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

by JAMES BRUGGEMAN

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Education
Montana State University

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

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

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

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

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

by

James George Bruggeman

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

of

Doctor of Education

MONTANA STATE UNIVERSITY
Bozeman, Montana

March 1985

APPROVAL

of a thesis submitted by

James George Bruggeman

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

3/26/85
Date

Chairperson, Graduate Committee

Approved for the Major Department

3/26/85
Date

Head, Major Department

Approved for the College of Graduate Studies

3-27-85
Date

Graduate Dean

STATEMENT OF PERMISSION TO USE

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

Signature James J. Ruggeman

Date March 18, 1985

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

Boris Pasternak, 1890-1960

TABLE OF CONTENTS

TABLE OF CONTENTS--Continued

TABLE OF CONTENTS--Continued

LIST OF TABLES

LIST OF TABLES--Continued

LIST OF TABLES--Continued

LIST OF TABLES--Continued

LIST OF TABLES--Continued

LIST OF FIGURES

Abstract

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

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

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

Chapter I.

INTRODUCTION

Problem-solving Research and Instruction

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

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

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

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

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

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

partial solutions for those solutions that satisfy the goal.

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

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

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

Computer Programming as Problem Solving

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

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

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

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

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

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

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

Learning Through Interactive Programming

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

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

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

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

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

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

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

Statement of the Problem

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

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

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

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

Need for the Study

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

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

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

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

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

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

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

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

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

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

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

Presentation of the Study

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

Summary

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

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

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

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

Chapter II.

REVIEW OF THE LITERATURE

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

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

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

Understanding and Problem Solving

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

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

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

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

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

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

representations in ways that provide direct access to effective strategies.

Procedural Specification

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Novice-Expert Distinctions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Barriers to Understanding Progamming Languages

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

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

Propositional Structures

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

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

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

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

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