledge accessed by the category labels.

CHAPTER V

STUDY 3

The purpose of this study was to gain more insight into the categorization and problem representation process. To this end, it is useful to know what knowledge is associated with the category descriptions given in Study 2. For purposes of this research, it is assumed that the category descriptions provided by experienced and novice programmers represent labels used to access related units of knowledge, i.e., schemata. Study 3 was designed to access the knowledge associated with these schemata. "It is the content of these problem schemata . . .that ultimately determines the quality of the problem representation" (Chi, Feltovich & Glaser, 1981, p.135). Hence the purpose of Study 3 was to uncover what knowledge is contained in the schemata of experienced (intermediate and advanced intermediate) programmers and novices.

Research Questions

1. What knowledge is associated with the category labels?
2. Do individual differences in inductive reasoning skills and computer programming experience influence the knowledge contained in the schemata of the subjects?

Method

Subjects

Twelve subjects from Study 2 participated: three novices of high inductive

reasoning, three novices of average inductive reasoning, three experienced programmers of high inductive reasoning, and three experienced programmers of average inductive reasoning. Only students with extreme scores on the Raven's Advanced Progressive Matrices Task were invited to participate.

Subjects with scores of 25 and below were referred to as average inductive reasoners (Gf). Subjects with scores of 30 and above were called high inductive reasoners (Gf). (Note: In Study 2 the cut off score was 26 points for the high inductive reasoning group.)

The novice group was composed of novice programmers with either no prior programming experience or one semester of programming in high school. The experienced group was made up of intermediate and advanced programmers with programming experience ranging from two semesters of college level programming (two different languages) to six semesters of college level programming plus work experience. The initial plan was to include only the advanced experienced programmers. However, of the five possible candidates with average inductive reasoning scores, only one was interested in participating in Study 3. Consequently, in an effort to equate the two experience groups, only one advanced intermediate was included in each group. In summary, the subjects were: (a) three novices of average inductive reasoning skills, (b) three novices of high inductive reasoning skills, (c) two intermediate and one advanced programmer of average inductive reasoning skills, and (d) two intermediate and one advanced programmer of high inductive reasoning skills. The intermediate and advanced programmers are referred to as "intermediate" programmers throughout this study.

The subjects were paid $3.50 per hour.

Materials

Materials consisted of a set of category labels ranging from surface features of problems to secondary features such as algorithms used to solve problems. Category labels (see Table 15) were chosen from typical category responses generated by both novice and experienced programmers during the categorization task in Study 2. One additional experimenter-generated label was added--the "test" label.

Table 15. Free-Association Labels

* Array	* Listing
* Boolean	* Loop
* Characters	* Money Operations/ Financial Problems
* Complex Math Problems involving manipulation of input over several steps	* Multiple Loops
* Easy Math Problems	* Procedures
* Function	* Real
* Geometry	* Sort
* Integers	* Test

Procedure

Subjects were given 16 pieces of paper with one category label written on the first

line of each page. The labels were listed in a different random order for each subject. All 16 pages were placed face down. At the test administrator's signal, the first page was turned over. Subjects were told that they had four minutes to write everything they could think of when they read the particular category label, including how problems in this category might be solved. The time keeper informed the subjects when it was time to go on to the next category label.

Results and Discussion

The research question, "What knowledge is associated with the category labels?", was decomposed into the following sub-questions:

1. Are natural language concepts associated with the label(s)?
2. Are computer language concepts associated with the label(s)?
3. Is the conceptual knowledge associated with the labels different for novice and experienced programmers of average and high inductive reasoning skills?

The data from this elaboration task was rated by two teaching assistants from the computer programming course. One rater determined the boundaries between concepts for each label. Then both raters scored each concept for all subjects across all labels. When deciding whether a concept was a natural language concept or a computer concept, the raters agreed on all but 5 of the 970 concepts. Thus, the agreement between the two raters was nearly 100%.

The following example may help to clarify the distinction between a natural language and a computer concept. For the label "characters", one subject gave the following two responses: "the Three Stooges are these" and "they need single quote marks around them or they will create errors". The first response was rated as a

natural language concept, whereas the later response was rated as a computer concept.

Production of Natural Language Concepts and Computer Concepts

Hypotheses.

1. Intermediate programmers will generate more computer concepts than natural language concepts.

2. Intermediate programmers with high inductive reasoning skills will generate the highest percent of computer concepts.

3. Novices will produce more natural language concepts than computer concepts.

4. Novices with average inductive reasoning skills will generate the lowest percent of computer concepts.

Novice and intermediate programmers both generated associations that were classified as natural language association and as computer associations. However, novice students produced more natural language concepts than did intermediate programmers. Table 16 reports the mean number of natural language concepts for novice and intermediate programmers.

McKeithen, Reitman, Rueter, and Hirtle (1981) also found that novices associate programming concepts with a rich variety of natural language associations. However, in their study, the novice group included both people with no prior programming experience and people with prior experience in BASIC and FORTRAN.

Since this task was administered only a few weeks into the semester, it was anticipated that the novices would not have well-developed schemata. Hence it was hypothesized that the novices would not generate a rich variety of computer

associations. Contrary to expectations, novice and intermediate programmers generated a similar number of computer concepts. These results are reported in Table 17.

Table 16. Mean Number of Natural Language Concepts for Novice and Intermediate Programmers of Average and High Inductive Reasoning (Gf)

	Inductive Reasoning (Gf) Average	High	ALL
Intermediate	37.33	11.33	24.33
Novice	42.33	28.50	35.42
ALL	39.83	19.92	

Note. N=12--three people per group.

Table 17. Mean Number of Computer Concepts for Novice and Intermediate Programmers of Average and High Inductive Reasoning (Gf)

	Inductive Reasoning (Gf) Average	High	ALL
Intermediate	43.00	59.67	51.34
Novice	35.67	65.50	50.59
ALL	39.34	62.59	

Note. N=12--three people per group.

When the same data on number of natural and computer concepts is analyzed from the perspective of inductive reasoning skills, some interesting patterns emerge.

Tables 16 and 17 both show that the main effect for Gf is greater than the main effect for programming experience. Subjects high in Gf produced fewer natural language concepts and more computer concepts than did subjects with average Gf scores. It is surprising that Gf is a better predictor of the number of computer-related concepts generated than is computer programming experience. This may be an artifact of sample size. It may also reflect the fact that most of the stimuli used in this task were identified as labels for different types of programming processes in Study 2. Furthermore, in Study 2 it was found that high Gf subjects appeared to focus more on representing the process whereby problems would be solved than did average Gf subjects. Thus, this somewhat surprising data in Table 17 may have a familiar explanation.

It is interesting to note that the proportion of the concepts generated by each group that were computer concepts follows a similar pattern as the scores for each group on the first exam in the course. This is shown in Table 18. This exam was taken approximately two weeks after the administration of this elaboration task.

Individual scores for number of concepts for both the Gf and experience dimensions are presented in Appendix F, Table 31. The individual ratings for subjects in that appendix are the result of averaging across all labels for each subject to achieve a single rating for that subject. Each rater did this process; then, the raters' ratings of each subject were averaged.

It was surprising to find that the novices produced on the average a similar

number of computer concepts as the students with intermediate level computer experience. A closer look at word production per concept sheds light on this finding.

Table 18. Percent of Total Concepts and Programming Course Exam Scores for Novice and Intermediate Programmers of Average and High Inductive Reasoning (Gf)

Programming Dimension	Inductive Reasoning	Elaboration Process		Course Exam 1	
		% of Total Concepts			
		Natural Language	Computer Language	Score	Grade
Intermediate	High	16.00	84.00	121.30	B
Intermediate	Average	46.50	53.50	97.00	C
Novice	High	30.30	69.70	107.00	C
Novice	Average	54.30	45.70	100.00[a]	C[a]

[a]Does note include Exam 1 scores and grades for the novice programmers of average inductive reasoning who dropped the course before the first exam.

Words Within Concepts

Hypotheses. No specific hypotheses were made.

Words were counted for all natural language and computer concepts. "Natural language words" refers to the words that compose the natural language concepts. For example, for the label "real" a natural language concept is "Dairy products carry this sign". This concept contains five words, consequently, it was given a score of 5 for word production. "Computer words" refers to the words that compose the computer

concepts. Again for the label "real", a computer concept is "It is important never to cross integer/real types which will cause a 'TYPE CLASH' error." This concept was given a score of 16 for word production.

Novices generated almost twice as many natural language words as experienced subjects. But once again, effects for Gf were larger: low Gf subjects generated more than twice as many natural language words as high Gf subjects. Results are shown in Table 19.

For computer words, on the other hand, effects for experience and Gf were comparable. These results are reported in Table 20. Intermediate level programmers generated more computer words than novices. So although the novices produced a similar number of computer concepts as the intermediate programmers, the experienced programmers' average word production per concept was higher.

Table 19. Mean Number of Natural Language Words for Novice and Intermediate Programmers of Average and High Inductive Reasoning (Gf)

	Inductive Reasoning (Gf)		
	Average	High	ALL
Intermediate	277.67	109.33	193.50
Novice	552.33	207.00	379.67
ALL	415.00	158.17	

Note. N=12--three people per group.

Table 20. Mean Number of Computer Words for Novice and Intermediate Programmers of Average and High Inductive Reasoning (Gf)

	Inductive Reasoning (Gf)		
	Average	High	ALL
Intermediate	762.67	912.33	837.50
Novice	555.67	796.67	676.17
ALL	659.17	854.40	

Note. N=12--three people per group.

A table summarizing the number of natural language and computer concepts per label for each group is reported in Appendix G, Table 32.

Summary

Thus far, comparisons have been made among the groups participating in Study 3 based on differences between natural language concepts and computer concepts. All groups displayed evidence of rich natural language and computer associations with the labels. Again, the labels designating schemata (i.e., related units of knowledge) were initially generated by the subjects in Study 2 (except the "test" label). The purpose of exploratory Study 3 was to uncover what knowledge is contained in the schemata of novice and intermediate level programmers. Thus far, the results suggest that both natural and computer associations are contained in the schemata for each experience/inductive reasoning group to different degrees. What levels of knowledge do these associations represent? For example, are the natural and computer language

associations mainly facts or definitions (declarative knowledge), or do they perhaps include information about how to process information (procedural knowledge)? Levels of knowledge expressed within each concept are investigated in the next section.

Levels of Knowledge Within Concepts

Are different levels of knowledge associated with the natural and computer language concepts? Are the levels of knowledge different for novice and intermediate programmers of different inductive reasoning skills who are all learning a new language?

Hypotheses. No specific hypotheses were made.

The concepts generated by the people in this study ranged from simple definitions to examples to solution strategies. To understand more about the kinds or levels of knowledge expressed within each concept, each concept was rated.

Rating Scale for Natural Language Concepts. The natural language concepts were rated on a two-point scale. One point was given for definitions and general associations. Two points were given for more elaborated statements including examples and solution-related associations. For example, the following subject-generated concept for the label "easy math problems" was given a rating of one: "anything that the calculator can help with." In response to the same label, a score of two was given to: "equations involving graphing lines, circles, parabolas, ellipses, hyperbolas."

Rating Scale for Computer Concepts. Since the focus of this research is computer-language acquisition, a greater level of discrimination was used in rating the computer concepts. Therefore, the computer concepts were rated on a three-point scale ranging from 3 to 5. Three points were given for definitions and general computer associations, four points for complex definitions and examples and five points for specific references to conditions and solution strategies for solving computer-programming problems. A rating of three was given to the following response for the label "procedure": "the part of a Pascal program that does all of the main work." In response to the same label, a rating of 4 was given to: "for example you wouldn't need to write code to print some blank lines several times. You would just have one procedure to do it and call that procedure anytime you needed blank lines printed." A rating of five was given for this response to the label "loop": "ex. for I:=1 to 10 do statement;".

Reliability Checks. Using a version of the Kuder-Richardson formula 20 for multiple raters, reliability checks were made on the ratings of both natural language and computer concepts. For the two raters, the reliability coefficients were .48 for natural language concepts and .80 for computer concepts. The raters were the same two teaching assistants mentioned previously. Each final concept rating was derived by averaging the ratings of the two raters.

Natural Language Concept Ratings. The majority of natural language concepts for all groups were simple definitions or general statements as opposed to examples or solution-related strategies. The means for the natural language concept ratings ranged from 1.25 to 1.41. These results are reported in Table 21.

Table 21. Means for Natural Language Concept Ratings for Novice and Intermediate Programmers of Average and High Inductive Reasoning (Gf)

	Inductive Reasoning (Gf) Average	High	ALL
Intermediate	1.25	1.33	1.29
Novice	1.35	1.41	1.38
ALL	1.30	1.37	

Note. N=12--three people per group.

Computer Concept Ratings. The means for the computer concept ratings for novice and intermediate groups ranged from 3.28 to 3.51 indicating that the majority of the computer concepts generated were simple definitions or general computer-related associations. These results are reported in Table 22.

The individual ratings for both natural language and computer concepts are presented in Appendix F, Table 31. These individual ratings for natural language and computer concepts are the result of averaging across all labels for each subject to achieve a single rating for that subject. Then both raters' ratings of that subject were averaged.

High Computer Concept Ratings. Of the 612 computer concepts rated, 226 or 37% received ratings of 4 or 5. (A rating of 4=complex definitions and examples; a rating of 5=conditions for action and solution strategies.) Who received these higher ratings? The largest number of complex definitions and examples, 86 or 42%, were

Table 22. Means for Computer Concept Ratings for Novice and Intermediate Programmers of Average and High Inductive Reasoning (Gf)

	Inductive Reasoning (Gf) Average	High	ALL
Intermediate	3.29	3.38	3.34
Novice	3.28	3.51	3.40
ALL	3.29	3.45	

Note. N=12--three people per group.

generated by the novices of high inductive reasoning. The intermediate programmers of high inductive reasoning produced 61 or 29% of the computer concepts rated 4. In contrast, the intermediate programmers of average inductive reasoning generated 31 or 15% of the complex definitions or examples. That is, they produced half as many as the intermediate programmers of high inductive reasoning. The novices of average inductive reasoning generated 28.5 or 14% of the concepts.

Who received the highest ratings for instances of conditions for action or solution strategies? For the intermediate programmer groups, only 2.5 computer concepts were rated 5, whereas 17.5 were rated 5 for the novice groups. Of the 17.5, 12.5 were from the novices of high inductive reasoning. Table 23 gives the total number of computer-language related concepts with high ratings generated by each subject.

Why did the intermediate programmers of high inductive reasoning receive fewer ratings of 4 and 5 than the novice subjects with high inductive reasoning? A

Table 23. Total Number of Computer-Language Related Concepts With High Ratings Generated by Each Programmer

	Complex Definitions & Examples	Conditions & Solution Strategy		Complex Definitions & Examples	Conditions & Solution Strategy
Intermediate Level Programmers					
	Average Inductive Reasoning			**High Inductive Reasoning**	
Subject 1	3.5	0.0	Subject 4	23.0	0.0
Subject 2	15.0	2.5	Subject 5	22.5	0.0
Subject 3	12.0	0.0	Subject 6	15.0	0.0
Totals	30.5	2.5	Totals	60.5	0.0
Means	10.17	.83	Means	20.17	0.0
Novice Programmers					
	Average Inductive Reasoning			**High Inductive Reasoning**	
Subject 7	17.5	2.0	Subject 10	37.5	9.5
Subject 8	9.5	3.0	Subject 11	21.0	2.5
Subject 9	1.5	0.0	Subject 12	27.5	.5
Totals	28.5	5.0	Totals	86.0	12.5
Means	9.5	1.67	Means	28.67	4.7

Note. The complex definitions and examples received a rating of four. Conditions for actions and solution strategies received a rating of five.

free-association task bears little or no resemblance to a problem-solving situation involving computer programming. Perhaps due to the low ecological validity of this task, experienced programmers did not tap, to the maximum extent, their knowledge bases from computer programming.

Error Analysis

Of the 973 concepts generated across all groups, only three major errors were made on computer concepts: one error by an intermediate programmer of average inductive reasoning; two computer concept errors by novices with high inductive reasoning. These concepts were not included in the totals. That is, no credit was given for the three major errors.

Twelve minor computer concept errors were generated. Nine of the twelve errors were made by novices. Of these nine minor errors, six were made by novices of high Gf and three by novices of average Gf. The three remaining minor errors were made by intermediate programmers of high Gf. Concepts with only minor errors were included in the totals.

Study 3 in Perspective

This exploratory study helped to illuminate knowledge contained in the schemata of intermediate and novice programmers. We have learned that both natural language and computer associations are contained in the schemata of novice and intermediate programmers of average and high inductive reasoning. Novices have a greater number of natural language associations whereas experienced programmers tend to have more computer associations with the labels that were generated in Study 2. Novice and intermediate programmers of high inductive reasoning generate more computer concepts than novice and intermediate programmers of average inductive reasoning. Furthermore, these subjects tend to produce more elaborated concepts, i.e., more complex definitions and examples. This free-association task helped to demonstrate the declarative knowledge of the subjects but not the procedural knowledge. Only

twenty concepts or 4% of the total computer concepts received ratings of 5 (indicating conditions for action and solution strategies). The novices of high inductive reasoning generated 12.5 of these concepts in contrast to zero concepts for the intermediate programmers of high inductive reasoning.

Perhaps the procedural knowledge contained in the schemata of intermediate programmers would be more apparent in an ecologically valid task such as solving computer programming problems. This leads us to Study 4.

CHAPTER VI

STUDY 4

Study 1 investigated the independent contributions of inductive reasoning skills and prior computer programming on Exam 1 achievement. Study 2 explored the differential effects of inductive reasoning skills and computer programming experience on categorization and problem representation. Study 3 investigated the particular contributions of inductive reasoning skills and programming experience on the contents of students' problem schemas.The purpose of Study 4 is to examine in greater detail the solution plans and programming achievements of novice and intermediate programmers of average and high inductive reasoning skills.

Study 4 was inspired by the research findings of Chi, Feltovich, and Glaser (1981), Snow and Lohman (1984), and Soloway, Ehrlich, Bonar, and Greenspan (1982).

Research Questions

1. Do novice and intermediate programmers of high and average inductive reasoning skills (Gf) generate correct solution strategies for solving the programming problems?

2. Is a correct solution strategy a good predictor of success? In other words, an individual may have a good basic approach but may not be able to execute the strategy plan in order to generate a correct solution to the programming problem.

3. Is successful strategy execution related to individual differences in programming experience and/or inductive reasoning skills (Gf)?

Method

Subjects

The same 12 subjects who participated in Study 3 participated in this study. The data for one subject, however, was unusable due to an audiotaping problem. Consequently, the intermediate group, high inductive reasoning skills (Gf), contained only two subjects--one advanced intermediate and one intermediate level programmer. The three remaining groups had three subjects in each group. The novice and intermediate groups were composed of students with extreme scores on the Raven's Advanced Progressive Matrices (Set 2), a measure of inductive reasoning (Gf). Average inductive reasoning refers to students who had scores of 25 and below; whereas high inductive reasoning (Gf) refers to subjects who had scores of 30 and above. Thus, the participants in Study 4 were: (a) three novices of average inductive reasoning skills, (b) three novices of high inductive reasoning skills, (c) three intermediate programmers (two intermediate and one advanced intermediate) of average inductive reasoning skills, and (d) two intermediate programmers (one intermediate and one advanced intermediate) of high inductive reasoning skills. The intermediate and advanced programmers are referred to as "intermediate" programmers throughout this study.

Subjects were paid $3.50 per hour. All subjects had a few weeks of exposure to the Pascal programming language in the context of the same course. The Study 4 task was administered approximately one week prior to the first examination in the course.

Materials

Ten problems were chosen from the thirty problems used in the categorization task in Study 2. The ten problem statements are listed in Appendix H. A teaching assistant for the introductory Pascal course chose the problems to be typical of those which students should be able to solve at that point in the course. During the pilot phase of the study, a student suggested an eleventh problem analogous to one of the ten problems because he was certain that the novice students would not have the knowledge to solve one of the problems. The problem under discussion was problem 8: Write a program which reads in 20 words and prints them out in alphabetical order. This problem requires string manipulations--a topic not yet covered in the introductory Pascal course. The analogous problem created by the student was: Write a program which reads in 20 integers and prints them out in ascending order.

Procedure

Each of the subjects was given the eleven problem statements. For each problem, subjects were asked to read the problem statement and to think aloud about the "basic approach" they would take toward solving the problem. They were encouraged to report all thoughts and hunches they had during the process of deciding upon this general plan for developing a program. Following this period, the subjects explicitly stated their general plan for solving the particular problem. Furthermore, they were asked to state the problem features that led to this plan.

For problems 8, 9, 10, and 11, subjects were asked to execute their solution strategies immediately after they stated their general plan for solving each problem. That is, subjects were asked to write programs in Pascal.

Results and Discussion

The audio tapes of the sessions were transcribed. The raters for this study were two teaching assistants for the course. Both were graduate students enrolled in a masters degree program in computer science. One rater read all the transcripts and scored the basic approach strategies and programming problems for each subject. A second rater scored the basic approach problems for the three subjects that were the most difficult for the first rater to score. In addition, the second rater scored one programming problem across all subjects. Reliability checks were made using a version of the Kuder-Richardson formula 20 for multiple raters. For the subjects who were difficult to score on the basic approach task, the reliability was .86. For the one programming problem across all students, the agreement between the raters was .69. Since the reliabilities were within an "acceptable" range, the scores for the main rater were used for all subjects on both basic approach and programming problems.

Hypotheses

No specific hypotheses were made.

Results for the Basic Approach Task (Problems 1-7)

The program plans (i.e., "basic approach") and the programs themselves were rated using the following scale: 1 = completely incorrect, 2 = poor, 3 = fair, 4 = good, 5 = very good. Table 24 shows the ratings of each student's basic approach for problems one through seven.

Contrary to expectations, the means were similar across all groups, with the intermediate programmers of average inductive reasoning skills receiving the lowest score. Programming experience appeared to be of little benefit in formulating a general

plan for writing a program. Once again, the larger differences were between students of average inductive reasoning skills and students of high inductive reasoning skills.

Table 24. Basic Approach Scores for Novice and Intermediate Programmers of Average and High Inductive Reasoning (Gf) for Problems 1-7

	Basic Approach Scores
Intermediate High Gf	3.4
Intermediate Average Gf	2.7
Novice High Gf	3.4
Novice Average Gf	3.1

Note. Scores are averages for seven problems. A score of 3 is a "Fair" score. A score of 4 is a "Good" score.

Results For the Programming Task

The results on the problems requiring both basic approach and programming (problems 8, 9, 10, and 11) are shown in Table 25. Problem 8 was considered a practice problem and therefore was not included in the analysis.

The ratings of "basic approach" or programs plans showed relatively little variability across the four subject groups. These three problems appeared to be more difficult than problems 1-7, where the average rating of "basic approach" was 3.1. Here, the average was only 2.4. If there is any relationship between program plans

Table 25. Three Programming Problems: Basic Approach and Programming Scores for Novice and Intermediate Programmers of Average and High Inductive Reasoning Skills (Gf)

		Program 1		Program 2		Program 3		Means	
		Basic Approach	Program Scores	Basic Approach	Program Scores	Basic Approach	Program Scores	**Basic Approach**	**Program Scores**
Intermediate High Gf	Subject 1	2	3	2	1	1	2	1.7	2.0
	Subject 2	4	5	4	4	4	3	4.0	4.0
Group Means		3	4	3	2.5	2.5	2.5	2.8	3.0
Intermediate Average Gf	Subject 3	4	5	2	1	4	2	3.3	2.7
	Subject 4	2	3	2	2	3	1	2.3	2.0
	Subject 5	2	3	1	2	2	4	1.7	3.0
Group Means		2.7	3.7	1.7	1.7	3	2.3	2.4	2.6
Novice High Gf	Subject 6	4	4	4	5	3	5	3.7	4.7
	Subject 7	2	2	2	1	1	4	1.7	2.3
	Subject 8	2	3	2	1	2	5	2.0	3.0
Group Means		2.7	3	2.7	2.3	2	4.7	2.4	3.3
Novice Average Gf	Subject 9	3	5	2	1	2	2	2.3	2.7
	Subject 10	3	2	2	1	2	1	2.3	1.3
	Subject 11	2	1	2	1	2	1	2.0	1.0
Group Means		2.7	2.7	2	1	2	1.3	2.2	1.7

Note. Scale: 1 = completely incorrect, 2 = poor, 3 = fair, 4 = good, 5 = very good

and learner characteristics for these more difficult problems, then it appears to be with an average of experience and Gf. Thus, ratings were lowers for the average Gf novices, highest for the high Gf intermediate programmers, and intermediate for the two mixed groups.

For ratings of the programs themselves, however, Gf was once again the more potent predictor than experience. The best programs were produced by high Gf novices and the worst programs by average Gf novices. Differences between high and average Gf intermediate programmers were in the same direction, but smaller. It is noteworthy, however, that the average Gf intermediate programmers did much better than the average Gf novices (means = 2.6 and 1.7, respectively).

Thus, although the correlation between the ratings of program plans and the program actually produced for the 11 subjects was fairly high ($r = .66$), the two scores appear to capture somewhat different aspects of performance. It appears that program plans can be developed by relying on programming experience and/or inductive reasoning abilities. When it comes to actually writing the program, however, average Gf students without experience did not perform as well as average Gf students with experience. Of course, sample sizes are perilously small and so these generalizations are merely hypotheses for future investigation.

Final Discussion

Consistent with results from Studies 2 and 3, the novice and experienced programmers with high inductive reasoning skills performed better than the novice and experienced programmers with average inductive reasoning skills. Gf or inductive reasoning skills are generally required when instruction involves new or unusual

stimulus conditions or when it requires decontextualization (Snow & Lohman, 1984). Program problems appeared to be particularly novel for the novice students, and so their performance was strongly related to Gf.

The finding that a good plan is a good predictor of a good program is consistent with the results of a study of Pascal learning by Soloway, Ehrlich, Bonar, and Greenspan (1982), who also found that choice of the appropriate looping strategy was a good predictor of programming success.

There were three instances out of ten, or 30%, where an incorrect basic approach was modified in the process of programming to produce a correct solution. Two novices of high inductive reasoning skills and one experienced programmer of average inductive reasoning skills were able to abandon their initial strategies for new correct strategies. The novices of high inductive reasoning skills (Gf) showed the greatest improvement in scores from basic approach to the actual programming of strategies. They were able to implement or modify their solution strategies in process to better meet the demands of the programming task. In contrast, the novice group with average inductive reasoning skills (Gf) did worse from basic approach scores to executing their solution strategies. (The means for the three basic approach and programming problems are in the last two columns in Table 25.)

In Chapter 1, it was argued that inductive reasoning skills are important in helping the problem solver design a plan for solving a problem. Furthermore, it was hypothesized that different levels of inductive reasoning skills should differentially effect the problem representations and, thus, the mental models students construct and, hence, their solutions to problems. Evidence from Studies 2-4 supports this

hypothesis. Again, it must be emphasized that this is exploratory research. Refinements of the tasks are needed as well as replications using many more subjects.

In the next chapter, the results from Studies 1-4 will be reviewed and discussed further. In addition, implications for instruction will be presented.

CHAPTER VII

FINAL DISCUSSION

Business and research firms need more programmers who have not only a strong knowledge base in computer science but who can also solve novel problems. Typically, only a small percent of the programmers in a given organization actually do the truly innovative programming, while the majority of the programmers do lower level design and coding. Some claim that this is because many programmers have difficulty solving novel problems. The roles of both novel problem skills (inductive reasoning skills) and domain specific knowledge were examined in Studies 1-4. Before summarizing the results of Studies 1-4, a brief historical review of the arguments for the role of general cognitive skills versus specialized domain knowledge will be presented. For a more detailed review, read Perkins and Salomon (1989).

Thirty years ago, achievement was widely held to be a product of finely-honed general problem-solving strategies. The particular knowledge base (e.g., chess patterns, programming knowledge) and its organization were incidental. This view was challenged and supplanted by the notion that expert performance is driven by a rich knowledge base of context specific schemata (e.g., deGroot, 1966; Chase & Simon, 1973; Reitman, 1976; Chi, Feltovich & Glaser, 1981).

> General heuristics appeared to be no substitute for the rich database of ramifications, stored in memory, accessed by recognition processes, and ready to go. Indeed, the broad heuristic structure of expert as contrasted to novice problem solving--the reasoning forward rather than reasoning backward--seemed attributable not to any heuristic sophistication on the part of the experts, but to the driving influence of the experts' rich database.

> General heuristics no longer looked as central or as powerful. (Perkins & Salomon, 1989, p.18)

Artificial intelligence research played a major role in this transition. Researchers progressed from generic programs (e.g., General Problem Solver) to expert systems. Research on transfer supported the view that the training of general cognitive skills had no clear benefits outside the specific domain in which it was taught.

However, recent results and theory have challenged the view of expert performance as driven primarily by a rich knowledge base (Perkins and Salomon, 1989). When experts face unfamiliar problems, they appear to apply many general strategies in addition to deploying domain specific principles (e.g., Clement, 1982). According to Perkins and Salomon, the general heuristics do not substitute for domain knowledge. They argue that general skills "operate in a highly contextualized way accessing and wielding sophisticated domain knowledge" (p.20). Perkins and Salomon suggest an analogy to explain the role of general cognitive skills in relation to domain specific knowledge:

> Cognitive skills are general tools in much the way the human hand is. Your hands alone are not enough; you need objects to grasp. . . Likewise, general cognitive skills can be thought of as general gripping devices for retrieving and wielding domain-specific knowledge, as hands that need pieces of knowledge to grip and wield and that need to configure to the kind of knowledge in question. (p.23)

In this research, the cognitive skills of interest are inductive reasoning skills, whereas the domain specific knowledge refers to prior computer programming experience. A summary of the results from each study will be presented next, followed by a brief discussion, implications for instruction, and conclusions.

Summary

Study 1

The goal of the first study was to determine whether individual differences in inductive reasoning abilities and prior computer programming experience make independent contributions to the prediction of first examination scores and course grades in an introductory computer programming course.

A model was created for predicting Exam 1 success. The model contained two components: inductive reasoning and computer-related knowledge base. The computer-related knowledge base included prior programming experience and ACT Math scores. A regression analysis revealed that each variable in the model made a significant and approximately equal contribution to the model. The model yielded an r-square of .31 for predicting Exam 1 success ($n = 52$). Predicting Exam 1 success was of particular interest in this study because many students drop the course as soon as they receive their Exam 1 scores. (By the 7th week of the course, 40% of the enrolled students had dropped the course.)

The same model was used for predicting course grades. A regression analysis revealed that the model was not a good predictor of final exam scores and an even poorer predictor of course grades.

Study 2

The purpose of Study 2 was to ascertain if novice programmers and intermediate level programmers, who are all enrolled in an introductory course in Pascal, have different organizational categories for their new knowledge. Do individual differences in inductive reasoning skills (Gf) influence categorization behavior?

A sorting task consisting of 30 computer programming problems was administered. The results for 17 novices and 16 intermediate level programmers of average and high inductive reasoning skills (Gf) were analyzed using multidimensional scaling and cluster analysis procedures. Results from the quantitative and qualitative analyses suggested that categorization behavior is influenced by the subject's level of inductive reasoning as well as by prior computer programming experience. As programming experience and inductive reasoning (Gf) increase, categories are based less on superficial features in the problem statements and more on underlying solution features or strategies.

Study 3

For purposes of this research it was assumed that the category descriptions provided by novice and experienced programmers in Study 2 represent labels used to access related units of knowledge, i.e., schemata. The purpose of Study 3 was to uncover what knowledge is contained in the schemata of novice and intermediate level programmers of average and high inductive reasoning skills.

Six novices of average and high inductive reasoning skills (Gf) and six intermediate level programmers of average and high inductive reasoning skills (Gf) completed a free association task based on category labels generated by subjects in Study 2. Results indicated that both natural and computer associations are contained in the schemata of novice and intermediate level programmers of average and high inductive reasoning skills (Gf). Novice and experienced programmers of high inductive reasoning skills (Gf) generated a greater number of computer concepts and more complex computer concepts. That is, the novice and intermediate level subjects

of high inductive reasoning skills (Gf) created more complex definitions and examples than the novice and intermediate level subjects of average inductive reasoning skills (Gf).

Study 4

The purpose of Study 4 was to examine in greater detail the solution plans and programming achievements of novice and intermediate level programmers of average and high inductive reasoning skills (Gf). Eleven of the twelve subjects who participated in Study 3 participated in this study. The subjects were asked to think out loud while generating their basic approaches to eleven computer programming problems. For the last four problems, subjects wrote programs.

Ratings of the quality of program plans did not vary markedly across the four groups of subjects. For easier problems, Gf was the best predictor of the rated quality of plans; for more difficult problems, both Gf and programming experience were important, perhaps both being required for the generation of a good plan.

For the three problems requiring programming, a correct basic approach was a good predictor of programming success. The novices of high inductive reasoning skills (Gf) as a group showed the greatest improvement in scores from the basic approach to actual programming. On the other hand, the novice group with average inductive reasoning skills (Gf) did worse translating basic approaches into solution strategies. Differences between high and average Gf experienced programmers were in the same direction, but much smaller.

Discussion

Why are intermediate level programmers of average inductive reasoning skills

(Gf) performing poorer than the intermediate programmers of high inductive reasoning skills (Gf)? Likewise, why are novices of average inductive reasoning skills (Gf) performing worse than novices of high inductive reasoning skills (Gf)? Clearly, inductive reasoning abilities are important contributors to good performance on many of the programming tasks used in this study. The fact that such skills are generally better predictors of performance than the amount of programming experience is itself an important contribution to the literature on the nature of expertise in computer programming. But what are thinking skills, here called "inductive reasoning," and how do they produce the effects observed in these studies? Snow (1980) offers one hypothesis. His explanation focuses on the difference between estimates of prior learning or achievement in a particular domain (called crystallized abilities or Gc) and estimates of inductive reasoning (called fluid ability or Gf):

> Gc may represent prior assemblies of performance programs retrieved as a system and applied anew in instructional situations not unlike those experienced in the past, whereas Gf may represent new assemblies of performance processes needed in more extreme adaptations to novel situations. The distinction, then, is between long-term assembly for transfer to familiar new situations versus short-term assembly for transfer to unfamiliar new situations. (p.37)

Here, "assembly" refers to the process whereby the learner produces an internal program or sequence of operations for solving a task. Assemblies can be of either conceptual or procedural knowledge. With practice, particular assemblies can become organized and retrieved as crystallized units. "Component" refers to a cognitive operation which transforms one representation into another. Whenever the learning conditions are similar to those in which the crystallized units have been useful in the past, the learner will most likely retrieve old assemblies and apply them in the new situation (Snow & Lohman, 1984). For example, experienced programmers with high

inductive reasoning skills recognized several programming problems as a particular problem type--sort problems. However, when problems are unfamiliar, then the learner must actually assemble, implement, and revise a solution strategy while attempting to solve the problem.

The results from Study 1 indicated that during the early stages of learning a new programming language, inductive reasoning is a particularly strong correlate of Exam 1 achievement (r = .39) and Exam 2 achievement (r = .45). The role of prior computer experience (crystallized knowledge) is important but not as striking, and decreases as the students progress from Exam 1 (r = .30) to Exam 2 (r = .25). The results from exploratory Studies 2-4 further support the importance of inductive reasoning skills in categorizing and representing problems, in schema development, in solution plans, and in programming achievement. Intermediate and novice programmers of high inductive reasoning skills (Gf) were better able to recognize when to apply their previously formed crystallized units and were more able to form new assemblies whenever previously crystallized units could not be applied. That is, learners with high inductive reasoning skills were able to flexibly adapt their performance programs to meet the demands of the situation.

As previously mentioned, the first step in Reif 's (1979) problem-solving model involves representing or redescribing any problem in terms of concepts provided by the knowledge base. This knowledge base is organized according to problem schemata, each of which, Reif hypothesizes, contains information necessary to solve a specific category of problem. In the process of identifying a problem as an instance of a particular type of problem, associations cue information in the knowledge base.

Chi, Feltovich, and Glaser (1981) found that physics experts classified physics

problems on the basis of the underlying physics principles that were needed to solve the problems, whereas novices tended to organize their knowledge around superficial features in the problem statements. In this study, intermediate level programmers organized their knowledge around both superficial and secondary features in the problem statements. The intermediate level programmers of high inductive reasoning skills (Gf) tended to organize their knowledge around solution strategies more than the intermediate subjects of average Gf. More specifically, in Studies 2, 3, and 4, intermediate level programmers exhibited signs of transfer from prior programming languages (e.g., algorithms). However, the results were more striking for the intermediate level programmers of high inductive reasoning skills (Gf). In contrast, the intermediate level programmers of average Gf displayed more novice type behavior (e.g., tendency towards categorizing problems by surface features). Data gathered in Study 4 on the basic approach and programming task were particularly striking in this respect. Intermediate level programmers of average inductive reasoning skills performed poorly in this novel situation.

Perhaps the intermediate level programmers with average inductive reasoning skills have problem schemata that are more syntax-based. That is, perhaps their prior programming knowledge is based more on the basic elements of language code. Syntax knowledge is precise and arbitrary and, therefore, more easily forgotten. Semantic knowledge, on the other hand, includes the meaning of a statement or a program module. It is more or less independent of the syntactic knowledge of a particular programming language. Perhaps the intermediate level programmers of high inductive reasoning skills (Gf) have their programming knowledge base arranged more around semantics than syntax and, consequently, perform better when learning a new

programming language. Perhaps their knowledge base includes strategies for learning new languages.

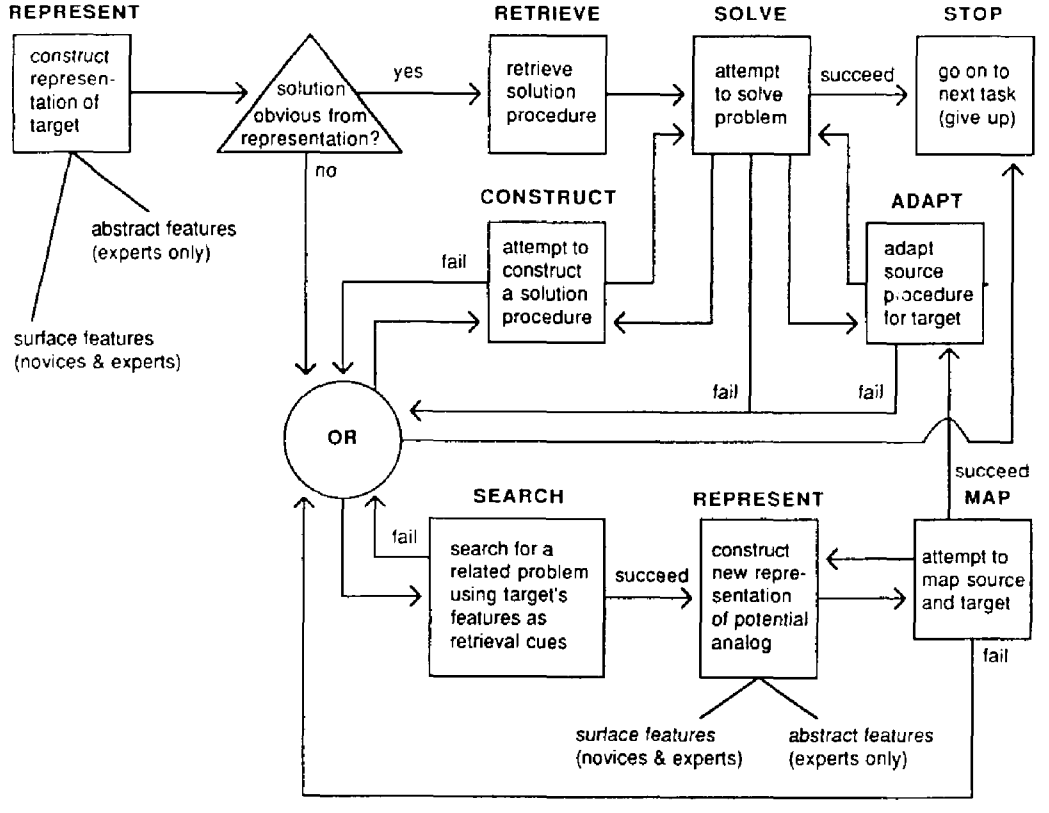
Another related hypothesis for the poor performance of the intermediate level programmers of average Gf is that they do not recognize analogous problem solving situations. Figure 12 shows one way to model the process of analogical transfer in expert and novice problem solvers. If a previous solution is represented by surface detail, then it will be retrieved by surface detail and hence will not aid the problem-solving process. If the surface detail and the solution strategy are one and the same, this does not pose a problem. However, in many situations the surface details are only moderately related to solution strategies. The intermediate level programmers of high inductive reasoning skills exhibited a greater tendency towards representing problems by solution features and strategies.

Perhaps the intermediate programmers did not perform as well as anticipated because of failure to retrieve relevant knowledge. Perkins, Schwartz, and Simmons (1988) found that prompting students enabled them to recall and apply a command correctly. The problem was not missing knowledge but "fragile knowledge", that is, knowledge not tied to the conditions of its use.

Implications for Instruction

This exploratory study should be refined and aspects replicated before any strong inferences are made concerning changing introductory programming language courses. The preliminary results, however, do suggest a need for alternative methods of teaching computer languages at the introductory level. Resnick (1976) reached a conclusion that is relevant in this context.

Figure 12. Framework for Conceptualizing Problem Solving and Analogy



Note. From Analogical Transfer in Expert and Novice Problem Solvers by L.R. Novick, 1986. Unpublished doctoral dissertation, Stanford University, CA.

> ...differences in learning ability--often expressed as intelligence or aptitude--may in fact be differences in the amount of support individuals require in making the simplifying and organizing inventions that produce skilled performance. Some individuals will seek and find order in the most disordered presentations; most will do well if the presentations (i.e., the teaching routines) are good representations of underlying structures; still others may need explicit help in finding efficient strategies for performance. (p.78)

Perhaps a different approach to teaching a programming language to students of average inductive reasoning skills would influence subsequent transfer to other programming languages. It is important to note that the mean score on the inductive reasoning task was 26.64 (S.D.= 3.67) for the 75 students who participated in Study 1. The projected mean score for the students enrolled in the course prior to Exam 1 was hypothesized to be a point or two lower. That is, the inductive reasoning ability of a typical student was probably at the upper end of the average range of inductive reasoning abilities in this population of students. Therefore, the following implications are relevant not just for a few students at the low end of the distribution but are relevant for the majority of the students initially enrolled in the course.

Snow and Lohman assert that for lower Gf learners the instructional treatment should be made explicit, direct, and structured. This type of approach provides learners with the conceptual and procedural knowledge necessary for success. It gives the learners the knowledge they may not be able to provide for themselves. Snow and Lohman also recommend that instructional treatments for lower Gf learners should include guided assembly and control during learning. "Direct training for low Gf learners, conducted in parallel with instructional treatments, should aim at the development of specified learning skills, and their flexible adaptation to real, instructional learning problems" (p.56). Many of the following suggestions from

educators and researchers may be useful for teaching introductory programming skills.

Perkins, Schwartz, and Simmons (1988) suggest a metacourse for enhancing the learning of programming. The metacourse developed at Harvard's Educational Technology Center consists of nine lessons "formulated to equip the students with thinking and learning heuristics specialized to learning programming" (p.159). For example, this course teaches a systematic approach to decomposing problems and a systematic framework for learning new commands. Research results indicate a substantial positive effect of the metacourse on the programming performance of high school students learning the BASIC programming language. The metacourse participants outperformed the control groups in all major categories including simple commands, hand execution, debugging, and program production.

Another suggestion for training was offered by deJong & Ferguson-Hessler (1986). These investigators found that good novice problem solvers had their knowledge arranged around problem types to a greater degree than poor problem solvers. The results of Studies 2, 3, and 4 indicated that after a few weeks of instruction in an introductory Pascal course, novice students with high inductive reasoning had their knowledge arranged around problem types to a greater degree than novice students of average inductive reasoning. When tested a few weeks later (Exam 1), novices of high Gf as a group were better problem solvers than novices of average Gf. DeJong and Ferguson-Hessler suggest that students should be taught to recognize problem types. This appears to be a promising approach.

Another approach to teaching introductory computer programming has been suggested by Dyck and Mayer (1989). Their research supports the use of a sequential method of language instruction in which semantics are taught using natural

language before the syntax of a new programming language is taught. Some computer scientists specializing in the field of programming language strongly support a more semantic approach to teaching introductory computer programming (C. Haynes, personal communication, June, 1988).

Conclusions

Inductive reasoning skills develop with education and experience (Snow & Lohman, 1988). The computer science curriculum can encourage their development by teaching novel problem-solving skills in the context of teaching computer programming. This is particularly important in the light of recent reseach which examines the cost of expertise--rigidity. Frensch and Sternberg (1987) claim that "the same cognitive mechanisms that produce experts' superior performance on familiar tasks also hinder their performance when task demands change" (p.1). Perkins (1988) recently proposed a conception of expertise that includes both "expertise" in a domain and flexible thinking. For Perkins, understanding a domain includes a "flexible grasp of the deep structures of the domain and justification, and a flexible capacity to deploy domain knowledge in unconventional cases" (p.3).

Some leaders in the field of computer science research have already begun exploring ways to foster flexible thinking. Soloway, Spohrer, and Littman (1988) suggest methods for teaching college students to explore alternative ways of solving the same programming problem. They present heuristics that can be taught to students "so that they can productively control their problem solving processes and avoid becoming locked into a single approach" (p.137). Other approaches have been advanced by researchers with different perspectives (e.g., Nickerson, Perkins, &

Smith, 1985). The studies reported in this dissertation do not address the issue of how to develop such thinking skills. But the studies reported here do show the importance of such skills in learning a new programming language. Experience in computer programming is also important, although clearly not as important as inductive reasoning skills during the early stages of learning a new language. When inductive reasoning skills were pitted against programming experience, inductive reasoning generally was the more important variable influencing performance. This finding was true for tasks involving (a) categorizing and representing problems, (b) schema development, and (c) solution plans and programming achievement. Of course, such conclusions cannot be easily generalized, given the many limitations of these studies. On the other hand, although samples here were small, subjects were systematically selected to be representative of the full range of experience and ability found in the original class. Thus, there are both strengths and limitations here.

Need for Further Research

Further research is needed on the differential effects of inductive reasoning and computer programming experience on knowledge representation and achievement in computer programming. Relevant issues for future study include:

1. How can novice and experienced programmers organize their knowledge so that it has a deep generic structure for facilitating the learning of subsequent programming languages?

2. How can programmers best organize their knowledge in preparation to solve novel problems?

3. What strategies can be taught in the context of a programming course that cultivate flexible thinking in the domain?

4. How can novice and intermediate programmers use metaphor and analogy to transfer information and procedures from one domain to another?

It is hoped that this dissertation research and the suggested research will help towards a process theory of aptitude for learning from instruction.

REFERENCES

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

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

Alspaugh, C. A. (1972). Identification of some components of computer programming aptitude. Journal of Research in Mathematics Education, 3, 89-98.

Anderson, J. R. (1976). Language, memory and thought. Hillsdale, NJ: Erlbaum.

Atwood, M. E., Jeffries, R., & Polson, P.G. (1980). Studies in plan construction. I: Analysis of an extended protocol (Report No. SAI-80-028 DEN). Englewood, CO: Science Applications, Inc.

Balzer, R., Goldman, N., & Wile, D. (1977). On the use of programming knowledge to understand informal process descriptions. Prodeedings of Pattern Directed Inference Workshop in Special Interest Group in Artifical Intelligence Newsletter, 63.

Barstow, D. R. (1977). A knowledge-based system for automatic program construction. Proceedings of the Fifth International Joint Conference on Artificial Intelligence, 382-388.

Blume, G. W. (1984, April). A review of research on the effects of computer programming on mathematical problem solving. Paper presented at the annual meeting of the American Educational Research Association.

Brooks, R. E. (1982). A theoretical analysis of the role of documentation in the comprehension of computer programs. Proceedings of the Conference on Human Factors in Computer Systems.

Butcher, D., & Muth, W. (1985). Predicting performance in an introductory computer science course. Communications of the Association for Computing Machinery, 28, 263-268.

Cetron, M., & O'Toole, T. (1982). Encounters with the Future: A Forecast of life into the 21st Century. New York: McGraw-Hill.

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

Chi, M. T. H., Feltovich, P. J., & Glaser, R. (1981). Categorization and representation of physics problems by experts and novices. Cognitive Science, 5, 121-152.

Clement, J. (1982). Analogical reasoning patterns in expert problem solving. Proceedings of the Fourth Annual Conference of the Cognitive Science Society. Ann Arbor, MI: University of Michigan.

Cronbach, L. J., & Snow, R. E. (1977). Aptitudes and instructional methods: A handbook for research on interactions. New York: Irvington.

Dalbey, J. (1983). Spider World reference manual and teacher's guide. (ACCEL Report). University of California, Berkley.

Dalbey, J., & Linn, M. C. (1985). The demands and requirements of computer programming: A literature review. Journal of Educational Computing Research, 1, 253-274.

deGroot, A. (1966). Perception and memory verus thought: Some old ideas and recent findings. In B. Kleinmuntz (Ed.), Problem solving. New York: Wiley.

deJong, T., & Ferguson-Hessler, M. G. (1986). Cognitive structures of good and poor novice problem solvers in physics. Journal of Educational Psychology, 78(4), 279-288.

Dyck, J., L. & Mayer, R. E. (1989). Teaching for transfer of computer program comprehension skill. Journal of Educational Psychology, 81(1), 16-24.

Egan, D. E., & Schwartz, B. J. (1979). Chunking in recall of symbolic drawings. Memory & Cognition, 7, 149-158.

Estes, W. K. (1982). Learning, memory and intelligence. In R.J. Sternberg (Ed.), Handbook of Human Intelligence (pp. 170-224). MA: Cambridge University Press.

Fenker, R. M. (1975). The organization of conceptual materials: A methodology for measuring ideal and actual cognitive structures. Instructional Science, 4, 33-57.

Ferguson, G. A. (1954). On learning and human ability. Canadian Journal of Psychology, 8, 95-112.

Ferguson, G. A. (1956). On transfer and the abilities of man. Canadian Journal of Psychology, 10(3) 121-131.

Frensch, P. A., & Sternberg, R. J. (1987). Expertise and flexibility: The costs of expertise. Manuscript submitted for publication.

Glaser, R. (1980). General discussion: Relationships between aptitude, learning, and instruction. In R.E. Snow, P. A. Federico, & W. E. Montague (Eds.), Aptitude, learning, and instruction, Vol. 2: Cognitive process analysis of learning and problem solving (pp. 309-326). Hillsdale, NJ: Erlbaum.

Hayes, J. R., & Simon, H. A. (1977). Psychological differences among problem isomorphs. In N.J. Castellan, Jr., D.V. Pisoni, & G.R. Potts (Eds.), Cognitive theory, Vol. 2 (pp. 21-41). Hillsdale, NJ: Erlbaum.

Holland, J. H., Holyoak, K. J., Nisbestt, R. E., & Thagard, P. R. (1986). Induction: processes of inference, learning and discovery. MA: MIT Press.

Hunt, J. M. (1961). Intelligence and experience. New York: The Ronald Press.

Kovalina, J., Wileman, S. A. & Stephans, L. J. (1983). Math proficiency: a key to success for computer science students. Communications of the Association for Computing Machinery, 26, 377-382.

Kurtz, B. B. (1980). Investigating the relationship between the development of abstract reasoning and performance in an introductory programming course. Special Interest Group Computer Science Education Bulletin, 12, 110-117.

Larkin, J. H. (1977). Problem solving in physics. Unpublished manuscript, University of California, Group in Science and Mathematics Education and Department of Physics, Berkeley.

Linn, M.C. (1985). The cognitive consequences of programming instruction in classrooms. Educational Researcher, 14(5), 14-29.

Lucas, H. D., & Kaplan, R. B. (1974). A structured programming experiment. The Computer Journal, 19(2), 136-138.

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

Mayer, R. E. (1983). Thinking, problem solving, cognition. San Francisco: W.H. Freeman & Co.

Mayer, R. E. & Dyck, J. L. (1984). Work problem translation test. Unpublished manuscript, University of California, Department of Psychology, Santa Barbara.

Mayer, R. E., Dyck, J. L., & Vilberg, W. R. (1985). A three minute mathematics test that predicts success in learning BASIC. Paper presented at the annual meeting of the American Educational Research Association, Chicago.

McDermott, J., & Larkin, J. H. (1978). Re-representing textbook physics problems. Proceedings of the 2nd National Conference, the Canadian Society for Computation Studies of Intelligence. Toronto: University of Toronto Press.

McKeithen, K. B. Assessing knowledge structures in novice and expert programmers. Unpublished doctoral dissertation. University of Michigan, 1979.

McKeithen, K. B., Reitman, J. S., Rueter, H. H., & Hirtle, S. D. (1981). Knowledge organization and skill differences in computer programmers. Cognitive Psychology, 13, 307-325.

Miller, M. L., & Goldstein, I. P. (1977). Problem solving grammers as formal tools for intelligent CAI. Proceedings of Association for Computing Machinery '77.

Miller M. L., & Goldstein, I. P. (1977). Structured planning and debugging. Proceedings of the Fifth International Joint Conference on Artificial Intelligence (pp. 773-779). Cambridge, MA.

Miller, M. L., & Goldstein, I. P. (1979). Planning and debugging in elementary programming. In P. H. Winston & R. H. Brown (Eds.), Artifical Intelligence: An MIT Perspective Vol. 1 (pp. 317-337). Cambridge, MA: MIT Press.

Newell, A., & Simon, H. (1972). Human Problem Solving. Englewood Cliffs, NJ: Prentice-Hall.

Nickerson, R., Perkins, D. N., & Smith, E. (1985). The teaching of thinking. Hillsdale, NJ: Lawrence Erlbaum Associates.

Novick, L. R. (1986). Analogical transfer in expert and novice problem solvers. Unpublished doctoral dissertation, Stanford University, CA.

Pea, R. D., & Kurland, D. M. (1983). On the cognitive prerequisites of learning computer programming, Project Report to the National Institute of Education. New York: Bank Street College of Education, Center for Children and Technology.

Pea, R. D., & Kurland, D. M. (1984). On the cognitive effects of learning computer programming. New Ideas in Psychology, 2(2), 137-168.

Pennington, N. (1982). Cognitive components of expertise in computer programming: A review of the literature (Tech. Rep. No. 46). Ann Arbor, MI: University of Michigan.

Perkins, D. N. (1988, April). Understanding and expertise: The double helix of mastery. Paper presented at the meeting of the American Educational Research Association, New Orleans.

Perkins, D. N., & Salomon, G. (1989). Are cognitive skills context-bound? Educational Researcher, 18, 16-25.

Perkins, D. N., Schwartz, S., & Simmons, R. (1988). Instructional strategies for the problems of novice programmers. In R. E. Mayer (Ed.), Teaching and learning computer programming (pp. 137-152). Hillsdale, NJ: Lawrence Erlbaum Associates.

Peterson, C. G., & Howe, T. G. (1979). Predicting academic success in the introduction to computers. Association for Education Data Systems Journal, 12, 182-191.

Polya, G. (1973). How to solve it: A new aspect of mathematical method (2nd ed.). NJ: Princeton University Press.

Raven, J. C. (1977). Advanced progression matrices, Sets 1 and II. New York: Psychological Corporation.

Raven, J. C., Court, J. H. & Raven, J. (1977) Manual for Raven's Progressive Matrices and Vocabulary Scales. London: Lewis & Co. Ltd.

Reif, F. (1979). Cognitive mechanisms facilitating human problem solving in a realistic domain: The example of physics. Unpublished manuscript.

Reitman, J. A. (1976). Skilled perception in Go: Deducing memory structures from inter-response times. Cognitive Psychology, 8, 336-356.

Resnick, L. B. (1976). Task analysis in instructional design: Some cases from mathematics. In D. Klahr (Ed.), Cognition and instruction (pp. 51-80). Hillsdale, NJ: Lawrence Erlbaum Associates.

Ricardo, C. M. (1983). Identifying student entering characteristics desirable for a first course in computer programming. Unpublished doctoral dissertation, Columbia University.

Rich, C. A. (1981). Formal representation for plans in the programmer's apprentice. Proceedings of International Joint Conference on Artificial Intelligence (pp. 1044-1052). Vancouver, BC.

Rich, C., & Shrobe, H. E. (1979). Design of a programmer's apprentice. In P.H. Winston & R.H. Brown (Eds.), Artificial Intelligence: An MIT perspective (pp.137-173). Cambridge, MA: MIT Press.

Rumelhart, D. E.(1981). Schemata: The building blocks of cognition. In R. Spiro, D. Bruce, & W. Brewer (Eds.), Theoretical issues in reading comprehension. Hillsdale, NJ: Erlbaum.

Rumelhart, D. E., & Norman, D. A. (1978). Accretion, tuning, and restructuring: Three modes of learning. In J. W. Cotton and R. Klatzky (Eds.), Semantic factors in cognition (pp. 37-53). Hillsdale, NJ: Erlbaum.

Salomon, G., & Perkins, D. N. (1987). Transfer of cognitive skills from programming: when and how? Journal of Educational Computing Research, 3(2), 149-169.

SAS Institute Inc. (1985). SAS user's guide: Statistics (5th ed.). Cary, NC: SAS Institute Inc.

SAS Institute Inc. (1986). SUGI supplemental library user's guide (5th ed.). Cary, NC: SAS Institute Inc.

Sauter, V. L. (1986). Predicting computer programming skill. Computer Education, 10(2), 299-302.

Shavelson, R. J., & Stanton, G. C. (1975). Construct validation: Methodology and application to three measures of cognitive structure. Journal of Educational Measurement, 12(2), 67-85.

Sheppard, S. B., Curtis, B., Milliman, P., & Love, T. (1979). Modern coding practices and programmer performance. Institute of Electrical and Electronic Engineers, 41-49.

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

Shneiderman, B. (1980). Software psychology: Human factors in computer and information systems. MA: Winthrop.

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

Silver, E. A. (1979). Student perceptions of relatedness among mathematical verbal problems. Journal for Research in Mathematics Education, 10, 195-210.

Simon, H. A. (1978). Information-processing theory of human problem-solving. In W. K Estes (Ed.), Handbook of learning and cognitive processes, Vol. 4 (pp. 271-295). Hillsdale, NJ: Erlbaum.

Snow, R. E. (1980). Aptitude processes. In R. E. Snow, P. Federico, & W. E. Montague (Eds.), Aptitude, learning and instruction: Vol. 1. Cognitive process analysis of aptitude (pp. 27-63). Hillsdale, NJ: Erlbaum.

Snow, R. E. (1981). Toward a theory of aptitude for learning: Fluid and crystallized abilities and their correlates. In M.P. Friedman, J. P. Das, & N. O'Connor (Eds.), Intelligence and learning (pp. 345-362). New York: Plenum.

Snow, R. E., & Lohman, D. F. (1984). Toward a theory of cognitive aptitude for learning from instruction. Journal of Educational Psychology, 76(3), 347-376.

Snow, R. E., & Lohman, D. F. (1988). Implications of cognitive psychology for education measurement. In R. Linn (Ed.), Educational Measurement (3rd ed.) (pp. 263-331). New York: Macmillan.

Snow, R. E., & Yalow, E. (1982). Education and intelligence. In R.J. Sternberg (Ed.) Handbook of Human Intelligence (pp.493-585). New York: Cambridge University Press.

Soloway, E., Ehrlich, K., Bonar, J., & Greenspan, J. (1982). What do novices know about programming? In A. Badre & B. Shneiderman (Eds.), Directions in Human Computer Interaction (pp. 27-54). Norwood, NJ: Ablex.

Soloway, E., Spohrer, J., & Littman, D. (1988). E unum pluribus: Generating alternative designs. In R. E. Mayer (Ed.), Teaching and learning computer programming (pp. 152-178). Hillsdale, NJ: Lawrence Erlbaum Associates.

Soloway, E., & Woolf, B. (1980). Problems, plans and programs. Special Interest Group in Computer Science Education Bulletin, 12(1), 16-24.

Taylor, H. G., & Mounfield, L. C. (1989). The effect of high school computer science, gender, and work on success in college computer science. Special Interest Group in Computer Science Education Bulletin, 21, 195-198.

Thro, M. P. (1978). Relationships between associative and content structure of physics concepts. Journal of Educational Psychology, 70, 971-978.

Waters, R. C. (1979). A method for analyzing loop programs. Institute of Electrical and Electronic Engineers Transactions on Software Engineering, SE-5, 237-247.

Wertheimer, M. (1959). Productive thinking (enlarged edition). New York: Harper.

Wills, J. A. (1982). Are programming and natural language skills related? Communications of the Association for Computing Machinery, 25, 221.

APPENDIX A

STUDY 1: LEARNER CHARACTERISTICS QUESTIONNAIRE

1. Is this your first programming language? Yes = 0, No = 9.
2. Is English your native language? Yes = 0, No = 9.
3. What is your current student status? Freshman = 0, Soph. = 1, Jr. = 2, Sr. = 3, Graduate = 4, Special = 5.
4. What is your major? Computer Science = 0, Math = 1, Natural Science = 2, Social Science = 3, Business = 4, Education = 5, Humanities = 6, Arts = 7, Health Science = 8, Other = 9.
5. What is your grade-point average? (1.5-1.9=0) (2.0-2.2=1) (2.3-2.5=2) (2.6-2.8=3) (2.9-3.1=4) (3.2-3.4=5) (3.5-3.6=6) (3.7-3.8=7) (3.9-4.0 =8)
6. What was your senior high school (grades 10-12) grade-point average? (1.5-1.9=0) (2.0-2.2=1) (2.3-2.5.=2) (2.6-2.8=3) (2.9-3.1=4) (3.2-3.4=5) (3.5-3.6=6) (3.7-3.8=7) (3.9-4.0 =8)
7. How many semesters of high school level math courses have you completed? (If more than 10, just mark 9).
8. How many semesters of college level math courses have you completed? (If more than 10, just mark 9.)
9. Is this course a high priority for you this semester? Low = 0, Med = 4, High = 9 (only use 0, 4, or 9).
10. Are you planning to take 22C:17 (Advanced Programming with Pascal)? Yes = 0, No = 9.

11. Do you expect to use a programming language in your future employment? Yes = 0, No = 9.
12. How often do you use a personal computer? Never = 0, Sometimes = 1, Frequently = 2.
13. Prior to this course, have you used a word processor? Yes = 0, No = 9.
14. For research purposes, may we have access to your ACT or SAT scores? Yes = 0, No = 9.
15. For research purposes, may we have access to your scores on assignments and tests in this course? Yes = 0, No = 9.

FOR PEOPLE WITH PREVIOUS COMPUTER PROGRAMMING EXPERIENCE

16. Prior to this course, have you studied top-down design (structured design of programming)? Yes = 0, No = 9.
17. How did you learn about top-down design (book, programming course)? Please be specific. Write out your answer.

The following directions pertain to questions 18 through 24:

* For each computer language you have studied in a course, indicate the number of semesters using circles 0 through 6.
* If you have studied a particular language for three weeks or less in a course or workshop setting, fill in circle 8.
* If you taught yourself the language, fill in circle 9.

	Language	Level
18.	BASIC	High School
19.	BASIC	College
20.	FORTRAN	High School
21.	FORTRAN	College
22.	COBOL	High School
23.	COBOL	College
24.	LOGO	(Fill in level)

25. Please specify other programming languages you have learned:

Language	How did you learn it?	Indicate your level of proficiency?
	Self-taught or course?	Beginning or Intermediate or Advanced

26. Indicate the languages in which you have had work experience.

APPENDIX B

STUDY 2: TABLES FOR REGRESSION ANALYSES FOR EXAM 2 AND FINAL

Table 26. Summary of Regression Analysis for Exam 2

Model	F Value	PR>F	R-Square
EXAM 2 = Inductive Reasoning, Computer Programming Experience, ACT Math	6.79	.0008	.33

Note: N = 46

Table 27. Contributions of Each Variable in the Model for Predicting Exam 2 Scores

Variable	Type III SS	F Value	PR>F
Inductive Reasoning	2241.13	4.62	.04
Computer Programming Experience	583.35	1.20	.28
ACT Math	3626.56	7.47	.01

Note: N = 46

Table 28. Summary of Regression Analysis for Final Exam

Model	F Value	PR>F	R-Square
Final Exam = Inductive Reasoning, Computer Programming Experience, ACT Math	1.63	.20	.10

Note: N = 46

Table 29. Contributions of Each Variable in the Model for Predicting Final Exam Scores

Variable	Type III SS	F Value	PR>F
Inductive Reasoning	178.34	0.08	.78
Computer Programming Experience	6767.79	2.99	.09
ACT Math	4413.46	1.95	.17

Note: N = 46

APPENDIX C

STUDY 2: PROBLEM STATEMENTS FOR SORTING TASK

For both parts 1 and 2, the problem numbers and letters are keyed to the problems in the multidimensional scaling figures in Study 2.

PART 1: Problems Organized by Surface Features

Money Related

K. A certain Swiss bank has ten customers, each with a unique account number. Write a program to read in ten pairs of account numbers and balances. Then your program should print them out so that an account with a larger balance is always listed before any of the accounts with smaller balances.

5. Write a program that computes the minimum number of coins and bills needed to make change for a particular purchase. The cost of the item and the amount tendered should be read as data values. Your program should indicate how many coins and bills of each denomination are needed for change. Use the following denominations.

    Coins: .05, .10, .25

    Bills: $1, $5, $10

U. The banks in your area all advertise different interest rates for various kinds of long-term savings certificates. The advertisements state the minimum interest period for the certificate (e.g., 2 years, 4 years, etc.) and the yearly interest rate. Write a program which, given an investment period in years, a yearly interest rate

in percent, and an amount of deposit in dollars and cents, will compute and print the yearly interest amount and the value of the certificate at the end of each year of the investment period.

O. Write a program to calculate the simple interest on a loan of less than one year's duration. The input parameters are the amount of the loan, the number of days it will be lent, and the interest rate.

E. Write a program which takes the current year, the current cost of a candy bar in dollars, and an annual inflation rate, and prints a table showing how much a candy bar will cost for each year during the next quarter century.

Geometry

Q. Given the values for the coefficients a, b, c, and d in the polynomial function: $f(x) = ax3 + bx2 + cx + d$ as well as two values for x such that $f(x0) < 0$ and $f(x1) > 0$, locate the point at which the function crosses the X axis (where $f(x) = 0$).

R. Write a program to read in the Cartesian coordinates for 16 points in space. Then the program should print out a neat looking table of the points, with the first point listed first, and the rest of the points listed in order of their proximity to the first one.

S. Given the number of vertices of a polygon and a series of Cartesian coordinates for its vertices, calculate its perimeter.

P. Write a program to read in the Cartesian coordinates for three points on a plane and then print 'TRUE' if they are collinear, or 'FALSE' if they are not.

7. Write a program to determine the length of the hypotenuse of a right triangle, given the lengths of the the other two sides.

Numbers

D. Write a program which reads a positive whole number N and determines whether or not N is a prime number. The program prints 'TRUE' if N is prime and 'FALSE' otherwise.

1. Given R, a positive real number, and N, a whole number greater than two, print a list of the vertices of an N sided polygon inscribed inside the circle with radius R and center at the origin.

T. Write a program that reads in 10 numbers; after reading each one, it should print the funny sum computed by the following rule: If the last number was greater than this one, then subtract this number from the previous funny sum, otherwise add this number to the funny sum. At the start, the funny sum is equal to zero.

J. Write a program to estimate cube roots to four decimal places. (The upper and lower limits for the value of the cube root of N are N and N / 3.)

L. Write a program which accepts two positive integers, X and Y, and prints XY (X to the Power Y).

N. Write a program which reads in two integers, each up to 40 digits long, and prints their product.

4. Write a program which reads in two integers, each up to 80 digits long, and prints their sum.

F. Write a program which reads a positive integer N, and prints out a list of the squares and cubes of all the integers from 1 to N inclusive.

6. Write a program which reads in two 5 by 5 matrices of numbers, multiplies them, and prints out the resulting matrix.

Letters/Words

3. Print all of the possible three letter words which contain at least one vowel and one consonant.

2. Write a program which reads a line of text and prints out each of the words correctly, but beginning with the last one entered. For the previous sentence it would print: '.entered one last the with beginning but correctly words the of each out prints and text of line a reads which program a Write'.

8. Write a program which reads in 20 words and prints them out in alphabetical order.

9. Write a program to read in a sample of natural-language text and print out a table showing the different alphabetic characters which appeared in the text along with the number of times each one appeared. Please have the table list the most frequently occuring characters first.

A. Write a program which reads in 100 characters and writes out a table showing the number of times each of the 26 letters of the alphabet appears.

G. Write a program which reads in a line of text and prints it out backwards. For the previous sentence it would print: '.sdrawkcab tuo ti stnirp dna txet fo enil a ni sdaer hcihw margorp a etirW'.

M. Write a program which reads a single line of characters and writes them out, replacing every sequence of consecutive blanks by a single blank.

C. Write a program to determine whether a five-letter word is a palindrome. A palindrome reads the same backwards and forwards.

I. Given a string of up to 80 characters, transform and print it according to the

following rule: If the current character is a caret (^) then convert the next character to upper case, otherwise print the current character.

H. Write a program to print all the members of the character set available on your computer keyboard, together with their ordinal numbers.

B. Write a program which reads in 150 names and addresses and organizes them in numerical order according to zipcode. Next, within each zipcode category arrange the names in alphabetical order.

PART 2: Problems Organized by Problem Types (Solution Oriented)

Simple

7. Write a program to determine the length of the hypotenuse of a right triangle, given the lengths of the other two sides.

O. Write a program to calculate the simple interest on a loan of less than one year's duration. The input parameters are the amount of the loan, the number of days it will be lent, and the interest rate.

P. Write a program to read in the Cartesian coordinates for three points on a plane and then print 'TRUE" if they are collinear, or 'FALSE' if they are not.

Loop

1. Given R, a positive real number, and N, a whole number greater than two, print a list of the vertices of an N-sided polygon inscribed inside the circle with radius R and center at the origin.

4. Write a program which reads in two integers, each up to 80 digits long, and prints their sum.

D. Write a program which reads a positive whole number N, and determines whether

or not N is a prime number. The program prints 'TRUE' if N is prime and 'FALSE' otherwise.

C. Write a program to determine whether a five-letter word is a palindrome. A palindrome reads the same backwards and forwards.

E. Write a program which takes the current year, the current cost of a candy bar in dollars, and an annual inflation rate, and prints a table showing how much a candy bar will cost for each year during the next quarter century.

F. Write a program which reads a positive integer N, and prints out a list of the squares and cubes of all the integers from 1 to N inclusive.

G. Write a program which reads in a line of text and prints it out backwards. For the previous sentence it would print: '.sdrawkcab tuo ti stnirp dna txet fo enil a ni sdaer hcihw margorp a etirW'.

H. Write a program to print all the members of the character set available on your computer keyboard, together with their ordinal numbers.

L. Write a program which accepts two positive integers, X and Y, and prints X^Y (X to the power Y).

S. Given the number of vertices of a polygon and a series of Cartesian coordinates for its vertices, calculate its perimeter.

U. The banks in your area all advertise different interest rates for various kinds of long-term savings certificates. The advertisements state the minimum interest period for the certificate (e.g., 2 years, 4 years, etc.) and the yearly interest rate. Write a program which, given an investment period in years, a yearly interest rate in percent, and an amount of deposit in dollars and cents will compute and print

the yearly interest amount and the value of the certificate at the end of each year of the investment period.

Multiple Loops

2. Write a program which reads a line of text and prints out each of the words correctly, but beginning with the last one entered. For the previous sentence it would print: '.entered one last the with beginning but correctly words the of each out prints and text of line a reads which program a Write'.
3. Print all of the possible three letter words which contain at least one vowel and one consonant.
5. Write a program that computes the minimum number of coins and bills needed to make change for a particular purchase. The cost of the item and the amount tendered should be read as data values. Your program should indicate how many coins and bills of each denomination are needed for change. Use the following denominations.

   Coins: .05, .10, .25

   Bills: $1, $5, $10

6. Write a program which reads in two 5 by 5 matrices of numbers, multiplies them, and prints out the resulting matrix.

N. Write a program which reads in two integers, each up to 40 digits long, and prints their product.

Sorts

8. Write a program which reads in 20 words and prints them out in alphabetical order.

9. Write a program to read in a sample of text and print out a table showing the different alphabetic characters which appeared in the text along with the number of times each one appeared. Please have the table list the most frequently occuring characters first.

A. Write a program which reads in 100 characters and writes out a table showing the number of times each of the 26 letters of the alphabet appears.

B. Write a program which reads in 150 names and addresses and organizes them in numerical order according to zipcode. Next, within each zipcode category arrange the names in alphabetical order.

K. A certain Swiss bank has ten customers, each with a unique account number. Write a program to read in ten pairs of account numbers and balances. Then your program should print them out so that an account with a larger balance is always listed before any of the accounts with smaller balances.

R. Write a program to read in the Cartesian coordinates for 16 points in space. Then the program should print out a neat looking table of the points, with the first point listed first, and the rest of the points listed in order of their proximity to the first one.

State Machine

I. Given a string of up to 80 characters, transform and print it according to the following rule: If the current character is a caret (^) then convert the next character to upper case, otherwise print the current character.

M. Write a program which reads a single line of characters and writes them out, replacing every sequence of consecutive blanks by a single blank.

T. Write a program that reads in 10 numbers; after reading each one, it should print

the funny sum computed by the following rule: If the last number was greater than this one, then subtract this number from the previous funny sum, otherwise add this number to the funny sum. At the start, the funny sum is equal to zero.

Reduction

J. Write a program to estimate cube roots to four decimal places. (The upper and lower limits for the value of the cube root of N are N and N/3.)

Q. Given the values for the coefficients a, b, c, and d in the polynomial function: f(x) = ax3 + bx2 + cx + d as well as two values for x such that f(x0) (0 and f (x) > 0, locate the point at which the function crosses the X axis (where f(x) = 0).

APPENDIX D

STUDY 2: SUBJECT RESPONSE FORM FOR SORTING TASK

Name:_____ I.D.#_____

Experience Category: Novice or Experienced (Circle one.) If experienced, please explain:

Directions: Enclosed are 30 typical introductory computer programming problems. You will not be asked to write programs. Instead, your task is to sort these computer programming problems into groups based on similarities of solution. In other words, sort these problems into groups on the basis of how you would solve them.

Note: Use only five categories.

Time limit: 25 minutes for the sorting aspect of the task (spend at least 15 minutes on this task).Allow 5-10 minutes to complete the answer sheet. Total time equals 30-35 minutes (maximum).

Your Name

Record your Starting Time_____

Record your Finishing Time_____

1. Finish Sorting Task_____
2. Finish Filling Out Form_____

Record answers below

Category A Member (list by numbers).

Give category A a name (5 words max.).

Why did you give category A this name?

Category B Members (list by numbers).

Give category B a name (5 words max).

Why did you give category B this name?

Category C Members (list by numbers).

Give category C a name (5 words max.)

Why did you give category C this name?

Category D members (list by numbers).

Give category D a name (5 words max.)

Why did you give category D this name?

Category E members (list by numbers).

Give category E a name (5 words max.)

Why did you give category E this name?

Indicate the five problems which were most difficult to categorize.

APPENDIX E

STUDY 2: SORTING TASK: NUMBER OF CATEGORIES FOR ALL GROUPS

Table 30. Sorting Task: Number of Categories for All Groups

	Natural Language						
	Problem Features			Problem Process			
	Surface Features	Organizing/Editing	Input Oriented	Simple Math	Complex Math	Outcome Oriented	Decision Makers
Novice (N=8) High Gf	10	5	2	4		6	5
Novice (N=9) Average Gf	25	6	2	4		4	3
Advanced Novice (N=14) High Gf	19	7	1	7	6	7	5
Advanced Novice (N=10) Average Gf	22	4	1	3		7	2
Intermediate (N=20) High Gf	44	4	2	11	5	9	4
Intermediate (N=4) Average Gf	10	2	2	1			1
Advanced Intermediate (N=11) High Gf	15	3	1	4	3	2	5
Intermediate (N=5) Average Gf	10	1				5	1

Table 30--continued

	Programming Knowledge					
	Problem Features		Problem Process			
	Data Types (except Boolean)	Boolean	Operations with Data Types	Output Oriented	Data Structures	Data Structures Plus Manipulation
Novice (N=8) High Gf	1			2		1
Novice (N=9) Average Gf						
Advanced Novice (N=14) High Gf	2		4	1	2	
Advanced Novice (N=10) Average Gf	1	1			3	
Intermediate (N=20) High Gf	2	2	7	1	1	
Intermediate (N=4) Average Gf	1		1			
Advanced Intermediate (N=11) High Gf	2	2	3			
Advanced Intermediate (N=5) Average Gf		1	1	1		

Table 30--continued

	Programming Knowledge							
	Problem Process							
	Loop	Nested Loops	Small Algorithm	Large Algorithm	Outline Approach	Procedures	Complicated Programs	No Credit
Novice (N=8) High Gf			1	2				1
Novice (N=9) Average Gf			1					
Advanced Novice (N=14) High Gf	2			1	1			5
Advanced Novice (N=10) Average Gf	1			1	3		1	
Intermediate (N=20) High Gf	1		2	3	2			
Intermediate (N=4) Average Gf				1		1		
Advanced Intermediate (N=11) High Gf	1		1	6	1	1	2	3
Advanced Intermediate (N=5) Average Gf	1			2				2

APPENDIX F

STUDY 3: NUMBER OF NATURAL AND COMPUTER LANGUAGE CONCEPTS AND RATINGS: DATA FOR INDIVIDUAL SUBJECTS

Table 31. Number of Natural and Computer Language Concepts and Ratings for Each Subject (S)

Intermediate Level Programmers	Inductive Reasoning Average				Inductive Reasoning High			
	S1	S2	S3	Means	S4	S5	S6	Means
No. Natural Language Concepts	27.00	1.00	84.00	37.33	20.50	5.50	8.00	11.33
No. Computer Language Concepts	34.00	37.00	58.00	43.00	42.50	88.50	48.00	59.67
Ratings 1-2 Natural Language	1.43	1.00	1.31	1.25	1.37	1.38	1.25	1.33
Ratings 3-5 Computer Concepts	3.08	3.54	3.25	3.29	3.58	3.26	3.31	3.38

Intermediate Level Programmers	Inductive Reasoning Average				Inductive Reasoning High			
	S7	S8	S9	Means	S10	S11	S12	Means
No. Natural Language Concepts	19.00	47.00	61.00	42.33	52.50	7.00	26.00	28.50
No. Computer Language Concepts	58.00	34.00	15.00	35.67	70.50	7.00	75.00	50.83
Ratings 1-2 Natural Language	1.27	1.48	1.30	1.35	1.48	1.33	1.42	1.41
Ratings 3-5 Computer Concepts	3.33	3.39	3.12	3.28	3.84	3.40	3.30	3.51

APPENDIX G

STUDY 4: NUMBER OF NATURAL LANGUAGE AND COMPUTER CONCEPTS PER LABEL FOR EACH GROUP

Table 32. Number of Natural Language and Computer Concepts Per Label for Each Group

		Novice Programmers Inductive Reasoning		Intermediate Level Programmers Inductive Reasoning	
		Average	High	Average	High
Money Operations	Simple Definitions	1	6.5	4	7.5
	Complex Definitions and Examples	0	1	0	0.5
Geometry	Simple Definitions	0	0	2	3.5
	Complex Definitions and Examples	0	0	0	1
Easy Math	Simple Definitions	3	3	2	7
	Complex Definitions and Examples	1	2	1	1
Complex Math	Simple Definitions	2.5	6	3.5	4.5
	Complex Definitions and Examples	2.5	1	3.5	3.5
Listings	Simple Definitions	5.5	5.5	5.5	8.5
	Complex Definitions and Examples	2.5	2	2	0.5
Characters	Simple Definitions	10	11.5	9	10.5
	Complex Definitions and Examples	0	5	0	1.5
Integers	Simple Definitions	5	4.5	5.5	4.5
	Complex Definitions and Examples	1	8.5	1	6

Table 32 -- continued

		Novice Programmers		Intermediate Level Programmers	
		Inductive Reasoning		Inductive Reasoning	
		Average	High	Average	High
Real	Simple Definitions	3.5	8	5	10
	Complex Definitions and Examples	6	5	1	4
Boolean	Simple Definitions	8.5	9	8.5	10.5
	Complex Definitions and Examples	4.5	6.5	2.5	6.5
Function	Simple Definitions	2	7.5	4.5	9.5
	Complex Definitions and Examples	3	5.5	0.5	5
Array	Simple Definitions	9	9	10.5	9.5
	Complex Definitions and Examples	2	6.5	2.5	5.5
Loop	Simple Definitions	5	3.5	11	2.5
	Complex Definitions and Examples	5.5	17	2.5	6.5
Multiple Loops	Simple Definitions	3.5	2	8	4.5
	Complex Definitions and Examples	4	9.5	3	7.5
Sort	Simple Definitions	3.5	5	6.5	9
	Complex Definitions and Examples	0.5	8	2.5	1
Procedures	Simple Definitions	7.5	9	8	5.5
	Complex Definitions and Examples	1.5	6.5	6	9.5
Test	Simple Definitions	4	7.5	1.5	9
	Complex Definitions and Examples	0	1.5	3.5	1

APPENDIX H

STUDY 4: PROBLEM STATEMENTS FOR BASIC APPROACH AND PROGRAMMING TASKS

Part 1 Basic Approach Problem Statements

1. Write a program which reads a line of text and prints out each of the words correctly, but beginning with the last one entered. For the previous sentence it would print: '.entered one last the with beginning but correctly words the of each out prints and text of line a reads which program a Write'.
2. Write a program that computes the minimum number of coins and bills needed to make change for a particular purchase. The cost of the item and the amount tendered should be read as data values. Your program should indicate how many coins and bills of each denomination are needed for change. Use the following denominations.

   Coins: .05, .10, .25

   Bills: $1, $5, $10
3. Write a program to determine whether a five-letter word is a palindrome. A palindrome reads the same backwards and forwards.
4. Write a program which reads a positive integer N, and prints out a list of the squares and cubes of all the integers from 1 to N inclusive.
5. Write a program which reads a single line of characters and writes them out, replacing every sequence of consecutive blanks by a single blank.

6. Write a program which reads in two integers, each up to 40 digits long, and prints their product.
7. Write a program to calculate the simple interest on a loan of less than one year's duration. The input parameters are the amount of the loan, the number of days it will be lent, and the interest rate.

Part 2 Basic Approach and Programming Problem Statements

8. Warm Up Problem: Write a program which takes the current year, the current cost of a candy bar in dollars, and an annual inflation rate, and prints a table showing how much a candy bar will cost for each year during the next quarter century.
9. Write a program which reads in 100 characters and writes out a table showing the number of times each of the 26 letters of the alphabet appears.
10. Write a program which reads in 20 words and prints them out in alphabetical order.
11. Write a program which reads in 20 integers and prints them out in ascending order.