1986

Using programming protocols to investigate the effects of manipulative computer models on student learning

Elizabeth June Bruene Hooper
Iowa State University

Hooper, Elizabeth June Bruene

USING PROGRAMMING PROTOCOLS TO INVESTIGATE THE EFFECTS OF MANIPULATIVE COMPUTER MODELS ON STUDENT LEARNING

Iowa State University PH.D. 1986

PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark __✓__.

1. Glossy photographs or pages _____
2. Colored illustrations, paper or print _____
3. Photographs with dark background _____
4. Illustrations are poor copy _____
5. Pages with black marks, not original copy _____
6. Print shows through as there is text on both sides of page _____
7. Indistinct, broken or small print on several pages __✓__
8. Print exceeds margin requirements _____
9. Tightly bound copy with print lost in spine _____
10. Computer printout pages with indistinct print _____
11. Page(s) _____ lacking when material received, and not available from school or author.
12. Page(s) _____ seem to be missing in numbering only as text follows.
13. Two pages numbered _____. Text follows.
14. Curling and wrinkled pages _____
15. Dissertation contains pages with print at a slant, filmed as received _____
16. Other _____

_____

_____

Using programming protocols to investigate the effects of manipulative computer models on student learning

by

Elizabeth June Bruene Hooper

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Department: Professional Studies in Education
Major: Education (Research and Evaluation)

Approved:

Signature was redacted for privacy.
In Charge of Major Work

Signature was redacted for privacy.
For the Major Department

Signature was redacted for privacy.
For the Graduate College

Iowa State University
Ames, Iowa

1986

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I: INTRODUCTION

Computer technology has become a vital part of the formal and informal education of our youth. Educators and students alike view computer skills as essential to success in a wide variety of careers. While not all careers will require an "expert" level of programming skills, a large number of individuals will need to communicate their intentions to the computer via a programming language.

Developing a functional knowledge of programming for an ever-increasing, diverse group of students has become one of the major challenges of computer science education. Many studies have been conducted in an effort to examine what can be done to facilitate the learning of computer programming. Properties of languages, characteristics of learners, and innovative instructional techniques have been studied extensively but results have been less significant than expected (Sheil, 1981). The question remains unanswered as to how educators can better help students learn about computers and computer programming.

One obvious reason that so many students fail at learning to program is because programming involves a complex set of skills. According to Pea and Kurland (1984) programming consists of a set of problem-solving activities including 1) understanding the task the program is to accomplish, 2) planning a programming strategy that will accomplish the task, 3) implementing the plan via a programming language, and 4) debugging the plan and the program code. In order to accomplish these activities, programmers must draw upon a large body

of ill defined knowledge. Identifying this knowledge and the organization or structure of this knowledge in memory may be the keys to unlocking the mysteries of becoming a successful programmer. What appears to be needed are more effective instructional methods that help novice programmers acquire and organize the knowledge needed for programming computers.

Probably the most effective instructional activities currently being used in introductory programming courses are the programming assignments themselves. Lectures and textbooks provide the students with information about the syntax and semantics of a particular programming language. The programming assignments, however, require the student to give meaning to this information. Although the programming assignments may be a good test of a student's ability to apply information, they are often frustrating because so much is involved. Not only must the student write programming code that will solve the task at hand, but she must also enter it into the computer, isolate and remove syntax errors, and determine whether the program does indeed satisfy the assignment. Most students are so consumed by this process that they fail to grasp many of the concepts that could be fundamental to a meaningful understanding of programming. Instructional activities that avoid some of the "mechanics" of getting a program to run and allow the student to focus on basic programming concepts should expedite the learning process.

A second reason so many students may fail in learning to program may be that they do not possess appropriate prerequisite knowledge

about how computers work. According to Mayer (1981) some knowledge about how computers work and what they can be instructed to do may be necessary for the meaningful learning of many programming concepts. In a series of studies, Mayer (1981) reported that students given a simplified static model of the operational components of a computer (described in familiar terms such as windows, scoreboards, and recipes) performed better on more difficult programming tasks than students who had not received the model. According to Mayer, allowing novices to "see the works" assisted their encoding process such that the information gained was encoded in a more coherent and meaningful way. Implicit in Mayer's findings is an indication that carefully designed, simplified, interactive models of computer operations and concepts should be a productive and efficient way to foster the meaningful learning of programming. The computer itself may be the most effective tool for establishing just such an environment.

In addition to being an object of instruction, computers can be mediums of instruction. For almost two decades now educators have been exploring the role computers can play in the teaching/learning process. Computers have been programmed to tutor, provide drill and practice, and simulate real-world events. Early research efforts in computer-assisted instruction (CAI) focused mainly on the feasibility of using computers to deliver instruction. These efforts did little more than substantiate the finding that students could learn from computers (Kearsley, 1977). When tested against instructional strategies that did not incorporate usage of the computer, computing

strategies produced performance scores that were similar to the scores of students who learned from other methods (Fletcher and Atkinson, 1972). Although a few studies documented a savings in learning time, the high cost of computer usage at that time usually neutralized the time-saving factor. Research prior to the late seventies failed to indentify instructional arenas in which the computer was clearly superior to traditional methodologies.

Instructional computing research in its infancy was extremely disappointing in view of the computer's versatility. Although it was documented that the computer could be an effective tutor and drill and practice device, very little research explored the computer's ability to perform more challenging instructional tasks. Only recently have educators begun to document the computer's capacity to act as an interactive environment that can be used by students to test new ideas and evaluate previously acquired models of understanding. Not only do these environments afford students the opportunity to "debug" their thinking (Papert, 1981; White, 1984), but they can also be used to study the roles that knowledge and knowledge acquisition play in the learning process.

Statement of the Problem

Two educational challenges, learning about computers and using computers to learn, logically merge in computer science education. In practice, however, this has not been the case. Computer science educators have only recently started to develop learning environments

that promote the acquisition of programming skills. Previously, this aspect of learning was only addressed by requiring students to write computer programs. Unfortunately, the research procedures for evaluating the effects of the new learning environments are lacking and the processes involved in learning to program are not yet understood. Thus, the problem to be investigated in this study was the fostering and documenting of the processes by which novices learn computer programming.

Goals of the Study

There were two related primary goals of the study. One was to investigate the effects of a manipulative computer model on novice learning of semantic knowledge and procedural literacy. The second was to determine whether programming protocols were useful tools in analyzing information about the procedural literacy aspect of computer programming. In the process of accomplishing these primary goals, a third goal of documenting the detailed behaviors of novice programmers as they attempted programming tasks was achieved.

Research Questions

There were three basic research questions addressed by the study. These were:

1. What are some of the observable programming behaviors of novices who are learning programming?

2. Do students who use a manipulative model before language instruction program differently than students who don't use the manipulative model?

3. Do programming protocols provide information useful in assessing aspects of student learning that cannot be measured using a paper-and-pencil test?

Limitations of the Study

The study was conducted in view of the following limitations:

1. It was necessary to collect and analyze a large amount of data for each participant in order to document the processes involved in learning programming. The scope of this task severely limited the size of the sample which could be studied.

2. The processes involved in programming are not well-defined; therefore, post hoc analyses based upon the presence or absence of solution features identified by the researcher were performed.

3. The panel of judges that classified the behaviors of the novice programmers consisted of only two individuals, the researcher and another person who had extensive programming instruction experience.

4. The programming behaviors on a limited number of programming tasks were analyzed.

5. The experimental subjects were from a single discipline.

6. Only one instructional simulation was used as the experimental treatment.

Definitions of Terms

Protocols - the history of a Pascal program or MEMOPS solution. Protocols of novice programming behavior for the posttest problems documented specific solution features of initial and final coding efforts as well as any intervening online programming problems. Protocols for the MEMOPS tasks documented intrinsic features of a student's solution algorithms.

Solution features - selected characteristics of a student's programming solution that were documented.

Algorithm - a solution procedure, plan, or approach that a student attempted to implement in solving a programming task.

Semantic knowledge - A multi-leveled set of concepts important for programming which have been "abstracted through experience and instruction . . . and are stored as general, meaningful sets of information that are more or less independent of the syntactic knowledge of a particular programming language or facility" (Shneiderman, 1980, p. 47).

Procedural literacy - "The process by which one determines the effect of a set of instructions, or alternatively, the set of instructions that will achieve a particular effect. It presumes not only the notion of information as a distinct entity, but also the separation of processor and instructions, a distinction between instances and general rules, and specialized versions of a whole collection of concepts . . . otherwise encountered only in mathematics" (Sheil, 1982, p. 83).

Tacit knowledge - ". . . knowledge that one needs in a given field but that is usually not explicitly taught or even verbalized" (Sternberg, 1986, p. 142).

CHAPTER II: LITERATURE REVIEW

In this chapter the research literature from computer science education and the psychology of learning which is germane to this study is reviewed. Initially, a brief history of the early research on computer programming is presented. This is followed by a summary of the research on the cognitive components of programming and general learning theory. In the final sections of this chapter new directions for programming education and considerations for evaluation are discussed.

Early Research on Programming and Programming Practices

Since the late 1960s a wealth of research has been conducted in an effort to investigate ways to produce more efficient programmers. The very first studies investigating programming stemmed from machine-related issues. Topics such as parsing ease, execution efficiency, and implementation of different character sets (Shneiderman, 1976) for the most part dealt only superficially with human factors involved in programming. One noted study, however, did focus on programming performance as a human activity. Grant and Sackman (1967) investigated programmer performance under interactive and batch processing conditions. Twelve experienced programmers coded and debugged two programs using either an interactive or batch processing facility. Results of the study slightly favored time-sharing for the debugging process only. However, the investigators noted that

individual variability in programming performance was the more striking finding.

The use of flowcharts, commenting, indentation, and meaningful variable names have all been advanced in programming instruction as desirable and beneficial in aiding programming performance. Empirically, the evidence supporting the use of these programming practices is at best tentative. Shneiderman et al. (1977) evaluated the utility of detailed flowcharting as an aid to various programming tasks and found no statistically significant difference between flowchart and non-flowchart group performance. Although Weissman (1977) found some evidence that well-placed, meaningful comments improved student's speed in tracing execution, Sheppard et al. (1979) found that commenting had no effect on accuracy of or time taken to modify FORTRAN code.

Indentation is a technique used by many programmers to present code in a much more readable format. However, research indicates that this technique does not measurably improve programming performance for a number of programming subtasks. Weissman (1977) found that indentation did not improve student performance on hand simulation tasks. Love (1977) found that comprehension (as measured by program reconstruction activities) was not improved through the use of indentation. Shneiderman and McKay (as reported in Shneiderman, 1980) investigated the ability to locate and modify programming errors in indented and unindented versions of two Pascal programs. In

accordance with the other studies, there was no performance advantage for groups using the indented versions.

Findings regarding the use of meaningful variable names in programs are also inconclusive. Newman (as reported in Shneiderman, 1980) found that students given programs with non-mnemonic variable names performed better on a program comprehension test than did students given mnemonic variable names. Shneiderman (1980) reported that while the use of mnemonic variable names did help novice programmers comprehend a program, they did not appear to help intermediate-level programmers locate errors or modify programs. Likewise, Sheppard et al. (1979) could find no evidence that the use of mnemonic variable names improved the recall performance of experienced programmers.

Besides examining common programming practices, numerous studies have examined the effects of language features on programming performance. The recent structured programming debate has spawned many investigations concerning language control structures. Again, the results of these studies have been inconsistent and disappointing in terms of measuring changes in programming performance.

Sime et al. (1977) conducted experiments with novice programmers on the three common styles of conditional statements found in programming languages. In general, they found that the use of nested conditional structures (IF...THEN Begin...End ELSE Begin...End; IF...NOT...;) led to the production of the highest number of logically correct programs but were the most difficult to debug. The jump

conditionals (IF...THEN GOTO...) were easier to debug, but were associated with more logic errors and incorrect programs. Similar results were later obtained by Green (1977) for experienced programmers. Miller (1974) found that nested conditionals were much more difficult to comprehend for novices than were jump conditionals.

The use of higher-level control structures such as DO-loops and WHILE-loops to replace lower-level IF tests and GOTO statements was studied by Weissman (1977), Lucas and Kaplan (1976) and Sheppard et al. (1979). Weissman measured comprehension and programming performance of both novice and experienced programmers and found no reliable differences for the use of structured constructs over the simpler constructs. Lucas and Kaplan discovered that although students who were only familiar with the GOTO types of structures struggled in attempting to write GOTO-less code, modification tasks on GOTO-less code were easier. Love (1977) and Sheppard et al. (1979) found that for both experienced and unexperienced programmers structured programs were much easier to recall than non-structured versions of programs. Although "chaotically" structured programs were significantly more difficult to recall and modify, no differences could be found between groups using the different kinds of structured control mechanisms.

Youngs (1974) and Gannon (1976) explored the types of errors programmers made based upon the language features utilized in different programming languages. Youngs noted that over one quarter of all programming errors occurred in assignment statements, but

conclusions about other language features could not be made as the languages under investigation implemented different features. Gannon attempted to control the differences between languages by altering nine mutually independent features of one language so that specific error comparisons could be made. Even though the overall error rates between groups of students using either the standard or modified version of the language did not differ, Gannon found that the use of the assignment feature as an operator caused more problems than its more traditional use as a statement.

The results of the early studies on innovative programming practices and language features that have been reviewed here as well as numerous others reported elsewhere (Shneiderman, 1980; Sheil, 1981; Du Boulay and O'Shea, 1981) indicate that innovations have had little measurable effect on programming performance. Reasons why these innovations failed are no doubt many. However, fundamental to their failure may have been a naive view of the nature of programming. Much research has been based on the view that programming consisted of a series of tasks and that these tasks could somehow be simplified by the use of flowcharts, better conditional statements, or some other innovative practice. Such a view fails to take into account the complex nature of the activity and the cognitive components that underlie programming skills. According to Sheil (1981),

> "Most psychological research on programming assumes that different programming tasks vary in difficulty and that the level of difficulty is an attribute of

> the task. The motivation for much of this work is the belief that the difficulty of large tasks is an aggregation of the difficulties of many component tasks . . . . Such assumptions are false for programming. They give no account of the most salient single fact about programming, which is that the difficulty of programming is a very nonlinear function of the size of the problem . . . . The simple aggregations of difficulty model provides no mechanism by which such nonlinearity could be generated" (p. 117).

Research on the Cognitive Components of Programming

Programming, like any other expert behavior, can be characterized by high level skills and complex cognitive structures. Recent research efforts that have examined the cognitive processes underlying computer programming are consistent with studies of other expert behaviors (Chase & Simon, 1973; Larkin et al., 1980). They indicate that an extensive amount of accessible knowledge is utilized in programing.

Brooks (1977) constructed an information-processing model of programming behavior based upon the transcribed protocols of experienced programmers engaged in various programming activities. Using the protocols, Brooks identified nearly 100 productions or rules utilized by programmers in code generation. From these findings,

Brooks predicted that the number of rules needed to represent all of an experienced programmer's knowledge must be on the order of "tens or hundreds of thousands".

According to Shneiderman (1976), expert programmers possess high-level semantic knowledge that enables them to organize information into meaningful "chunks". Shneiderman attributed the recall results of his shuffled program studies to these "chunking" abilities of the experienced programmers. His findings indicated that the more experienced programmers were able to recode and group language statements such that several could be remembered as a "chunk". Nonexperienced programmers, however, could not remember several statements as a single unit. Instead, they remembered individual statements as units and therefore could not recall as much of the code as the experienced programmers did. Adelson's (1981) findings regarding experience and automation of programming constructs supported Shneiderman's views.

Atwood and Ramsey (1978) conducted several exploratory investigations in an effort to collect information about the mental representations of computer programs. They hypothesized that debugging requires programmers to form hypotheses about the functions of individual program segments and the hierarchical relationships between these functions. The investigators predicted that it would take programmers longer to find bugs that were embedded deeper in the hierarchy than bugs which were located in surface levels of the hierarchy. Furthermore, they felt that the propositional hierarchy

formed during initial debugging attempts would be useful for subsequent attempts to locate different bugs in the same program and therefore decrease the time needed to locate the new bug.

Results of the Atwood and Ramsey study indicated that depth in the underlying propositional hierarchy and serial positioning (number of propositions preceding the one with the error) did appear to influence the participant's debugging performance. More specifically, serial positioning affected the time taken to locate an error while depth in the hierarchy affected the probability of finding the error. Since debug times consistently decreased for all students on the second program, Atwood and Ramsey proposed that the propositional macrostructures formed during the debugging of the first program were useful in debugging the second program. According to the investigators, these macrostructures served as sets of expectations about what the program should do and how the program would do it.

McKeithen, Reitman, Rueter, and Hirtle (1981) explored programmer macrostructures and chunking abilities in greater detail. Based upon the belief that programming statements consistently recalled as a group could be considered chunked together in memory, the investigators hypothesized that chunks recalled in close proximity might indicate the higher organizational components involved in programming. In a preliminary experiment, McKeithen et al. compared the differences in recall ability of expert and novice programmers who viewed either a coherent or a scrambled version of a computer program on five separate trials. Results on the scrambled version were

consistent with those of Shneiderman (1976) in that very few lines of the scrambled program were recalled by either the novices or the experienced programmers. For the coherent program, experienced programmers recalled significantly more lines of code than the non-experienced programmers and a close inspection of the recalled lines revealed characteristic patterns of recall. While inexperienced programmers recalled only short lines of code (BEGINs and ENDs), the more experienced programmers consistently recalled 1) the BEGIN and END statements, 2) the beginning statements of nested loops that read in matrix values and 3) parts of other loops that exchanged values and began output sequences.

A second study was then conducted to examine the differences in the way subjects organized their recall. Subjects of different programming skill levels were given a deck of cards containing unfamiliar ALGOL W reserved programming words and instructed to learn them. Later, they were asked to recall the words without the aid of the cards under both cued and non-cued testing situations. The recall orders from each subject were analyzed using an algorithm that searched all of a subject's recall strings for groups of items that consistently appeared contiguously, regardless of order. These groups were then arranged to form a tree with branches descending from the original string to mark the consistency and direction of the recalled groups. Results of the analyses indicated that the organization of the recall performances did differ according to programming experience. Mnemonic techniques such as grouping words according to

length, initial letter, and common language sequences were used by the non-experienced programmers. More meaningful organizations reflecting an understanding of programming constructs were used by the more experienced programmers.

Although the results of the McKeithen et al. studies do not prove that certain mental organizations produce programming expertise, they do suggest that subjects with an existing skill level seem to possess a particular common organization. Similarities between skill level and debugging strategies have also been noted. Results of a study conducted by Jeffries (1982) suggested that expert programmers performed much deeper readings of programs than did non-experienced programmers. These "deep" readings involved searching out the flow of control and consistently conducting global searches for the program's organizational structure. Novices, on the other hand, took "surface" readings of the program by conducting line by line searches. Jeffries attributed this difference not only to the experts ability to view chunks of code as instantiations of familiar programming tasks, but also to their ability to simulate computer operations in response to specific problem inputs.

Whereas the earlier studies on programming practices and language features were concerned with programming efficiency, the more recent studies have been concerned with clarifying the cognitive aspects underlying programing skill. Several studies have emphasized the complexity of the programming process by examining in greater detail nonexperienced and experienced programmer behavior. While these

studies may be somewhat useful in identifying some of the components of expert programming skill and providing goals for programming education, they do not directly address the issue of how learning experiences could and should be organized to help the novice acquire programming knowledge. This issue requires programming educators to take better advantage of recent developments in the fields of cognition and educational psychology if they wish to experience greater success in educating novice programmers.

Learning Theory and Instructional Methodology

Learning is not a passive activity. It requires the processing and assimilation of information if it is to be transferable and useful in problem-solving activities not explicitly taught. Bransford (1979) and Mayer (1981) define meaningful learning as integrated learning, a "process in which the learner connects new material with knowledge that already exists in memory" (p. 121). In similar fashion, Bruner (1966, 1973) has declared that organizing what is encountered is a necessary condition for transforming information for better use. Several advances in cognitive and educational psychology indicate that in order for meaningful learning to occur, instructional techniques must 1) take into account the current models and systems of knowledge possessed by learners at the time of instruction, and 2) tap into any prerequisite knowledge that might facilitate the assimilation of new information.

In his early classics, The Process of Education and Toward a Theory of Instruction, as well as more recent writings Bruner has maintained that a principle factor influencing meaningful learning is a student's existing cognitive structure. He advocates discovery in learning, maintaining that such an emphasis requires the learner to become a constructionist, to organize what is encountered in a manner designed to discover regularity and relatedness. Four general themes, concerning the nature and development of the student's cognitive processes, are emphasized in his teachings. These include 1) how a student's knowledge system might be made central to teaching, 2) learner readiness, 3) the nature of intuition and how educators should assist in its development, and 4) the desire to learn and how it could be stimulated. Nearly three decades of discovery strategy research, however, have been unable to produce consistent replicable results regarding the specific benefits of discovery in learning (Wittrock, 1966; Farhnam-Diggory, 1972).

Like Bruner, Ausubel (1968) also believes that the learner's existing cognitive structure influences subsequent learning. He has proposed the use of advance organizers to draw out the components of the learner's existing structures that could be particularly relevant to the situation at hand. By serving as both an anchoring and a linking mechanism, Ausubel claims that advance organizers would assist learners in making more useful and transferable connections between what the learner already knows and what is about to be learned. In short, advance organizers may facilitate meaningful learning by 1)

calling attention to and building upon knowledge already present in the learner's cognitive structure, 2) providing a skeleton upon which new information could be anchored, and 3) rendering unnecessary student learning by rote memorization.

A wide variety of studies have been conducted to investigate the effectiveness of advance organizers. Reviews of early efforts report a lack of consensus concerning their benefits (Barnes and Clawson, 1975; Hartley and Davies, 1976). More recent reviews (Ausubel, 1977; Mayer, 1979b) suggest that organizers may have the most effect in situations where the learner is inexperienced and unlikely to possess useful prerequisite information, or for tasks requiring creative solutions to solve unfamiliar problems (Mayer, 1981). Working with environments unfamiliar to the learner, Siegler and White found that the interaction between knowledge and learning is even more important than either Bruner or Ausubel indicated.

Siegler (1983a, 1983b) conducted a series of studies demonstrating that what a learner knows influences the conditions under which learning can occur. In each study, he utilized a rule-assessment approach in designing situations that would explicitly test a student's understanding of the concepts of time, speed, and velocity. First, errors patterns were studied and used to establish the rules that the students seemed to be applying in attempting to solve difficult problems. Next, the students were exposed to learning sessions consisting of problem sets that forced them to re-evaluate their current knowledge systems. Although many students were able to

alter their rules based upon these confrontations, Siegler noted that others could not. Subsequent analyses of videotapes revealed that while some students failed to consider some of the critical dimensions of the problem, others appeared to have encoded these dimensions in ways that were not useful for forming more adequate rules. Based upon this additional information, Siegler designed unique instructional tasks that would facilitate a more useful encoding of the critical dimensions of the problem. The performance of students receiving the encoding instruction indicated that they did adopt more advanced rules.

One of the educational implications of Siegler's work is the need for experiences that allow students to confront the inadequacies of their knowledge. White (1984) developed an environment that not only allowed her to study the knowledge systems of physics students, but also forced the students to examine these systems. Based upon previous research suggesting that students incorrectly extend beliefs about the motion of baseballs and cars to frictionless situations, White designed a sequence of computer games that required students to apply impulse forces to objects in order to alter their speed and direction of movement. Pretest results verified that students who had just studied Newton's laws possessed misconceptions and were unable to successfully solve problems focusing on the implications of the laws. Posttest results indicated that those students exposed to White's Newtonian microworld were able to answer more questions correctly than did students not exposed to the games. Furthermore, input records

collected by the computer as students played the games demonstrated that many of the students did indeed struggle between what their naive intuitions told them and what their physics knowledge told them.

Many other types of concrete models and manipulatives (objects such as bundles of sticks, coins, or blocks that allow students to make computational procedures more concrete) have also been effective in facilitating learning (Brownell and Moser, 1949; Branch, 1973; Resnick and Ford, 1980). West and Fensham (1976) found that concrete models used as advance organizers improved examination performance for low-ability physics students. Scandura and Wells (1967) used mathematical games as advance organizers to strengthen existing intuitions about mathematical groupings. Lesh (1976) found that the use of videotaped models as organizers for motion geometry produced higher achievement scores than treatments that did not use the organizers. Whether manipulatives or concrete models are used prior to instruction to establish frameworks for assimilating new knowledge or used after instruction to explore the hidden implications of abstract constructs, they do appear to greatly benefit instruction and learning.

The efforts of cognitive theorists and educators such as Ausubel, Bruner, Siegler, and White offer many valuable guidelines for the design of meaningful learning activities. Fundamental to these guidelines is the belief that existing knowledge plays an important role in future learning. Instructional tasks designed by White and Siegler were based upon what was known about the mental models and

current systems of knowledge possessed by beginning physics students and young children learning about time, speed, and velocity. The instructional methods that grew from this knowledge provided a framework that allowed learners to actively assimilate newly acquired knowledge in a more meaningful context. Similar approaches have recently been initiated for fostering the meaningful learning of programming.

New Directions for Programming Education

According to Pea and Kurland (1984) programming is an extremely complex intellectual activity. It involves a set of problem-solving activities that include 1) understanding the task the program is to accomplish, 2) planning a programming strategy that will accomplish the task, 3) implementing the plan via a programming language, and 4) debugging the plan and the code used to implement the plan. Studies examining experienced programmer behavior indicate that programmers apply these procedures recursively until their program works properly. In so doing, it has been suggested that expert programmers draw upon an extensive, highly organized body of knowledge consisting of syntactic and semantic pieces of programming information as well as sets of procedural skills or heuristics useful in applying this information. Of particular note is the fact that these cognitive qualities appear to be the consequence of an active constructive process that is able to capture the lessons of program writing experience rather than the effects of particular programming

practices, language features, or traditional methods of formal instruction.

Du Boulay and associates (Du Boulay and O'Shea, 1981; Du Boulay, O'Shea, and Monk, 1981) characterize the basic approaches toward programming instruction as "black box" and "glass box" approaches. In the "black box" approach, the operations of the computer remain hidden to the learner so that the learner has no idea of what goes on inside the computer. In contrast, the "glass box" approach provides a mechanism by which learners can study the relationship between programming statements and computer operations. With this in mind, Du Boulay and colleagues designed an interactive model of a simplified computer that permits learners to view selected parts and processes of a programming language in action. They have hypothesized that such a model would assist the user in developing intuitions about what transpires inside the computer which may, in turn, foster the meaningful learning of programming.

Mayer (1981) has also hypothesized that knowledge of how a computer works is necessary for the meaningful learning of programming. In a series of studies, students who were given a concrete model of a computer with explanations of its main components in terms of input/output windows, a memory scoreboard, and a program list and pointer arrow performed better on more difficult programming tasks than did students who were not exposed to the model. According to Mayer, allowing novices to "see the works" assisted their encoding process such that the information gained was encoded in a more

coherent and meaningful manner. Mayer's findings suggest that 1) static models of the computer can produce a framework for assimilating new information concerning computers and the programming process and 2) models presented before formal instruction on the syntax and semantics of a programming language are more effective than models presented after instruction.

If the view that programming consists of syntactic and semantic knowledge as well as procedural skills is correct, novice programmers are confronted with at least three major tasks. The novice must 1) learn the syntax of a language, 2) build up a store of coding segments that represent common programming subtasks, and 3) acquire the procedural skills necessary to be successful at programming. The learning of the syntax of a language is relatively trivial compared to acquiring semantic programming knowledge and developing procedural skills. While syntax may be satisfactorily learned by rote, automation of programming segments and procedural skills require more constructive, meaningful learning.

Implicit in Mayer's findings is an indication that carefully designed, simplified models of computer operations and concepts should be a productive and efficient way to encourage student acquisition of semantic programming knowledge. What could be an even more powerful learning environment, however, is an interactive model that would allow the student to actually confront some of his intuitive beliefs about programming and develop the procedural skills that seem so vital to the discipline. If used prior to formal instruction, such an

environment might prove even more useful in establishing foundations for formal instruction on programming. Furthermore, such an environment might be useful as a data acquisition system to further our understanding of the development of some of the cognitive processes underlying programming success. The task of designing, building and incorporating models of this nature into the instructional process is a major and important challenge. The task of evaluating their effectiveness in a deep and meaningful manner is an even greater challenge.

Measuring and Evaluating Fragile Novice Programming Skills

Measurement is a process that attempts to obtain a quantified representation of the degree to which a pupil reflects a particular trait (Ahmann and Glock, 1975). Paper-and-pencil tests are the most common measurement devices used to produce these quantitative representations, although there are many others (performance tests, rating and ranking scales, anecdotal records, questionnaires, etc.). In fact, their very value to the evaluation process is producing quantifiable evidence that, when considered alongside qualitative evidence and some other highly subjective impressions, contributes to the value judgements we call evaluations.

Quantifying with some type of precision the degree to which a student possesses a trait is much easier for concepts in which forums of agreement exist. Specifying the criteria for goal-attainment requires not only advance knowledge of how one can achieve the goal,

but also agreement concerning the criteria used to determine success in goal-attainment. Regarding programming performance, such a forum of agreement exists only in terms of "Does the program work?" and perhaps "Does the program work efficiently?". Until more is known about the discipline of programming, more profound forums of agreement will probably not be forthcoming.

The cognitive components and processes underlying novice programming behavior are probably a fragile and unreliable set of knowledge structures. Even though programming is founded on a common object, the computer, its acquisition and meaning may be unique to each individual student based upon her previous experiences. For this reason, the use of paper-and-pencil tests may be inadequate. Even an analysis of a "one-shot" attempt to write a program or a segment of a program has not proven to be a very productive measure of the processes involved in programming. What appears to be needed is a means of soliciting and collecting entire programming sessions which can be analyzed both individually and collectively. The student should be permitted to present a solution, receive feedback which is meaningful within the programming environment, and revise the solution until satisfaction or frustration is reached. Data collected as a student engages in programming should provide information concerning the novice's intermediate thought processes and thus more accurately reflect programming knowledge than would an answer to a written test question.

The methodology of studying programming behavior by collecting every syntactically correct version of a program is not new. Online records of programming efforts have been extensively utilized in studying compiler error messages in an effort to generate more meaningful messages (Shneiderman, 1980). However, the literature reports fewer instances of using this methodology to study the more intellectually taxing aspects of programming.

One investigation that did attempt to study the thought processes of programmers by collecting all syntactically correct versions of programs was conducted by Bonar, Ehrlich, Soloway, and Rubin (1982). Using a computer program called the BUG FINDER, the investigators located the semantic and pragmatic errors in each program version. These errors were then used to develop a catalog of programming errors and to identify 1) patterns of errors over an entire semester, 2) stereotypic correction methods employed by the students and 3) individual programming styles. Although the investigators caution that the implementation of this methodology required significant resources, they were encouraged by their success in studying psychological aspects of programming that could not be studied via written solutions to test questions.

Summary of Literature Review

Numerous directions have been taken in programming research in an effort to learn more about the nature of programming and the cognitive processes underlying programming success. These directions have

included studying the effects of innovative practices and language features on programming performance as well as investigating aspects of expert programming behavior. While these efforts have been useful in identifying some of the features of programming skill and can serve as goals for programming education, they do not directly address two important instructional issues, namely how learning experiences should be organized to help the novice acquire programming knowledge, and specifically what novices must learn in order to achieve programming success.

Recent advances in the fields of cognitive psychology have indicated that what is already known may influence subsequent learning. Instructional techniques that build upon this aspect of learning have been successful in facilitating more meaningful learning of cognitively demanding material. Three of these techniques are 1) using manipulative objects to make abstract concepts more concrete, 2) using advance organizers to prepare the learner for subsequent information and 3) creating models of reality that force the learner to confront incorrect intuitions. The computer's ability to model itself provides a unique environment for implementing similar techniques that could facilitate learning, specifically, the learning of programming.

The development of computer environments designed to facilitate the acquisition of programming knowledge is a challenging task. But an even greater challenge is evaluating the effectiveness of these environments in a meaningful manner. Evaluation must be based upon an

analysis of subsequent programming performance. Since programming normally takes place in an interactive environment, written solutions to test questions may not adequately measure the subtle effects of such environments on programming performance. A suggested alternative is to allow novices to program solutions to test problems. Besides reflecting a more normal environment for programming, this methodology allows data to be collected that may be useful not only in clarifying knowledge exhibited by a written solution, but also in studying the novice's intermediate thought processes. This approach does, however, require significant resources.

CHAPTER III: METHODS AND PROCEDURES

This study resulted from a need to resolve the discrepancies between the theoretical instructional potential of the computer and the multitude of applications which have been produced. It was developed in full recognition of probable differences between simulations and drill and practice applications, testing for process and testing for recall, and learning computer programming and learning a subject like spelling. The study was intended to be but one small step in determining the ultimate role of computers in the learning process.

Because the study delves into the emerging area of cognitive psychology, employs newly developed instructional software packages, investigates the relatively new subject of computer programming and relies on the results of untested evaluative instruments, it can only be viewed as descriptive research. For this reason, the study is based on research questions rather than hypotheses. In addition, the methodology that is employed was designed to reveal characteristic behaviors. The intent of the study is to provide the ground work on which more formal research can be based.

Subjects

The participants in the study were those students who had enrolled in an introductory computer applications course offered through the Industrial Education and Technology department at Iowa State University in the fall of 1985. The introductory portion of

this course was designed to familiarize students with the Pascal programming language. The latter portion required the use of Pascal in developing programs for industrial applications.

A ten-item questionnaire was developed to collect descriptive data concerning each participant's educational background and previous programming experience. Information requested from the students included age and sex, year in college, previous computing experience, computer ownership, mathematics background, college grade point average, course expectations, and reasons for enrolling in the course. A copy of the questionnaire is provided in Appendix A.

Results of the questionnaire revealed that the average age of the participants was 21.9 years with a range of 20 to 31 years. Thirty-five of the thirty-six participants were male industrial education and technology majors. Four students were currently classified as sophomores, sixteen were juniors, thirteen were seniors, and three were graduate students. Twenty-six students stated that they were taking the course because it was required, although several also stated that learning to program would benefit their future careers.

The computing experiences of the students varied extensively. Slightly more than one-third (13) of the students stated on the questionnaire that they had taken some type of high school computer literacy or programming course. Twenty students stated that they had previously taken programming or computer literacy courses in college prior to enrolling in the present course. Of the students who indicated that they had done programming, six had written BASIC or

LOGO programs, ten had written FORTRAN programs, and six had written either Pascal, COBOL, or PL/1 programs. Fourteen students indicated that they had never programmed a computer. Several of the participants (14) also reported having used computers for word processing, statistical analysis, and engineering design (CAD). Nine students stated that they owned a microcomputer.

Unlike the variety of computer experiences possessed by the participants, most of the students had taken similar mathematics courses. Thirty-five of the thirty-six students took at least three high school mathematics courses (Algebra I, Algebra II, and Geometry). Twenty-six students indicated that they had also studied either trigonometry and/or calculus in high school. In college, a majority of the students continued to study mathematics. Eight students reported that they had taken a refresher course in algebra and/or trigonometry. Twenty-seven students indicated that they had taken one semester of calculus in college and three of these students stated that they took additional calculus courses.

The majority of the participants in the study (21) had a college grade point average between 2.00 and 2.49 on a 4-point scale. Another nine students had averages between 2.50 and 2.99. Five students had averages above 3.00 and one student had an average below 2.00. Thirty-two students, however, expected to earn an "above average" grade in the course in which this study was conducted; twenty-four expected to receive at least a "B" and eight students expected to earn an "A".

In summary, the majority of the participants in the study were male, junior or senior industrial education and technology majors. In high school these students had taken approximately four years of mathematics that included two algebra courses, a geometry course, and an advanced mathematics course. In college, the students took at least one semester of calculus. Computer experiences ranged from none to extensive, with fourteen of the students never having programmed a computer prior to enrolling in the course. The college grade point average for the majority of the students was between 2.00 and 3.00. Nearly all of the students expected to receive an above average grade in the present course.

Description of Computer-Based Materials

Four different types of computer programs were used to provide instructional activities and collect pertinent data for the study. These programs will henceforth be referred to as MEMOPS, CHALLENGER, PASTUT, and MINIPAS. All four of these lessons were programmed in the Digital Authoring Language and were made available to students through the Courseware Authoring System on one of the Iowa State University VAX Clusters.

MEMOPS

MEMOPS was a lesson designed to give novice programmers an early opportunity to experience elementary programming tasks in what appears to the novice as a "non-programming" set of activities. Using a "move" (MOVE X[1] TO Z) and/or a "compare" (COMPARE X[1] WITH X[2])

instruction, students were required to work through a set of five manipulative exercises designed to familiarize them with the following concepts: 1) writing to memory is a destructive operation, 2) reading from memory is a copy operation, 3) temporary storage is needed to preserve information, 4) computers linearly process the instructions of a program one at a time, 5) the form of a programming statement must adhere to the syntax rules stipulated by the compiler in use, and 6) array cells must be addressed using an index.

The five manipulative activities presented in MEMOPS included moving the smallest value stored in an array of elements to another memory location (Figure 1), moving the largest value in an array of elements to another location, swapping the values stored in two different memory locations, sorting the values in ascending order (Figure 2), and sorting the values in descending order. Students were required to perform all five activities using both a visible model of memory (Figure 2) and a non-visible model of memory (Figure 3). Following the successful completion of all activities using both the visible and non-visible models, the student was asked to summarize what was learned by answering two multiple-choice questions (Figures 4 and 5).

For the manipulative activities, the MEMOPS program provided two types of feedback. If the instruction that the student typed was syntactically correct, the program performed the operation irrespective of whether it contributed to the solution of the problem. If the instruction was not syntactically correct, the program informed

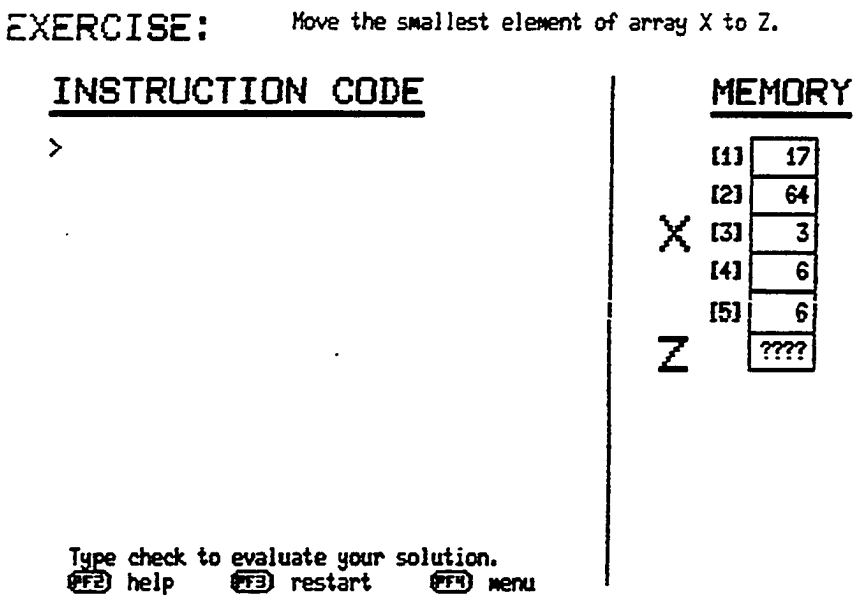


FIGURE 1. Visible memory model for MEMOPS Task 1 (moving the smallest value to Z)

the student of the error and provided examples of correct formats and types of operations that could be performed. At no time during any of the five activities did the lesson offer "cook-book" instructions on what to enter to solve a given problem. In order to determine if an activity had been successfully completed, the student had to request that the computer check the final status of the values in the model. Success was based only upon the status of the model, not upon the student's efficiency in attaining that state. The student was allowed to restore the original values and restart the activity at any time.

In addition to providing the manipulative activities, MEMOPS was programmed to record the student's efforts in attempting each activity. Individual files were maintained containing all

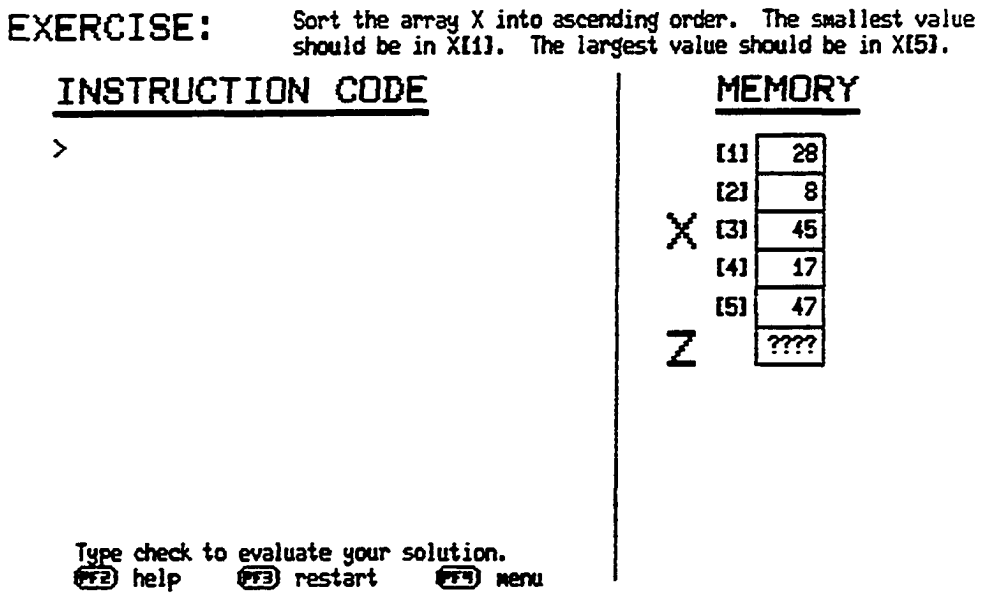


FIGURE 2. Visible memory model for MEMOPS Task 4 (sorting the values of an array in ascending order)

syntactically correct operations entered by the student. The initial and final status of the model were also recorded so that the researcher could reconstruct each student's MEMOPS session at a later date.

CHALLENGER

The CHALLENGER program was used to provide a placebo computer activity for those students not assigned to the MEMOPS treatment group. CHALLENGER was a two-dimensional puzzle that looked like one face of a Rubik's cube. It was a 3 X 3 matrix of squares each of which could be either white or green (Figure 6). The color of a specific set of squares could be changed by moving a blinking cursor

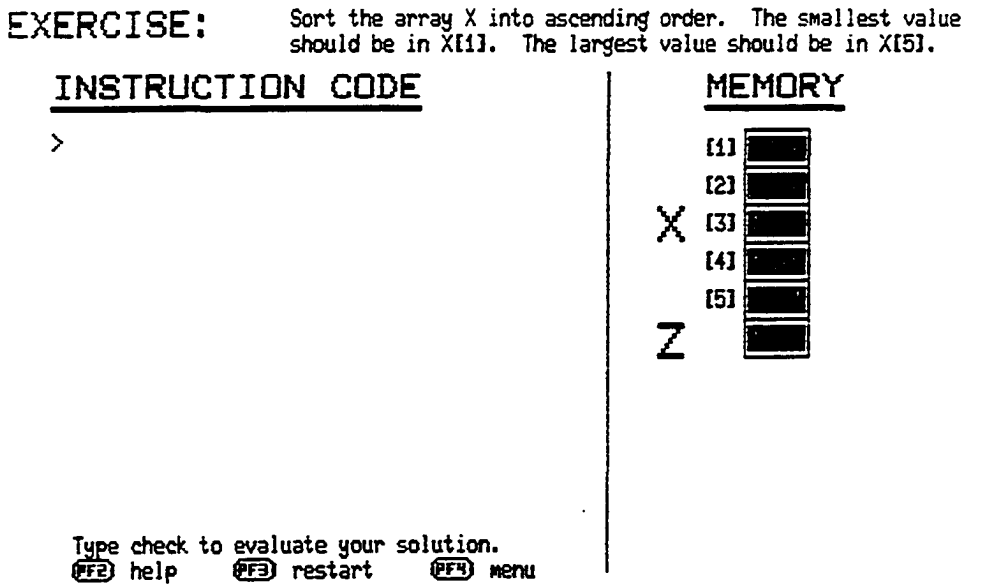


FIGURE 3. Hidden memory model for MEMOPS Task 8 (sorting the values of an array in ascending order)

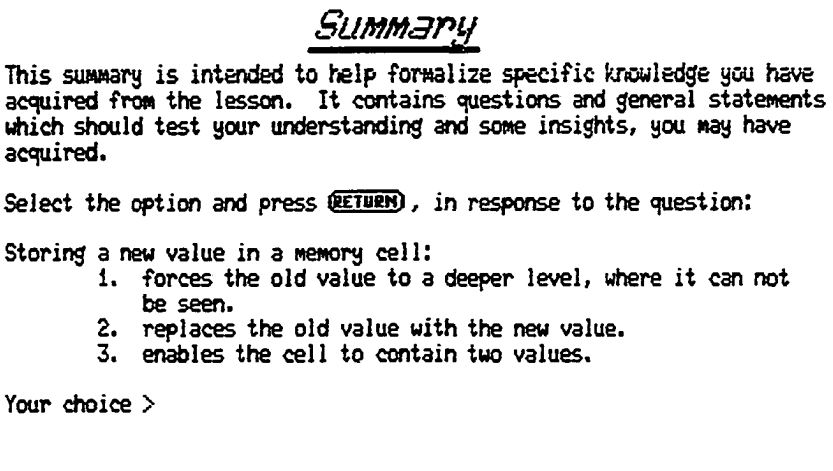


FIGURE 4. First MEMOPS summary question

Summary

This summary is intended to help formalize specific knowledge you have acquired from the lesson. It contains questions and general statements which should test your understanding and some insights, you may have acquired.

Select the option and press (RETURN), in response to the question:

In swaping the values of two memory cells A and B:

1. cell B should be loaded with the value of cell A and then cell A should be loaded with the value of cell B.
2. a third cell would be needed.
3. the two values would be exchanged simultaneously.

Your choice >

FIGURE 5. Second MEMOPS summary question

to a particular square and pressing the <RETURN> key. Due to the symmetry of the matrix only three distinct moves were possible. By placing the indicator in the middle square along one side, all three squares along that side would change color; the white ones became green and the green ones became white. If the indicator was placed in a corner square, that square and the three surrounding squares changed color. If it was placed in the center square, that square and the four center squares on each side changed color.

Using the three moves, the student's goal was to change the pattern from an arbitrary arrangement of white and green squares to the final matrix containing a single white cell surrounded by eight green cells. Since the blinking cursor could only be placed on a green cell, the goal was not easily attainable. Unlike the other

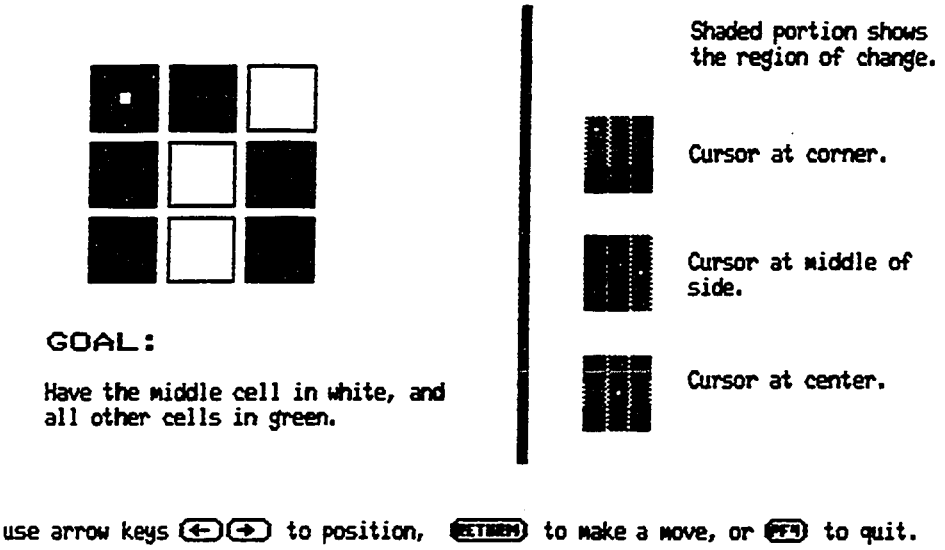


FIGURE 6. CHALLENGER display

computer lessons described in this section, the CHALLENGER lesson was not intended to introduce students to any programming concepts. Rather, it was used to equalize the computer-operating experiences between the two experimental groups.

PASTUT

The PASTUT lesson was a tutorial type of computer lesson designed to reinforce information presented in lecture regarding the syntax and semantics of Pascal statements. This lesson consisted of brief narratives followed by short-answer questions. The purpose of the program was to ensure that all of the students were familiar with and had a working knowledge of the Pascal instructions needed to complete the assigned programming tasks. The instructions covered in this

tutorial were the assignment, IF, PROGRAM, VAR, READLN, and WRITELN statements. In addition, the tutorial covered the function of semicolons as statement terminators and overall Pascal program structure. In order to successfully complete the lesson, students were required to generate several commands of each type presented.

MINIPAS

MINIPAS was a computer program that created a simplified environment for running Pascal programs. Included in MINIPAS was an editor for entering Pascal statements, a compiler that would perform the typical syntax checks done by any standard compiler, and a visible memory window that allowed the user to view the values of variables during program execution (Figure 7). A tracing feature permitted the user to execute a program one statement at a time in order to observe the action taken by the computer in response to individual Pascal statements. By tracing, the user could observe the function of an individual statement as well as the collective action of a group of statements. Thus, MINIPAS was designed to facilitate the learning of the language and the debugging of algorithms.

In addition to serving as a simplified programming environment, MINIPAS stored all versions of all programs that each student successfully compiled. The successive versions of programs and additional data such as compilation errors and length of time in MINIPAS were used in developing the performance protocols that documented programming behavior.

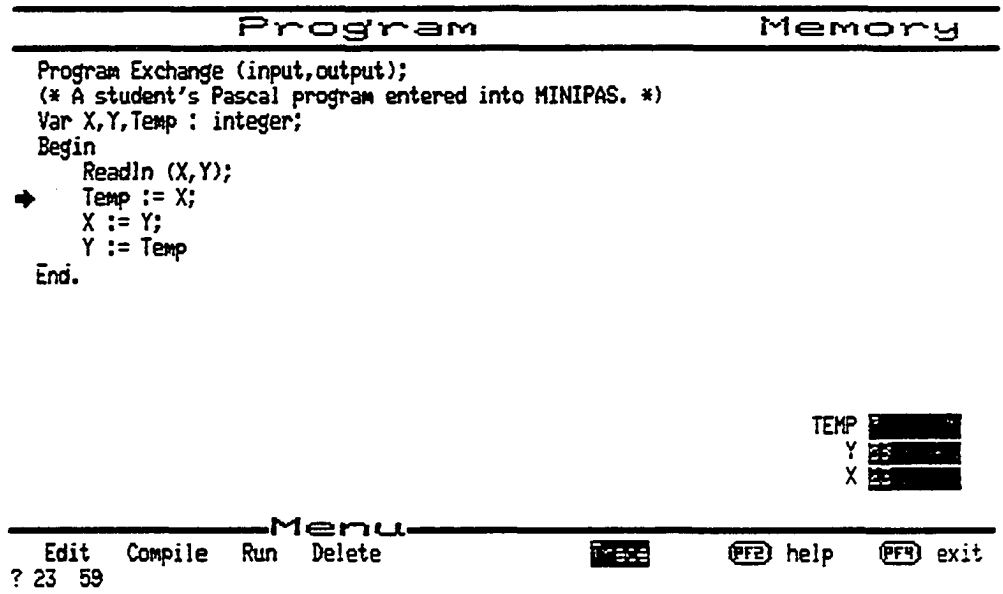


FIGURE 7. MINIPAS display

Instruments

Three types of instruments were used to collect the data pertinent to the study. Besides the background questionnaire, two paper-and-pencil tests were administered and the online programming actions for both the MEMOPS and MINIPAS lessons were recorded.

The two sets of paper-and-pencil tests were designed to measure a student's knowledge regarding memory operations, syntax, and ability to generate Pascal code to perform selected programming tasks. On the first test, the students were instructed to generate Pascal code for two programming problems. The first problem was to write a program that would swap the values of two variables. The second problem consisted of writing a program that would request the user to input

three numbers (in any order) and then sort the numbers from smallest to largest. Refer to Appendix C for a copy of this test.

The second paper-and-pencil test measured more complex programming concepts including the implementation of array data structures and looping constructs. On this test, students were asked to identify illegal array declarations and run-time errors caused by inappropriate index values, hand-execute two Pascal programs and state the final values that would be stored in the arrays declared in each program, and generate Pascal code that would perform selected array manipulation tasks. These array manipulation tasks included writing Pascal programs that would compare the contents of two arrays, reverse the order of the values stored in an array, and sort the values of an array in ascending order. Refer to Appendix D for a copy of this test.

Programming is usually performed in an environment that is interactive and provides the programmer with feedback vital to the programming process. As discussed in the literature review, the normal paper-and-pencil testing environment may be of questionable worth in measuring certain programming skills. More specifically, normal testing procedures would appear to be inappropriate for collecting information concerning the interactive nature of programming and for measuring the student's ability to utilize the feedback provided by the computer in developing programming solutions. In order to study this aspect of programming behavior, students were permitted to enter their written solutions to three of the test

questions into the computer and run and modify these solutions until they performed the assigned programming tasks. These runs were collected by the computer and were later used in developing individual protocols of programming behavior.

Research Procedure

The study was conducted as an integral part of an industrial educational and technology course. The sixteen week course schedule included word processing for the first two and one-half weeks and programming in the Pascal language for the remainder of the course. The experiment was conducted in two parts. The first part took place during the third through the fifth weeks of the semester when the students were learning the general characteristics of languages, compilers and programming practices. During this period the PROGRAM, VAR, IF and assignment statements were presented. The second part of the study took place during weeks eleven through thirteen when FOR statements and array declaration statements were covered. Instruction on using a word processor preceded the first part of the study. Detailed instruction on IF statements, graphics capabilities of the Apple IIe microcomputer, and a program to simulate the actions of a solar collector preceded the second part of the study.

The research design for the study consisted of a post-test quasi-experimental design using a matching strategy for assigning participants to treatments. Based upon background information collected using the questionnaire, students were assigned to matched

pairs. The criteria for matching was based primarily upon previous programming experience, mathematical background, college grade point, and grade expectation. Once the matched pairs had been made, one member of each pair was randomly assigned to the treatment group and the other member of the pair was assigned to the control group. The instructional treatments received by the two experimental groups differed only in initial exposure to either the MEMOPS lesson or the CHALLENGER lesson. Subsequent lecture presentations, programming assignments, and posttest activities were identical for both groups (Figure 8).

Group			Weeks 3 - 5			Weeks 11 - 13	
Treatment	Q	R	MEMOPS	I1	P1	I2	P2
Control	Q	R	CHALLENGER	I1	P1	I2	P2

Q - Questionnaire administered
R - Randomly assigned students to groups using matching strategy
I1 - Classroom instruction, lab exercises and programming assignments covering simple memory operations, Pascal declaration statements, assignment statements, and IF statements
P1 - Posttest 1 on memory operations, swapping, sorting three numbers
I2 - Classroom instruction, lab exercises and programming assignments covering looping constructs, array declarations and implementation
P2 - Posttest 2 on array declarations and implementation, sorting

FIGURE 8. Sequence of instructional events

The background questionnaire was administered to all participants during the second class meeting and the experimental groups were formed. Instruction on the use of the MINIPAS editor was given and

students worked through the lesson's "Learn to Edit" section. In the next class period, students were exposed to either the MEMOPS or CHALLENGER lessons. Both lessons required approximately an hour to an hour and a half to complete. Assigned seats were used to prevent students in one treatment group from viewing the screen displays of the other group's lesson.

The next three class periods were spent introducing students to simple memory operations, Pascal declaration statements, assignment statements and simple IF..THEN..ELSE structures. Students worked through the corresponding sections of the PASTUT lesson and wrote short programs requiring the use of variable declarations and assignment statements.

The first posttest, consisting of two programming problems, was then administered. Students were asked to generate Pascal code that would 1) interchange the values of two variables and 2) order three numbers that were entered at random. After writing a programming solution down on paper, each student was allowed to enter the solution into the MINIPAS program and test it to see if it performed the assigned task. Students were allowed to modify their solutions using the MINIPAS lesson until they were satisfied that their code successfully performed the assigned task.

The second part of the study was conducted during the latter part of the semester. Students were given instruction on the Pascal FOR loop and on array declarations as well as the use of loops and arrays in programs. Five class periods were spent working through examples

of programs that implemented array data structures. Several short programming assignments were given requiring the students to utilize arrays and FOR loops.

The second posttest measuring the student's understanding of FOR loops and array implementation followed. On the first part of the test, students answered questions concerning array declarations, run-time errors caused by inappropriate index values, and the values stored in the cells of an array after program execution. The final three questions of the test required the students to generate Pascal code that would compare the contents of two arrays, reverse the order of values stored in a single array, and sort the values of an array in ascending order. Students were allowed to input their solutions to the sort problem into MINIPAS and run and modify their solutions until they were satisfied with their code.

Methods of Analysis

To ensure that the treatment groups did not significantly differ on the matching criteria used in assigning students to experimental groups, chi-square tests of independence were performed. None of the chi-square values were found to be statistically significant. (Refer to Appendix Tables A-1 through A-7 for frequency counts and chi-square values.) Based upon this information, it was concluded that the backgrounds of the participants assigned to each of the two treatment groups did not differ significantly.

The standard criteria for evaluating a programming solution is primarily based upon whether the solution, if entered into a computer, will execute properly and perform the specified task. Partial credit is often given for syntactically incorrect solutions that exhibit "desirable" implementation features, or for solutions that correctly process only a subset of instances in the problem's domain. This technique, of assigning a numerical score to a programming solution, was determined to be an inadequate measure of programming performance for this investigation because it could potentially conceal features of a student's solution that might reflect subtle but important aspects of programming knowledge.

Therefore, the primary evaluation technique used in this study consisted of documenting the characteristic features of a student's initial and final solution efforts in attempting to generate a computer program for a given task. Analyses that compared the number of treatment students exhibiting a feature to the number of control students exhibiting the feature were then performed. Again, chi-square tests of independence were used to perform these group comparisons. Additional comparisons of group performance using t-tests were performed on the number of initial and total compilations recorded for a programming session as well as the number of unique versions of a program.

Although posttest scores were, in general, not considered to be an adequate measure of programming performance in this investigation, scores were used as a supplementary measure of performance on the

second posttest. The scoring procedure for this posttest can be found in Appendix D. T-tests were used to compare the posttest scores of the treatment students to the posttest scores of the control students.

CHAPTER IV: FINDINGS

The goals of this study were; 1) to document novice programming behavior in an attempt to determine the processes of learning to program, 2) to evaluate the effects on student learning of a manipulative computer model used prior to formal instruction on computer programming, and 3) to evaluate the use of protocols as tools in studying programming behavior. In this chapter a documentation of the behavior of beginning programmers over a series of programming tasks is presented. In addition, the behavior of students who did experience a manipulative model prior to attempting the programming tasks is contrasted with those who did not.

The chapter is subdivided into three major sections that describe in detail the behaviors documented in the protocols for the MEMOPS activities and the two posttest programming problems. Within a section, the behaviors for each activity or problem, including a comparison of experienced and non-experienced programmers, are presented first. For the sections describing the two posttests, the behaviors of students who used MEMOPS is then contrasted with those who did not. At the end of each section is a summary of that section.

Throughout this chapter, identifiers that denote the students whose solution efforts displayed the characteristic under discussion are listed. Identifiers for the students who received the MEMOPS treatment begin with the letter "T" and those of the students in the control group begin with a "C". The inclusion of these identifiers provides a visual comparison of the treatment and control students and

permits the interested reader to follow the key behaviors of individual students.

MEMOPS Protocol Findings

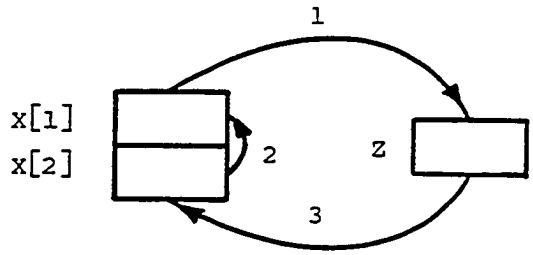
The MEMOPS lesson consisted of two subsets of tasks. In the first subset of tasks, the contents of a five-element array (X) and a simple integer variable (Z) were visible to the student. In the second subset, the values of these memory locations were hidden from view. In this section, the behaviors that were documented for the two subsets of tasks will be discussed independently. Individual solution features for the two subsets are reported in Appendix Tables B-1 and B-2.

Student performance on the visible memory tasks

The first two MEMOPS tasks were designed to provide practice with the syntax of the MOVE command and to ensure that students understood its primary function. The first exercise requested that the student move the smallest value in the array to a specified memory location and the second exercise asked the student to move the largest value. Completing these tasks required neither complex logic nor sequencing decisions since the student could visually determine which values needed to be moved. Therefore, detailed protocol characterizations were not made.

On the third MEMOPS task students were asked to interchange the values of two memory locations. This activity required the student to possess and use the knowledge that storing one value in a memory cell

destroys any previous value that may be stored there. It also required the sequencing of instructions. In order to complete this activity, the student needed to first move the value of one of the cells into a third unused memory location. After this had been done, the value of the second cell could be moved into the first cell. The final step was to move the value in the third memory location into the second cell. Figure 9 graphically displays an efficient solution to this swapping problem as well as a possible series of MOVE instructions that would successfully solve the problem if issued separately and in the given sequence.



1. Move X[1] to Z
2. Move X[2] to X[1]
3. Move Z to X[2]

FIGURE 9. Solution to the MEMOPS swapping task

Two distinguishing characteristics related to previous programming experience were discovered by studying performance on the swapping task. These characteristics were method of moving values and choice of auxiliary storage location. For each characteristic, performance patterns of students with little or no previous

programming experience differed from those of students with more extensive programming experience.

The first distinguishing performance characteristic between experience groups was revealed by analyzing the error patterns of the inexperienced programmers. Of the ten students who had little or no programming experience, nine tried to interchange the values of the two memory cells by moving the value of the first cell into the second (MOVE X[1] TO X[2]), and then moving the value of the second cell into the first cell (MOVE X[2] TO X[1]). Of the eight students who had previously programmed in either FORTRAN or Pascal, three students (T09, T10, T15) also started to employ this solution. Two of these students perceived that an original value was lost after issuing only one MOVE instruction. Only one experienced student (T15) completed this solution attempt. These findings suggest that the knowledge that a MOVE instruction destroys the original contents of a memory cell, or the ability to use this knowledge, appeared to be lacking in beginning programmers and even caused some difficulties for those with previous programming experience.

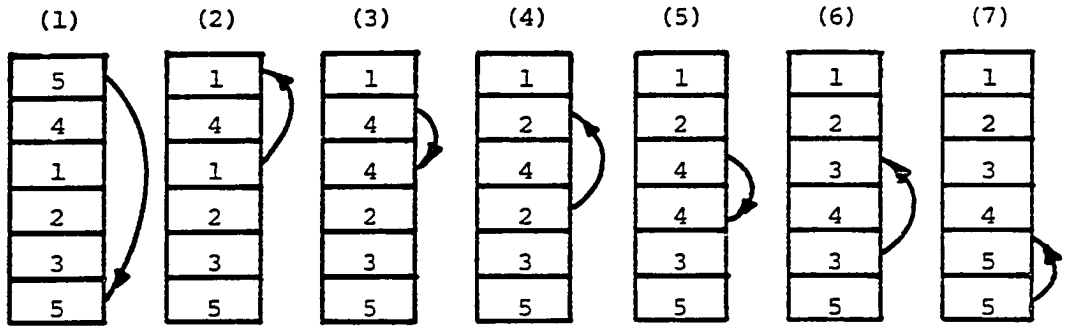
The second documented characteristic for the swap task was the student's selection of a memory location to use in preserving an original value. Eight students stored a value in the "Z" location and another eight students stored a value in one of the array's unused cells (X[3]). An examination of the data revealed that students with little or no previous programming experience used the "Z" location to store a value. Students with more extensive programming experience

used the free element closest to the cells containing the values that were to be swapped. Thus, the experienced students demonstrated facility in generalizing the form of the statement while the beginning students restricted their use to the form they had previously used.

In the fourth and fifth MEMOPS tasks the students rearranged the given values of an array and put them into either ascending or descending order. Solving the two visual sorting tasks required the students to develop a procedural plan or algorithm for sorting and then translate the plan into a sequence of move instructions. Characterizations of algorithm development and implementation were made, beginning with the original cell values given to each student and then retracing the series of move instructions issued by the student in solving each sorting problem. Two features of the implemented algorithms were documented for the visible sorts.

The first feature documented in the sorting protocols was the order in which the student attempted to fill the cells of the array. Most students tried to fill the array in a sequential manner. These students issued MOVE instructions that placed the smallest (largest) value into its proper cell first. Instructions placing the second smallest (largest) value into its proper location were issued next. These were followed by instructions that permitted the middle value to be moved into its proper location and so forth. An example of a series of MOVE commands that illustrate sequential filling is shown in Figure 10. No patterns concerning filling the array in a sequential manner and previous programming experience could be detected since

nearly all of the students used sequential filling to complete both visible sorting tasks.



(1) Move X[1] to Z
(2) Move X[3] to X[1]
(3) Move X[2] to X[3]
(4) Move X[4] to X[2]
(5) Move X[3] to X[4]
(6) Move X[5] to X[3]
(7) Move Z to X[5]

FIGURE 10. A series of MOVE instructions illustrating sequential filling of an array

The second feature recorded for the visible sorting performances identified the manner in which the student interchanged element values. For example, the student may have been given a 3-cell problem with a value of 3 stored in the first cell, 1 in the second cell, and 2 in the third cell. This problem could be solved in one of two manners. One method would be to employ two 2-cell swaps and interchange cells one and two and then interchange cells two and three. The other method would be to move the original value from the first cell to a free memory location, move the value of the second

cell into the first cell, move the value of the third cell into the second cell, and then move the value in the free location into the third cell. Figure 11 graphically illustrates these two swapping techniques for a 3-cell problem.

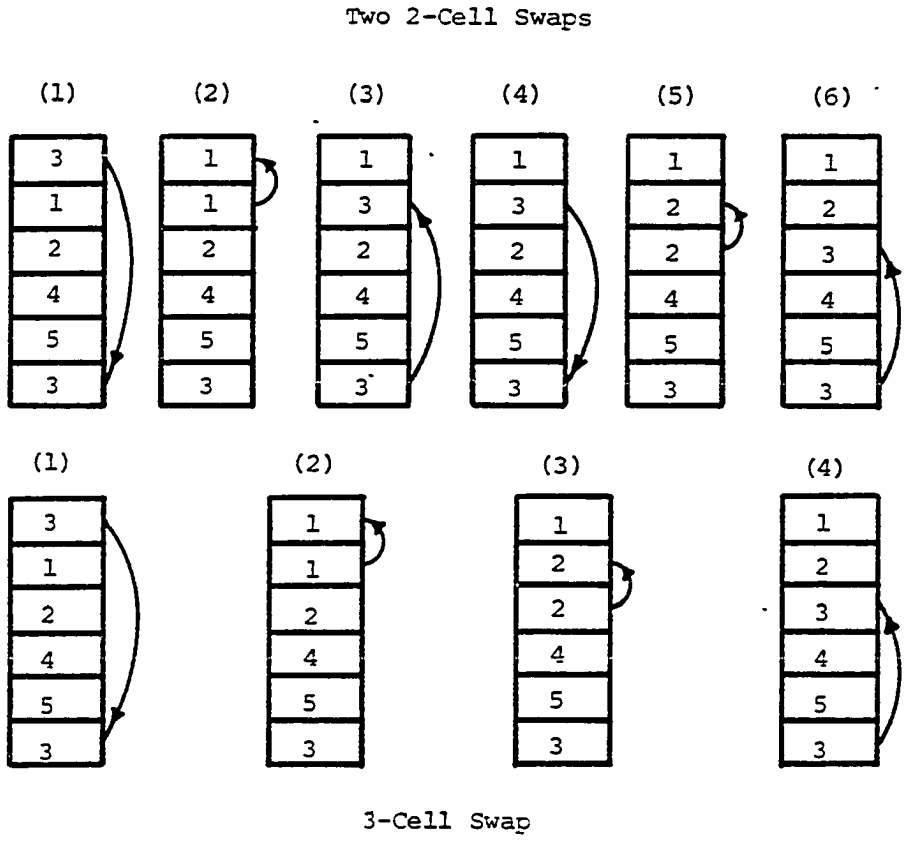


FIGURE 11. Two swapping techniques for a 3-cell sort problem

Nearly all of the students implemented a swapping method that minimized the number of MOVES needed to complete the reordering task. Although this approach is very easy for a human processor, it would be

very difficult to implement on a computer. No patterns concerning previous programming experience and selection of a particular swapping technique could be detected from examining the swapping histories.

Student performance on the hidden memory tasks

The cell values for the last four MEMOPS tasks (Tasks 6-9) were not visible, thus these tasks are referred to as the hidden memory tasks. By issuing a COMPARE instruction (i.e., COMPARE X[1] TO X[2]), the student could learn the relationship between two hidden values (i.e., X[1] is larger than X[2]). The MOVE instruction was still available for use in moving values from cell to cell.

MEMOPS tasks six and seven requested that the students move the smallest and largest values stored in the array to the Z location. These two tasks were designed to familiarize the students with the COMPARE instruction and to force the students to use it in conjunction with the MOVE instruction in order to locate and move values. The predominant feature noted for these tasks was the use of a "keeps best" algorithm to determine which element contained the value that should be moved to Z. In this algorithm, the smaller (larger) value is always "kept" and compared to the next element's value. Figure 12 illustrates a "keeps best" algorithm for locating the smallest value stored in an array.

Most of the students, regardless of prior programming experience, utilized the "keeps best" algorithm in solving the hidden largest and hidden smallest tasks. Only four students failed to use this algorithm. The algorithms of these students were characterized by a

A student's MEMOPS instructions	MEMOPS feedback
compare X[1] to X[2]	X[1] is greater than X[2]
compare X[2] to X[3]	X[2] is less than X[3]
compare X[2] to X[4]	X[2] is less than X[4]
compare X[2] to X[5]	X[2] is greater than X[5]
move X[5] to Z	(value in X[5] moved to Z)

FIGURE 12. A sequence of MOVES illustrating a "keeps best" algorithm for locating the smallest value stored in an array

failure to efficiently utilize the comparison feedback. This resulted in excessive comparisons between elements. Of the four students who failed to use the "keeps best" algorithm two (T01, T05) had never programmed before, one (T10) had programmed in BASIC, and one (T11) had programmed in FORTRAN.

The final two hidden memory tasks asked students to put the values of an array's cells into ascending (MEMOPS Task 8) and descending order (MEMOPS Task 9). In solving the hidden sorting tasks, algorithm development and implementation became even more complicated. The values of the arrays were unknown and the student could not immediately determine the final destination of each value by visual inspection. In addition to the sequential fill used for the visible sorts, three other solution features were used to document the processes the student used in determining the order of the hidden values.

The first noted feature of the student's algorithm for determining the order of the values was the selection of elements for comparison. Several students used a "keeps best" method to compare values. For example, the four COMPARE instructions in 12 might be issued to determine which cell contained the smallest value. Next, a series of COMPARE instructions might be issued to locate the second smallest value. This process of "keeping" the element containing the smallest value for use in the next comparison would continue until the order of all of the values was known. Eight students (T02, T06, T08, T09, T13, T16, T17, T18), six of whom had previously programmed, employed the "keeps best" technique on both the ascending and descending sorts.

The second distinguishing feature for determining the order of the hidden values was the manner in which students sequenced their MOVE and COMPARE instructions. Five students (T08, T09, T13, T17, T18) with prior programming experience combined the "keeps best" algorithm for locating values with the sequential fill for choosing the cell to be filled. These students issued a sufficient number of COMPARE instructions to determine the value that needed to reside in a particular cell, and then immediately moved that value. This process of comparing and moving values was repeated until all values were properly ordered. The remaining thirteen students compared values to determine all of the relationships and then issued MOVES until the values were ordered.

The final feature documented for the hidden sorts was whether all of the necessary comparisons had been made for determining the order of the values. Students who made all of the necessary comparisons were judged as having attained closure for the problem since they knew the relative order of all values. All but four students (T01, T07, T10, T11) attained closure on the two hidden sorting tasks. In this case, there was not an apparent pattern between closure and previous programming experience.

Upon finishing the MEMOPS activities, students were asked to complete two multiple-choice summary statements. The first statement summarized the action that occurred when a new value was stored in a memory cell. The second statement summarized how the values of two memory cells could be interchanged. Only four students (T03, T05, T15, T18) selected an incorrect completion option for the first statement. All of the students correctly completed the second summary statement.

The final feature documented for the MEMOPS tasks was the number of times tasks were restarted. The average number of total restarts for students with little or no previous programming experience was 4.00 restarts with a standard deviation of 3.23 restarts. The average number of restarts for students with more extensive programming experience was 1.50 with a standard deviation of 1.69. This difference in number of restarts indicated that the MEMOPS tasks provided exposure to programming concepts with which the inexperienced

students were unfamiliar. Thus, the treatment did, in fact, have the potential for affecting the student's future programming behavior.

Summary of MEMOPS findings

The MEMOPS simulation presented a series of tasks intended to move beginning programmers from the robust realm of human communication to the more restricted environment of computer programming. The activities were selected and sequenced so that a concept or technique learned in one task was useful in performing the next. In completing these tasks, students were required to interchange the values of two variables, order the values of a visible array, select a value from a hidden array, and order the values of a hidden array.

The MEMOPS protocols exposed some differences between experienced and inexperienced programmers as well as procedural differences among students within these groups. On their first attempt at interchanging the values of two memory cells, nearly all of the less experienced programmers destroyed an original value. The technique of preserving a value by copying it into an unused memory location was quickly mastered, however, and presented only minor problems on subsequent tasks. A second interesting difference between the experience groups was the choice of memory location used in preserving a value. All of the inexperienced students used cell Z to preserve a value whereas all but one of the experienced students used array element X[3]. The novice programmers probably selected the Z cell because they had been required to use it in the previous MEMOPS activities. The more

experienced programmers, however, may have chosen cell X[3] because it was physically closer to the cells containing the two elements that were to be exchanged. Students with no prior programming experience found this interchanging of values to be somewhat challenging, although once the technique was mastered it presented only minor problems on subsequent tasks.

Nearly all students used similar methods to complete the visible reordering and hidden selection tasks. Visible reordering was accomplished by inspecting all of the values and shuffling those that were out of order. For most students, the cells of an array were filled in a sequential manner from top to bottom. On the hidden selection, a "keeps best" technique was used by all but four of the students.

The hidden ordering problem could have been efficiently solved by combining the "keeps best" technique with a sequential fill; however, no inexperienced students and only half of the experienced students chose this approach. Most of the students made sufficient comparisons to determine the relationships among all of the values and then completed the task as they had the visible ordering. Throughout the MEMOPS activities, established previous experiences were chosen over recently acquired techniques as tools for building solutions.

Posttest 1 Findings

The first posttest was designed to measure the student's ability to generate Pascal code for two simple programming problems, a two-variable swap and a three-variable sort. For both problems, students were given an incomplete program and instructed to add whatever Pascal code might be necessary to complete it. After writing an initial solution on paper, students were allowed to enter the solutions into the MINIPAS computing environment and modify them until the program successfully solved the given task. The findings reported in this section are based upon analyses of both the paper solution attempts and the MINIPAS computing histories for each of the two posttest programming tasks.

Individual student performance on the swap problem

The first programming task on the posttest was to swap (interchange) the values of two variables. Conceptually, this task was identical to the swap performed by the students in the MEMOPS environment. The optimal solution for this problem consisted of adding three assignment statements to the incomplete code that was given. Six features of the paper solution attempts were documented. They included use of 1) syntactically correct Pascal statements to complete the task, 2) correct logic, 3) MEMOPS MOVE instructions, 4) free memory locations to preserve values and 5) unnecessary programming statements. The sixth feature documented the presence of

a "wrong-way" assignment error in the student's code. Appendix Tables C-1 and C-2 document these solution features for the swap problem.

Fourteen of the thirty-six students produced initial solution attempts which were entirely correct while another thirteen produced solutions that were judged to be logically correct. Many of these logically correct solutions contained syntactically flawed Pascal assignment statements. Five experimental students incorporated MEMOPS MOVE statements into their programs that logically interchanged the values. The attempts of nine students (T01, T03, T10, T15, C02, C04, C05, C07, C08) were illogical for the given problem and also contained unnecessary and often syntactically incorrect Pascal statements. Of these nine students, one (T15) had previously programmed in FORTRAN and another (C08) had programmed in BASIC.

Another feature on which the solutions differed was the number of variables used to preserve original values. In the MEMOPS program, the students in the treatment group had used a single variable for this purpose. They did not, however, universally transfer this procedure to the Pascal problem. A majority of students (24) completed the program by assigning the original values of A and B to variables C and D, respectively, and then reassigning the values of C and D to B and A. Only seven students (T05, T09, T14, T17, T18, C14, C16) wrote a solution that used a single additional memory location to temporarily preserve an original value. Four students (T01, T03, C02, C04) failed to use any additional variables to preserve values.

In addition to the use of MEMOPS MOVE statements, three other types of errors were made on the initial solution attempts to the swap problem. These errors reflected misunderstandings concerning the functions of specific coding statements, Pascal syntax requirements, and direction of assignment. Eight students (T01, T03, T08, T15, C02, C04, C05, C07) included extraneous IF, READLN and WRITELN statements in their solutions. Over half of the students wrote code that contained numerous syntax errors. The two most prominent syntax errors were failure to separate statements using semicolons (;) and use of "=" as the assignment operator rather than ":=". Seven students (T02, T07, T08, T10, C05, C06, C13) wrote assignment statements that exhibited a "wrong-way" assignment error. Unlike the mathematical equality operation where the "=" does not denote a direction, imperative programming languages such as Pascal invoke right-to-left assignment where the value of the variable to the right of the assignment operator (:=) is assigned to the variable named on the left of the operator. Figure 13 illustrates a solution to the swap problem that exhibits a "wrong-way" assignment error.

Protocols of each student's online programming efforts for the Pascal swap problem were developed using the data recorded in the MINIPAS histories. They documented the 1) number of initial compilation attempts prior to obtaining the first executable version of the program, 2) total number of compilations attempted for all versions, 3) number of unique executable program versions, 4) specific programming problems encountered by the student, 5) time taken to

Correct Solution	Incorrect Solution
C:=A; A:=B; B:=C;	A:=C; B:=A; C:=B;
(right-to-left assignment)	(left-to-right assignment)

FIGURE 13. A correct solution and one exhibiting the "wrong-way" assignment error

generate a correct solution using the MINIPAS environment, 6) correctness of logic for the final solution attempt and 7) number of additional memory locations used.

In using the MINIPAS compiler to produce a final solution, the students compiled, altered and recompiled their programs until all syntax errors were eliminated. They then executed their programs, modified them for logic errors and recompiled them until their programs executed properly. An average of 4.83 compilation attempts per student were required to produce the first executable version of the solution and 6.47 total compilations were made. The average number of unique executable program versions per student for the problem was 2.33. Only three students (T03, C02, C04) failed to successfully compile their solution attempts and thus had no executable versions.

The MINIPAS histories indicated that syntax was a major problem for several students (T03, T06, T09, T11, T13, C02, C03, C04, C05, C07, C08, C17). Of noted difficulty, and recorded separately, was the

use of a "wrong-way" assignment statement. On their paper solutions, seven students demonstrated confusion on assignment direction. However, the MINIPAS histories revealed that thirteen students (T01, T02, T05, T06, T07, T08, T09, T10, T11, C05, C06, C07, C13) experienced this problem.

Algorithm development caused difficulty for six students (T03, T15, C02, C04, C05, C08). Three of these students (C04, C05, C08) initially tried to interchange values by issuing the assignment statements "A:=B; B:=A". Among the students who issued syntactically correct assignment statements, proper ordering of these statements was yet another problem (T01, T05, C04, C05, C07, C08). As one would expect, students with little or no prior programming experience encountered a wider range of problems attempting to solve the swapping task than did the students who had previously programmed in either FORTRAN or Pascal.

In addition to the fourteen students who had correct paper solutions, sixteen more students were able to generate correct solutions utilizing the MINIPAS programming environment. Six students (T03, T06, C02, C04, C07, C08) failed to produce an acceptable solution to the problem in the allotted fifty minutes. Of these six students, four had never programmed before enrolling in the present course and one had written some BASIC programs.

Treatment group comparisons for the swap problem

The performances of the students in the MEMOPS and NON-MEMOPS groups were compared on both their initial and final solution attempts to the swap problem. For the initial solution attempt, chi-square tests of independence were conducted for the number of students 1) writing a syntactically correct solution and a logically correct solution, 2) using one or two variables to preserve original values, 3) inserting unnecessary code into the solution, and 4) writing code that exhibited the "wrong-way" assignment error. These tests indicated that performances on the initial solution attempts did not differ significantly between treatment groups. The summary data used in these analyses are presented in Table 1.

TABLE 1. Number of students exhibiting selected solution features in their initial solution attempts for the swap problem

	Solution Features				
Group	1	2	3	4	5
Treatment (n=18)	6	14	(1 variable) 5 (2 variables) 11	4	4
Control (n=18)	7	13	(1 variable) 2 (2 variables) 14	4	3

Solution Features:
1 Syntactically and logically correct solutions
2 Logically correct solutions
3 Additional variables used to preserve values
4 Solutions containing unnecessary code
5 Solutions exhibiting the WRONG-WAY assignment error

Using the MINIPAS history data, t-statistics were computed on the average number of initial compilations, number of total compilations, and number of unique versions. No statistically significant differences between the groups were found on the average number of initial compilations ($t(34) = .58$, $p < .57$) or total compilations ($t(28) = .46$, $p < .66$). A test of homogeneity performed on the variances of the groups for the number of total compilations was statistically significant ($F(17,17) = 2.85$, $p < 0.04$) and indicated that the variance for the treatment group was less than that of the control group. No differences between treatment groups were found for the mean number of executable versions ($t(34) = .55$, $p < .58$). Descriptive statistical information for the two experimental groups using the MINIPAS compiler is reported in Table 2.

The types of problems students encountered in attempting to program a solution did not differ between the two experimental groups except for the "wrong-way" assignment error. As is shown in Table 2, the number of treatment students exhibiting this problem was more than twice as large as the number of control students making the same mistake (9 students versus 4 students). This difference approached statistical significance (chi-square = 3.01, df = 1, $p < .08$). The total number of problems encountered by the two groups revealed little difference (19 for the MEMOPS students and 22 for the control group). However, in comparing only those students from each group who had little or no previous programming experience, the NON-MEMOPS control students encountered a greater variety of problems than did the MEMOPS treatment students.

TABLE 2. MINIPAS history statistics for the swap problem

Group	MINIPAS History Features						
	Initial Compilations		Total Compilations		Unique Versions		Programming Problems
	Mean	S.D.	Mean	S.D.	Mean	S.D.	
Treatment (n=18)	4.28	4.47	5.94	5.00	2.47	2.43	(A) 6 (D) 2 (B) 9 (E) 0 (C) 2
Control (n=18)	5.39	6.85	7.00	8.44	2.19	2.37	(A) 7 (D) 4 (B) 4 (E) 3 (C) 4

Programming Problems:
(A) Syntax
(B) WRONG-WAY assignment
(C) Logic
(D) Ordering
(E) Attempted A:=B; B:=A solution

Whereas the non-experienced treatment students primarily made "wrong-way" assignment errors, non-experienced control students encountered syntax, logic, and ordering problems.

Since several students did not successfully complete the program, a direct comparison of completion times for the two groups was deemed inappropriate. However, an indirect comparison was made by dichotomizing this variable into completion times greater than ten minutes versus completion times less than ten minutes. Ten minutes was chosen as the criterion after an inspection of the data revealed that minor problems could be resolved within that time frame. Based on this classification, the students in the MEMOPS treatment group took more time to solve the

problem than did the students in the NON-MEMOPS control group (chi-square = 5.35, df = 1, $p < .03$). This information is reported in Table 3.

TABLE 3. Number of students exhibiting selected solution features in their final solution attempts to the swap problem

	Solution Features				
Group	1	2	3	4	
Treatment (n=18)	6	10	2	(1 variable)	3
				(2 variables)	14
Control (n=18)	12	2	4	(1 variable)	2
				(2 variables)	12

Solution Features:
1 Correct solutions generated in 10 minutes or less
2 Correct solutions generated in more than 10 minutes
3 Incorrect solutions
4 Additional variables used to preserve values

Of the four MEMOPS treatment students (T01, T03, T15, T10) whose initial solution attempts were illogical and poorly defined, two students (T15, T10) were able to generate a correct solution using MINIPAS. Of the five NON-MEMOPS control students whose first attempts were illogical (C02, C04, C05, C07, C08), only one student (C05) was able to generate a correct solution using MINIPAS. The final solution attempts of the other four students suggested that they made very little progress toward generating a correct solution in the allotted fifty minutes.

Individual student performance on the three-variable sort problem

The second programming task on the first posttest was to sort three numbers from smallest to largest. The students were given a partial program in which three values were stored into the variables A, B, and C. The students were required to add code to order the values so that A contained the smallest and C contained the largest value. The challenge in solving this problem was to break the problem into three parts which could be attacked separately.

An efficient solution to the three-variable sort problem is shown in Figure 14. In this solution, the values of A and B are compared and if A is larger they are reordered. Then the values of B and C are tested, and if B is larger, B and C are reordered. At this time in the execution of the program, C will always contain the largest value. However, if C initially stored the smallest value, the preceding changes would cause the values of A and B to be improperly ordered. This condition necessitates a second comparison of A and B as the final step in the solution. For this particular solution, the state of the problem resulting from the execution of an IF statement is always the same. That is, the first test comparing A and B always results in the larger value being placed in B and the smaller in A.

A second solution arises from analyzing the problem with respect to all possible initial conditions and processing each independently. That is, if the values are arranged so that A is greater than B and B is greater than C, then the values of A and C need to be exchanged. With three variables there are six possible initial states, five of which must

```
If A > B then
        begin
        D:=A; A:=B; B:=D
        end;

If B > C then
        begin
        D:=B; B:=C; C:=D
        end;

If A > B then
        begin
        D:=A; A:=B; B:=D
        end;
```

FIGURE 14. Three-variable sort problem: efficient solution

be identified and reordered. An "isolate all cases" algorithm of this type is shown in Figure 15.

```
If (A > B) and (B > C) then {code to order all three numbers}
If (A > C) and (C > B) then {code to order all three numbers}
If (B > A) and (A > C) then {code to order all three numbers}
If (B > C) and (C > A) then {code to order all three numbers}
If (C > A) and (A > B) then {code to order all three numbers}
If (C > B) and (B > A) then {code to order all three numbers}
```

FIGURE 15. Three-variable sort problem: isolate all cases solution

A third solution is obtained when a free memory location is used to retain the larger value discovered by a comparison. This approach leads to a complex solution because it necessitates "remembering" the variable from which the temporary cell received its value. This third solution reflects an incomplete segmentation of the original problem into

independent parts. An example of the "complex shuffle" algorithm is shown in Figure 16.

```
If (A < B) then  D:=B
           else  begin D:=A; A:=B end;
If (A < C) then  B:=C
           else  begin B:=A; A:=C end;
If (B < D) then  C:=D
           else  begin C:=B; B:=D end;
```

FIGURE 16. Three-variable sort problem: complex shuffle solution

Like the efficient solution in Figure 14, the complex shuffle contains only three comparisons. After the first test (A < B), the variable D is given the larger value and A the smaller. The relative size of B is unknown at this point. After the second test (A < C), variable A will contain the smallest of the three values and variables B and D will contain the two larger ones. Now, the content of variable C is irrelevant. The third test (B < D) permits the proper ordering of B and C. In developing this algorithm, the task of determining the output of one step which can serve as input to the next places a heavy burden on the programmer. This is complicated by the use of the variable D and the unknown relative size of one of the variables.

The students were required to write their initial solutions to the three-variable sort problem on paper. These attempts were analyzed to determine 1) whether the solution was syntactically correct, 2) whether the solution's logic was correct, 3) what algorithm was being attempted (efficient solution, isolate all cases or complex shuffle), 4) whether

there was evidence the student knew that original values could be potentially destroyed, 5) how values would be interchanged, 6) if assignment statements were used, and 7) whether compound statements were being used in the IF statements. Solution features for the three-variable sort problem are documented in Appendix Tables C-3 and C-4.

A solution was judged to be correct if it could be entered into the computer and would produce the correct output with no modification. If the solution had only minor errors in form but showed correct and complete logic, it was judged to be a logically correct solution. For example, solutions which would produce a compilation error only because of missing semicolons or omitted BEGINs and ENDs surrounding assignment statements were judged to be logically correct. Using these criteria, the paper attempts of three students (T17, T16, C14) were judged to be logically and syntactically correct. Three more students (T13, C15, C16) wrote logically correct solutions that contained syntax errors. All six of the students who wrote logically correct solutions had previous programming experience in either FORTRAN or Pascal. The solution attempts of the remaining thirty students contained both logic and syntax errors.

Solution algorithms to the three-variable sort problem were classified as one of the following four types: an efficient solution (Figure 14), an "isolate all cases" solution (Figure 15), a "complex shuffle" solution (Figure 16), or an indeterminate solution. If the solution contained IF statements similar to those shown in Figure 14 and some indication that the student intended to exchange the values of two

variables, it was classified as an efficient solution. If the solution consisted of a sequence of IF statements containing boolean expressions that included three variables, it was classified as an isolate all cases solution. A solution in which 1) a value was assigned to a free memory location (D) and 2) an IF statement was encountered before the value was copied to another variable was classified as a complex shuffle. Solutions that did not fit into one of these three categories were classified as indeterminate solutions.

Ten students (T10, T12, T13, T16, T17, T18, C11, C13, C14, C17), all of whom had prior programming experience, tried to implement the efficient solution algorithm on their paper attempts. Eight students (T05, C01, C03, C06, C08, C09, C12, C16) tried to use the "isolate all cases" algorithm and twelve students (T02, T04, T07, T08, T09, T11, T14, T15, C07, C10, C15, C18) attempted the complex shuffle. The algorithms of six students (T01, T03, T06, C02, C04, C05) could not be categorized as they were incomplete and showed no distinct initial features. None of these last six students had any previous programming experience.

Solution attempts were also analyzed for evidence that the student possessed the knowledge that original values of variables would be destroyed when new assignments were made. Classification for this particular characteristic was complicated by two factors. In cases where students issued code such as "A:=a; B:=b; a:=B; b:=A", it was assumed that the student was attempting to preserve the original values even though these values would not be saved since Pascal compilers do not distinguish between upper and lower case letters in variables names. In

other cases, the destruction of values was attributed to faulty logic rather than a lack of knowledge concerning the principle. For example, consider the code "If A > C then D:=A; If B > C then A:=C". In this case, it was assumed that the student was thinking something like "If A > C then begin D:=A; If B > C then A:=C; end". Although the logic is fragmentary, the student did appear to be trying to save the original value of variable A. Using these criteria, twenty-one students were judged to have written code that demonstrated knowledge that original values would be destroyed when new assignments were made. The code of six students (T07, C01, C05, C06, C08, C09), two of whom had programmed in BASIC, suggested that they were unaware of this principle. The written code of six inexperienced students and one student who had programmed in Pascal (T03, T05, T06, C02, C03, C04, C18) could not be classified.

The swapping technique utilized by students who realized the potential for destroying values was also documented. In general, students with previous FORTRAN or Pascal experience compared two variables and if appropriate exchanged the values before making any other comparisons. Students with little or no previous experience either failed to complete the exchanges before making additional comparisons or wrote code that made no attempt to exchange values at all. Only eleven students (T10, T12, T13, T16, T17, T18, C11, C12, C13, C14, C17) completed the exchanges before making additional comparisons.

Two students displayed unique behaviors concerning the swapping features of their algorithms. One student (C03) attempted to store the

order of all three values into a single variable (D := A, B, C). Another student (C16) declared three new variables and placed the original values in order into these variables.

In addition to documenting the type of solution attempted and the swapping technique used to exchange values, the use of assignment and IF statements was also recorded. Four students (T03, T05, C04, C18) failed to issue any assignment statements in their initial solution attempts. Only one student (T01) failed to use an IF statement. Closer inspection of the IF statements showed that twenty-four students used multiple assignment statements that were to be executed based upon the outcomes of IF tests. Eleven students (T05, T06, T07, T09, T15, C02, C03, C04, C07, C08, C18) failed to realize the need for multiple operations and used a single assignment statement. Of these eleven students, nine had little or no previous programming experience.

Information recorded in the MINIPAS histories was used to develop protocols of online programming performance for the three-variable sort. As was true for the swap problem, the online protocols documented 1) the number of initial compilation attempts prior to obtaining the first executable version of the program, 2) the number of total compilations attempted for all program versions, 3) the number of unique executable versions, 4) specific programming problems encountered by each student, 5) MINIPAS completion time for correct solutions, and 6) final solution attempts.

Besides the three students who had written correct paper solutions, an additional nine students (T04, T12, T13, T16, C10, C12, C15, C16, C17)

generated a correct solution using the MINIPAS programming environment. Disregarding syntax errors, two more students (C01, C11) wrote logically correct solutions to the three-variable sort problem. Of the eleven students who had logically correct solutions on their final attempt, all but two (T04, C01) had previous programming experience.

Students averaged 6.75 initial compilation attempts prior to executing the first version of their program in MINIPAS. The standard deviation for initial compilation attempts was 7.24 compilations. This indicated extensive syntax problems and wide differences among students. The average number of total compilations required to debug the logic for the sort problem was high at 11.47 with a standard deviation of 10.68. Five students (T06, T15, C03, C04, C08) failed to successfully compile their initial attempts and thus had no executable versions.

The MINIPAS histories disclosed several types of programming problems encountered by the students. The most prominent problems involved the syntax of the IF statements. Ten students (T01, T03, T15, C01, C02, C03, C04, C05, C08, C09) had trouble correctly formatting the boolean expressions used in the IF statements. Nine of these students had little or no previous programming experience. Sixteen students (T02, T04, T05, T08, T09, T10, T11, T12, T13, T16, C01, C10, C11, C12, C13, C18) failed to surround multiple statements in the alternatives of the IF with the reserved words BEGIN and END. Because the BEGIN/END structure is not found in languages such as FORTRAN and BASIC, students with prior programming experience as well as inexperienced programmers made this error. The order in which comparisons between variables were made and

the order in which values were assigned to variables also caused problems for several other students.

Three students (T05, C03, C16) had unique programming problems. As previously mentioned, student C03 attempted to store the order of the values into a single variable. Student T05 tried to use READLNs to interchange values. Student C16 assumed that the memory locations labeled A, B, and C were different from the locations of a, b, and c.

In spite of the severity of difficulties that many students encountered, the protocols of the final solution attempts revealed that only four students (T14, C01, C07, C15) implemented a different solution algorithm than the one used on the initial solution attempt. Only one (T01) of the six students whose initial solution algorithm was indeterminable employed a classifiable algorithm on his final attempt. Two students (C01, C09) who had not employed a swapping technique on their initial attempts did so on their final attempts. Eight students (T01, T03, C02, C03, C04, C06, C08, C18) did not employ any swapping technique on their final solution attempts. Six of these eight students had no previous programming experience.

All of the students issued at least one IF statement in their final solution attempts to the three-variable sort problem, but two students (T03, T05) with no programming experience failed to use any assignment statements in their final solutions. Six students (T08, T14, C01, C08, C11, C13) generated solutions that correctly ordered at least one specific set of values but did not solve all possible combinations of

values. All but one of these students had previous programming experience.

Treatment group comparisons for the three-variable sort problem

As was true for the swap problem, comparisons between the performances of the students in the MEMOPS and NON-MEMOPS groups were made for the three-variable sort problem. Concerning their initial solution attempts, chi-square tests of independence were conducted for the number of students 1) writing a syntactically correct solution, 2) writing a solution that was logically correct, 3) attempting to implement the efficient, "isolate all cases" or "complex shuffle" algorithms, 4) preserving original values, 5) completing value exchanges before making additional comparisons and 6) using assignment and IF statements.

For the initial paper solutions, no differences were found between the groups for the number of logically correct or completely correct solutions, the number of students demonstrating knowledge that original values might be destroyed, the swapping technique used to interchange the values of variables, or the use of assignment and IF statements. The test statistic for demonstrating knowledge that original values might be destroyed only approached significance (chi-square = 2.16, df = 1, $p < .15$). Fourteen students in the MEMOPS group and nine students in the control group demonstrated this knowledge.

A significant difference was found, however, for the type of algorithm (efficient, isolate all cases, or complex shuffle) that was used (chi-square = 6.23, df = 2, $p < .05$). Whereas seven students in the control group attempted the "isolate all cases" solution, only one

student attempted to use it in the treatment group. Eight students in the treatment group attempted the "complex shuffle" solution as compared to four students in the control group. Summary data for the two treatment groups on the students' initial solution attempts to the three-variable sort problem are presented in Table 4.

TABLE 4. Number of students exhibiting selected solution features in their initial solution attempts to the three-variable sort problem

	Solution Features						
Group	1	2	3	4	5	6	7
Treatment (n=18)	2	3	(E) 6 (I) 1 (CS) 8	(+) 14 (-) 1	(+) 6 (-) 8	16	(SI) 5 (CI) 12
Control (n=18)	1	3	(E) 4 (I) 7 (CS) 4	(+) 9 (-) 5	(+) 5 (-) 8	16	(SI) 6 (CI) 12

Solution Features:
1 Syntactically and logically correct solutions
2 Logically correct solutions
3 Solutions attempting to implement the efficient (E), "isolate all cases" (I), and "complex shuffle" (CS) algorithms
4 Solutions with code demonstrating presence (+) or absence (-) of the principle concerning preservation of values
5 Solutions containing value exchanges that were completed (+) before additional comparisons were made and those that didn't complete the exchanges (-) before making additional comparisons
6 Solutions containing assignment statements
7 Solutions containing IF statements with single assignment statements (SI) or compound assignment statements (CI)

Using the MINIPAS history data, t-statistics were computed on the average number of initial compilations, number of total compilations, and

number of unique versions. No statistically significant differences between the two treatment groups for the mean number of initial compilations ($t(28) = 1.78$, $p < .09$), total compilations ($t(34) = -.02$, $p < .99$), or the number of unique program versions ($t(34) = -.55$, $p < .59$) were found. A test of homogeneity of variances on the number of initial compilation attempts indicated that there was a difference between treatment group variances ($F(17,17) = 2.90$, $p < .04$). The variance of the control group was nearly three times larger than the variance of the treatment group for this particular characteristic. Chi-square tests comparing the number of students encountering each of five specific programming problems again showed no statistically significant performance differences for the two treatment groups. A summary of the data recorded in the MINIPAS programming histories is shown in Table 5 for the two treatment groups.

Using data obtained from the MINIPAS histories on the final solution attempts to the three-variable sort problem, additional comparisons were made between the treatment groups. These comparisons were made on the following features: type of algorithm implemented, demonstration of knowledge that original values might be destroyed, swapping technique used to exchange variable values, use of assignment and IF statements, and number of solutions that solved a limited set of values.

As was true for the initial solution attempts, a significant difference was found for the types of algorithms implemented by the students in the two treatment groups on their final solution attempts (chi-square = 6.69, df = 2, $p < .04$). Eight students in the control

TABLE 5. MINIPAS history statistics for the three-variable sort problem

Group	MINIPAS History Features						
	Initial Compilations		Total Compilations		Unique Versions		Programming Problems
	Mean	S.D.	Mean	S.D.	Mean	S.D.	
Treatment (n=18)	4.67	5.03	11.50	12.57	2.28	0.54	(A) 3 (D) 5 (B) 10 (*) 1 (C) 6
Control (n=18)	8.83	8.57	11.44	8.77	2.53	0.60	(A) 6 (D) 5 (B) 6 (*) 1 (C) 4

Programming Problems:
(A) IF syntax (boolean expression component)
(B) IF syntax (BEGIN END for compound statements)
(C) Order in swapping code
(D) Order of IF tests
(*) Unique problem

group attempted the "isolate all cases" algorithm as compared to two in the treatment group. Seven students tried to implement the "complex shuffle" in the treatment group as compared to two students in the control group. The chi-square statistics for all of the remaining features (demonstration of value preservation principle, swapping technique, use of assignment and IF statements, and number of solutions that solved a limited set of values) failed to reveal any statistically significant performance differences between the two treatment groups for these factors. The frequencies used in these computations are presented in Table 6.

TABLE 6. Number of students exhibiting selected solution features in their final solution attempts to the three-variable sort problem

	Solution Features							
Group	1	2	3	4	5	6	7	8
Treatment (n=18)	6	6	(E) 7 (I) 2 (CS) 7	(+) 15 (-) 0	(+) 6 (-) 8	15	(SI) 7 (CI) 11	2
Control (n=18)	6	8	(E) 5 (I) 8 (CS) 2	(+) 11 (-) 4	(+) 8 (-) 3	18	(SI) 5 (CI) 13	4

Solution Features:
1 Syntactically and logically correct solutions
2 Logically correct solutions
3 Solutions attempting to implement the efficient (E), "isolate all cases" (I), and "complex shuffle" (CS) algorithms
4 Solutions with code demonstrating presence (+) or absence (-) of the principle concerning preservation of values
5 Solutions containing value exchanges that were completed (+) before additional comparisons were made and those that didn't complete the exchanges (-) before making additional comparisons
6 Solutions containing assignment statements
7 Solutions containing IF statements with single assignment statements (SI) or compound assignment statements (CI)
8 Solutions solving limited sets of values

Summary of posttest 1 findings

The first posttest consisted of two Pascal programming problems. The initial problem required the students to write code to swap the values of two variables. This problem was easily solved by all but one of the students who had previously programmed in either FORTRAN or Pascal. It was a challenge, however, for many of the students with less experience. The beginning programmers experienced difficulties in

formulating the required logic and encountered severe problems with Pascal syntax.

The total number of syntax problems for the treatment and control groups were approximately equal; however, the types of errors encountered were noticeably different. The treatment students made many more "wrong-way" assignment errors while the control students made a greater number of errors of other types. The "wrong-way" assignment errors appear to be directly and logically attributable to the direction of the MOVE statement that the students used in the MEMOPS program. As a result of the confusion caused by the direction of the assignment statement, the time required to complete the problem was much greater for the treatment students. In contrast to syntax difficulties, logic difficulties appeared to be slightly more prevalent and more persistent among students in the control group.

The second problem on the posttest was a three-variable sort problem. It proved to be much more challenging than the swap problem, causing both logic and syntax difficulties. The syntax of the IF statement was particularly troublesome. While the number of difficulties were greater for the inexperienced programmers, experience was not a factor in identifying the type of difficulties encountered.

Algorithms chosen by the MEMOPS and NON-MEMOPS groups differed significantly for the three variable sort. Many students in the NON-MEMOPS group elected to identify all possible cases and handle each case separately. This choice necessitated a complex boolean expression within the IF statements which produced syntax errors. In contrast, many

students in the MEMOPS group chose an algorithm similar to what they had used in the MEMOPS visible sorts. This was a very complex algorithm resulting in problems of determining correct logic. Also attributable to this algorithm were syntax errors in the use of BEGIN and END words in the argument portion of the IF statements.

For both the swap and three variable sort problems, some students wrote code that destroyed the original values. For the swap problem three students, all of whom were in the control group, destroyed values. In dealing with the complexity of the three variable sort problem, six students wrote code which destroyed at least one of the original values. Five of these students were members of the control group and one had experienced MEMOPS. Of the students who preserved values the vast majority in both groups used two additional variables. This was somewhat surprising for the MEMOPS students since they had used a single variable for swapping values throughout their MEMOPS activities.

Posttest 2 Findings

The second posttest was administered after students had been given instruction on Pascal looping constructs and on array implementation. This test was divided into two parts which were evaluated separately. On the first part of the test the students were expected to read and interpret Pascal code. The problems required identification of incorrect array declarations, locating boundary violations of arrays addressed within FOR loops, and computing the final values of arrays after program execution. There were 29 subitems on this part of the test.

The second part of the test consisted of three programming problems. For these problems the program headings and variable declarations were provided and the student's task was to supply Pascal code that performed three specific functions. These functions were 1) comparing the contents of two arrays, 2) reversing the order of the values stored in a single array, and 3) sorting the values of an array into ascending order. For the first two problems students were only asked to write the code, but for the third problem they were allowed to enter their solutions into the computer and debug them. A copy of the second posttest as well as a description of the scoring procedures for each programming problem can be found in Appendix D.

The mean achievement scores for the MEMOPS and NON-MEMOPS students on both parts of the tests were compared. These scores did not differ significantly on either the first part of the test ($t(27) = -.39$, $p < .701$) or the second part ($t(27) = -.18$, $p < .859$). Group means and standard deviations for these two scores are reported in Table 7. Protocols of the solution features of the programming problems on the second part of the posttest, however, suggested that the MEMOPS students approached two of the three problems in a different manner than did the NON-MEMOPS students.

Individual student performance on the 2-array comparison problem

The fifth problem on the posttest required the student to write code that would sequentially compare the values of the elements of two arrays. Messages were to be printed following each comparison indicating which element contained the larger value. The most efficient solution to this

TABLE 7. Mean achievement scores and standard deviations for the second posttest

Groups	Part I			Part II		
	Maximum Score	Mean	S.D.	Maximum Score	Mean	S.D.
Treatment (n=14)	29	21.00	7.40	28	18.50	9.20
Control (n=15)	29	19.93	7.45	28	17.93	7.82

problem was to use the index variable of a FOR loop to sequentially compare the elements in the two arrays. This solution is illustrated in Figure 17. The protocols for this problem documented 1) the correctness of each student's solution in terms of logic and syntax, 2) use of a FOR loop to sequentially move through the array, 3) use of an IF statement to compare cell values and 4) use of an index variable to address the cells of both arrays. Appendix Tables D-1 and D-2 document the solution features of the treatment and control groups for this problem.

```
For I := 1 to MAX Do
      Begin
      If X[I] > Y[I]
         then Writeln ('X[',I,'] is larger than Y[',I,']');
      If Y[I] > X[I]
         then Writeln ('Y[',I,'] is larger than X[',I,']')
      End;
```

FIGURE 17. Solution to the 2-array comparison problem

Of the twenty-nine students attempting the comparison problem, sixteen students wrote syntactically correct solutions. The solutions of nine more students were logically correct since they used a FOR statement to sequentially move through the arrays and an IF statement to compare cell values. The four students who failed to solve this problem encountered major difficulties with the looping structure. Two students (T10, T15) used the indices of two FOR loops to address the array cells and other student (C08) used an incorrect form of the WHILE structure (e.g., WHILE I:=1 TO 5 DO). Only one student (T09) failed to use a looping structure in his solution. Additional solution errors included placing a semicolon before an ELSE statement (a syntax error) and attempting to address a cell using an incorrect index. Students with prior FORTRAN or Pascal experience tended to write syntactically correct solutions. The less experienced students wrote logically correct solutions that contained minor syntax errors.

Individual student performance on the reversal problem

The programming task on the sixth problem was to reverse the original order of an array's values. Besides filling in the bounds to the given FOR statement, the student was required to add the code that would interchange element values. A correct solution to this problem consisted of writing Pascal code that exchanged the values of the first and last elements, the second and next-to-last elements, and so forth until the midpoint of the array was reached. Two slightly different approaches might be taken to solve the reversal problem.

In the first approach, all array elements are addressed directly using the index of the FOR loop. If MAX is defined to be the size of the array and I is the index variable, X[I] would be interchanged with X[MAX + 1 - I]. A solution that uses an index variable to address both elements is shown in Figure 18.

```
For I := 1 to MAX DIV 2 do
    Begin
    TEMP := X[I];
    X[I] := X[MAX+1-I];
    X[MAX+1-I] := TEMP
    End;
```

FIGURE 18. Single index solution to the reversal problem

A second approach to the problem would be to use two different variables rather than one to address the cells of the array. The index of the FOR loop is used to address one of the array cells and a second variable is used to address the other cell. This second variable is assigned the value of the constant MAX before the first pass is made through the loop and decreased by one for each additional pass. A solution that uses two variables to address the elements of the array is shown in Figure 19. As was true for the first solution, MAX has been defined to be the size of the array.

Protocols for the reversal problem documented correctness of solution in terms of logic and syntax, algorithm implementation (single index, two-variable, or indeterminate), knowledge of the principle concerning the need to preserve values when new assignments are made, use

```
J := MAX;
For I := 1 to MAX DIV 2 do
   Begin
   TEMP := X[I];
   X[I] := X[J];
   X[J] := TEMP;
   J := J-1;
   End;
```

FIGURE 19. Two-variable solution to the reversal problem

of correct bounds in the FOR statement, and number of additional memory locations used for preserving original values. Credit for correct logic was awarded if the student wrote only assignment statements to complete the solution. Credit for use of correct bounds was awarded if the bounds were expressions evaluating to one and five. Appendix Tables D-1 and D-2 document the solution features for this reversal problem.

Nine students (T04, T11, T12, T14, T16, T17, C06, C14, C17) wrote correct solutions to the reversal problem. Five more students (C05, C10, C11, C12, C13) wrote solutions that were logically correct. Of these fourteen students who wrote correct or logically correct solutions, only three (T04, C05, C06) had no previous programming experience. The nineteen students who were unable to solve the reversal problem encountered a variety of difficulties. Sixteen students wrote incorrect bound values to the FOR statement and seven students (T09, T15, C02, C03, C08, C09, C18) failed to use a free memory location to preserve values. Other errors included attempts to use two FOR loops to sequentially move through the array (T13, T18, C16), ordering errors in the assignment statements that exchanged values (T02, C05, C13) and misplacement of

initialization statements with respect to the body of a FOR loop (C11, C12).

Seventeen students attempted to solve the reversal problem using the single index algorithm. Eight, all of whom had previous FORTRAN or Pascal experience, attempted to use the two-variable approach. The algorithms of five students (T09, T15, C08, C09, C18) were not classifiable.

Treatment group comparisons on the 2-array comparison and reversal problems

The comparison problem was very easy for most students. Logically correct solutions were produced by all but three students in the treatment group and one student in the control group. A nearly equal number of students in both groups wrote logical solutions that contained syntax errors (four in the treatment group, five in the control group). No performance differences were found for use of a FOR and an IF statement, nor for using the index variable of the FOR loop to address the elements of both arrays. Thus, no programming differences between the groups were revealed on the comparison problem.

Although the differences were not statistically significant, the more experienced students in the the two groups did appear to approacn the reversal problem in a slightly different manner. Of the tnirteen students in the treatment group whose solution algorithms could be classified, ten attempted to implement the single index algorithm. For the control group, students writing classifiable algorithms were more evenly split with seven using the single index algorithm and five

attempting the two-variable algorithm. Two students in the treatment group failed to demonstrate the need for preserving original values, as opposed to five students in the control group. Students in both groups who did realize the need for preserving original values used only one additional variable for this purpose.

Individual student performance on the ascending sort problem

The programming task for the seventh problem was to reorder the values of an array into ascending order. The student was given an incomplete program that contained the declarations and statements that would read values into a six-cell array. The student's task was to add the code that would reorder the values so that the first cell (X[1]) contained the smallest value originally stored in the array and the sixth cell (X[6]) contained the largest value. Efficient solutions to the problem require the use of loops to sequentially compare cell values and interchange them if they are out of order.

Prior to the second posttest, all students had been conceptually introduced to two different sorting algorithms, a selection sort and a bubble sort. This introduction focused on differences in problem representation between the two algorithms, not upon any Pascal coding implementations. In fact, students were not shown any coding details for either algorithm. The content of the introduction was similar in nature to the operational descriptions of the algorithms that follow, sans references to Pascal coding statements.

In a selection sort, an element is selected and compared to each subsequent element in the array. After each comparison, values of the

two elements are interchanged if the test element is larger than the comparison element. On the first pass through the array, the first element is selected as the test element and compared to all other elements in the array. On the second pass, the second element is selected as the test element and is compared to the remaining values stored in the array. This process, of selecting a test element and comparing it with all subsequent elements, is repeated until all of the values are reordered.

Figure 20 graphically illustrates a selection sort. The arrow to the left of each array denotes the test element for a particular pass and the arrows to the right mark the comparisons that are made between the test element and the remaining elements. Note that the effect of this algorithm is to fill the array with the reordered values from top to bottom.

The Pascal code to implement the selection sort for reordering array values in ascending order is shown in Figure 21. The index variable of the outer FOR loop (I) is used to select the test element for each pass. The index variable of the inner FOR (J) is used to address the subsequent elements that will be compared to the test element. MAX is a constant defined to be the size of the array.

In the bubble sort, multiple passes through the array are also made. However, the values of successive pairs of adjacent elements are compared and, if found to be out of order, are interchanged. The first pass begins with a comparison of elements one and two and continues until elements five and six have been processed. The second pass begins with

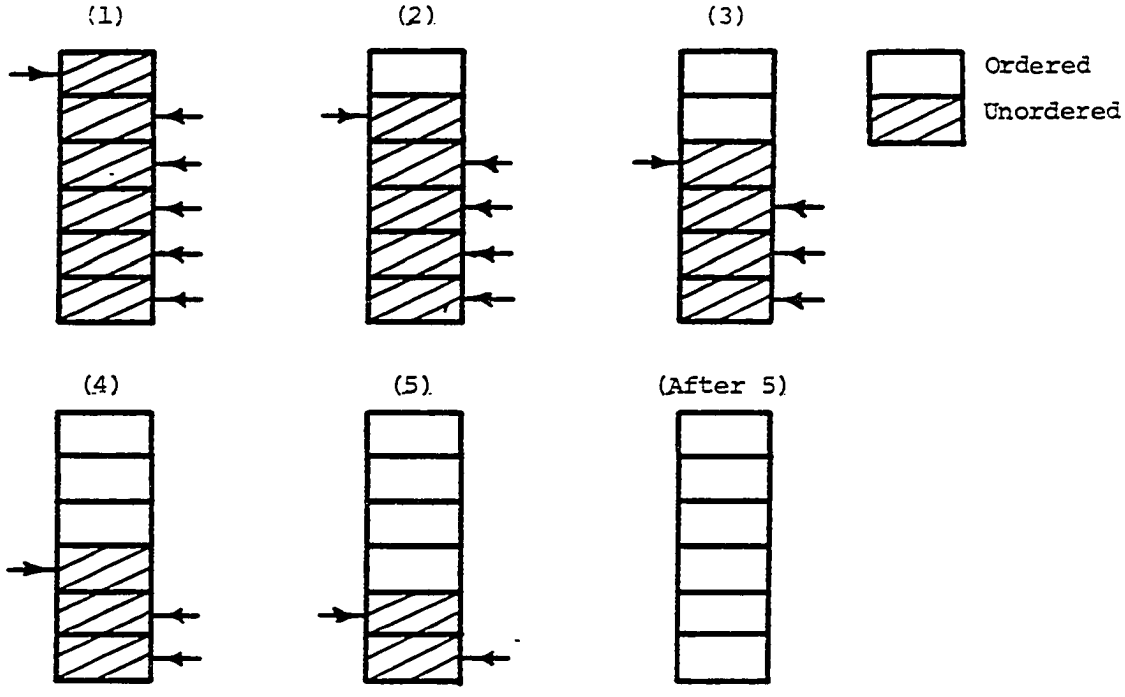


FIGURE 20. Graphical illustration of a selection sort

```
For I := 1 to MAX-1 do
   For J := I+1 to MAX do
       If X[I] > X[J]
          then begin
               TEMP := X[I];
               X[I] := X[J];
               X[J]:= TEMP
               end;
```

FIGURE 21. Pascal code for implementing a selection sort (ascending order)

the same comparison of elements one and two and continues through the array. Since the first pass "bubbled" the largest value down to cell

six, the second pass terminates with a comparison of cells four and five. The remaining passes repeat the process until the array is reordered. A bubble sort is illustrated in Figure 22. Note that it has the effect of reordering the array from bottom to top.

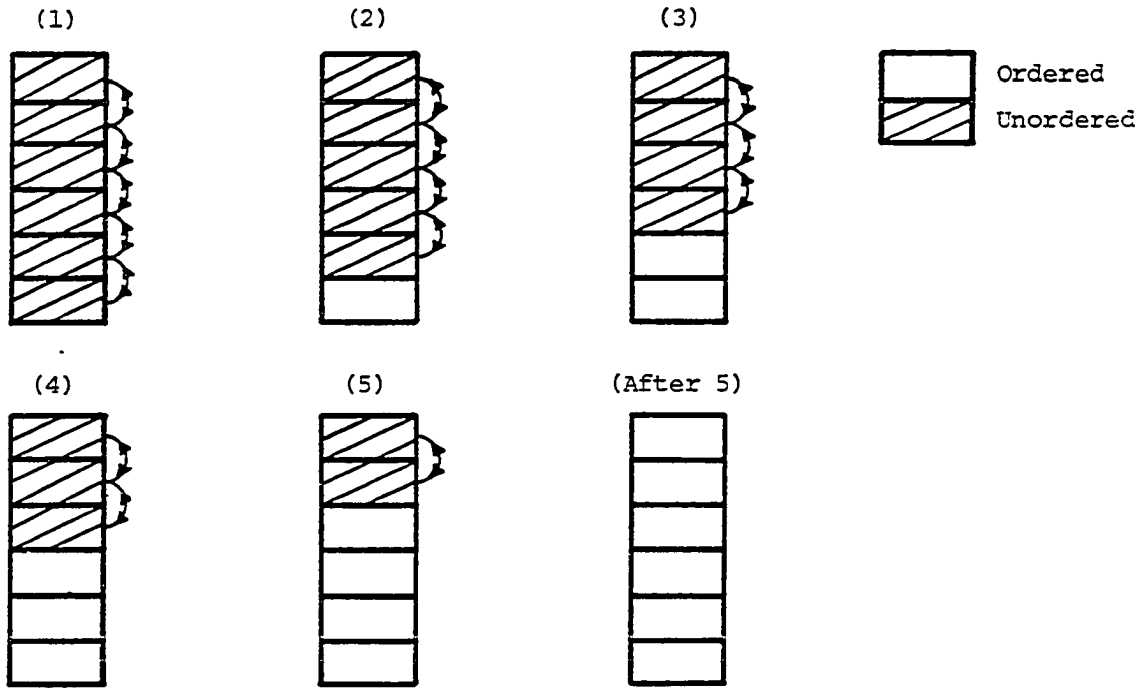


FIGURE 22. Graphical illustration of a bubble sort

The Pascal code for implementing a bubble sort is shown in Figure 23. Although two FOR loops are used to process the array, only one index variable is used to address the array cells. The outer FOR controls the number of passes that will be made through the array. The index variable of the inner FOR is used to denote which adjacent elements are being

compared at a given time (X[J] and X[J+1]). The computation MAX-I, where MAX is defined as the size of the array, determines how many comparisons will actually be made on each pass. At the end of the first pass the last cell will contain the largest value. At this point, comparisons between the last cell and all other cells become unnecessary. In fact, each subsequent pass through the array requires one less comparison than the previous pass since the largest value always "bubbles" down to the last comparison element.

```
For I := 1 to MAX-1 Do
   For J := 1 to MAX-I Do
      If X[J] > X[J+1]
      then Begin
           TEMP := X[J];
           X[J] := X[J+1];
           X[J+1] := TEMP
           End;
```

FIGURE 23. Pascal code for implementing a bubble sort (ascending order)

The protocols of the students' initial attempts to the ascending sort problem documented several solution features. These features were 1) correctness of solution in terms of syntax and logic, 2) correctness of solution in terms of logic only, 3) algorithm implementation (selection sort, bubble sort or indeterminate), 4) preservation of original element values, 5) efficiency in terms of the number of passes made through the array, 6) efficiency in terms of the number of comparisons per pass, 7) use of nested looping structures, 8) use of an

IF statement to compare element values and 9) use of assignment statements to interchange values were also recorded. A logically correct solution was defined to be a solution that contained two nested FOR loops, correct use of index variables to address array elements, and use of assignment statements to exchange element values. Appendix Tables D-3 and D-4 document the solution features of the students for the ascending sort problem.

Determining the bounds of the FOR loops on either the selection or the bubble sort is a difficult task for beginning students. Incorrect determination of bounds can result in too many or too few passes through the array, unnecessary or insufficient comparisons, or "out-of-range" runtime errors. Since incorrect bounds had the potential for causing the processing errors noted, the efficiency of the bounds was documented in the protocols.

The initial paper solutions of only four students (T12, T17, C06, C12) were syntactically and logically correct. Ten more students (T02, T04, T11, T13, T16, T18, C13, C14, C15, C16) wrote solutions that exhibited correct logic. Of the fourteen students with logically correct initial solutions, only three (T02, T04, C06) had no prior FORTRAN or Pascal programming experience. Algorithm selection was fairly evenly divided among the students. Eleven students (T02, T04, T10, T14, T16, T17, T18, C09, C11, C14, C16) attempted the selection sort and twelve (T09, T11, T12, T13, C02, C03, C05, C06, C12, C13, C15, C17) attempted the bubble sort. The algorithms of six students (T03, T05, T15, C08, C10, C18) could not be classified. A pattern between previous

programming experience and algorithm selection was not apparent from the data.

The two most obvious errors in the student's initial solution attempts were failure to preserve original values and failure to use nested looping structures. Seven students (T03, T09, T14, T15, C02, C03, C08) made no effort to preserve original values. Eleven students (T03, T05, T09, T15, C03, C05, C08, C09, C10, C17, C18) failed to use nested loops. The use of an IF statement to compare values and assignment statements to exchange values by nearly all students demonstrated that the functions of these statements were fairly well-understood.

Very few students wrote bound expressions that, when executed, would efficiently process the entire array regardless of the array's initial values. Twenty-three students issued inefficient bounds for the FOR loop that controlled the number of passes that would be made through the array. Of the six students who issued efficient bounds, only one (T02) had no previous programming experience. Twenty-seven students wrote inefficient bound expressions for the loop that controlled the number of comparisons per pass. Again, the two students (T14, T17) who did issue efficient bounds for the comparisons had previous programming experience.

Consistent with earlier programming protocols, the online programming protocols for the ascending sort problem documented 1) the number of initial compilation attempts prior to obtaining the first executable version of the solution, 2) the number of total compilations attempted across all program versions, 3) the number of unique executable versions, 4) specific programming problems encountered by the students,

5) a MINIPAS completion time if a correct solution was generated, and 6) the student's final solution.

Students averaged 3.14 initial compilation attempts on the ascending sort problem. The standard deviation for the initial compilation attempts was 2.36. These statistics were much lower than those found for the three-variable sort problem. The average number of total compilations attempted was 6.93 with a standard deviation of 5.64. The mean number of executable versions per student was 2.86 with a standard deviation of 1.83 versions.

The modifications made by each student in attempting to program a correct solution to the ascending sort problem in MINIPAS were analyzed to determine the specific programming problems encountered by each student. Other than syntax errors, the most common code modifications included alterations to the bounds of the looping indices and the addition or deletion of a looping structure. Nine students (T03, T05, T10, C05, C09, C10, C11, C17, C18) consistently added or deleted looping structures in attempting to develop a solution that would properly process the array. Bounds on the looping structures were troublesome for thirteen students (T02, T03, T05, T10, T16, T18, C02, C03, C05, C13, C14, C15, C16). Syntax was a major problem for eight students (T02, T03, T11, T15, C03, C08, C11, C18). Six students (T02, T09, T18, C02, C03, C11) struggled with the formats of IF statements. Only three students (T02, T14, T18) made modifications to the assignment statements that exchanged values.

In addition to the four students who had correct paper solutions, ten more students (T04, T11, T14, T16, T18, C05, C11, C14, C15, C16) generated correct solutions using the MINIPAS compiler. The final solutions of three students (T13, C09, C13) were logically correct, but due to an incorrect bound on one of the FOR loops didn't completely process the entire array. Of the fourteen students who wrote logically correct solutions only three (T04, C05, C06) had no previous programming experience.

Protocols of the final solutions also showed that seven students (T03, T15, C02, C05, C09, C10, C18) who hadn't used nested loops in their original solutions used them in their final solutions. One of the seven students who failed to preserve values on the initial attempt did so on the final solution attempt (T14). Besides the two students who initially used efficient bounds on the inner FOR loop, three more students (T04, T16, T18) used them on their final solutions.

The number of students opting to use either the selection or the bubble sort did not change dramatically between initial and final solution attempts. Twelve students tried to implement a selection sort and thirteen students tried to implement a bubble sort on their final solutions. Four students (T03, T05, T15, C10) whose algorithms on the initial solution attempts could not be classified made some progress in implementing either the selection or the bubble sort on their final attempts. Three students (T09, C08, C03, C18) wrote final solutions whose algorithms could not be classified.

Treatment group comparisons on the ascending sort problem

Chi-square statistics for nine solution characteristics were computed to determine whether there was any performance difference between the treatment groups on the students' initial solution attempts to the ascending sort problem. These nine characteristics included correctness of solution and logic, algorithm implementation (selection or bubble sort), preservation of original values, use of specific types of language statements (nested loops, IF and assignment statements), and efficiency of boundary expressions. The frequency counts used in all but one of the tests were the number of solutions in each group that exhibited the characteristic versus the number that did not. For the algorithm characteristic, indeterminate solutions were ignored and only the number of solutions containing the bubble or selection sort algorithms were used. Frequencies used in the chi-square tests are reported in Tables 8.

Although no statistically significant differences were found for any of the nine characteristics on the initial solution attempts, the chi-square statistic for type of algorithm attempted approached significance (chi-square = 2.11, df = 1, $p < .15$). Seven students in the MEMOPS treatment group initially attempted the selection sort as compared to only four students in the NON-MEMOPS control group. Eight students in the control group attempted the bubble sort as compared to only four in the treatment group.

T-tests comparing the treatment groups on number of MINIPAS compilation attempts and number of unique program versions were conducted

TABLE 8. Number of students exhibiting selected solution features in their initial solution attempts to the ascending sort problem

	Solution Features								
Groups	1	2	3	4	5	6	7	8	9
Treatment (n=14)	2	8	(S) 7 (B) 4	10	10	14	14	4	2
Control (n=15)	2	6	(S) 4 (B) 8	12	8	15	14	2	0

Solution Features:
1 Syntactically and logically correct solutions
2 Logically correct solutions
3 Solutions attempting to implement the selection sort (S) and bubble sort (B)
4 Solutions with code demonstrating knowledge of the principle concerning preservation of values
5 Solutions containing nested loops
6 Solutions containing IF statements
7 Solutions containing assignment statements
8 Solutions exhibiting an efficient number of passes through array
9 Solutions exhibiting an efficient number of comparisons per pass

to determine whether there was a difference in programming performance on the problem. The means and standard deviations used in performing the t-tests are shown in Table 9. There was not a statistically significant difference between the groups on number of initial compilation attempts, number of total compilation attempts, and number or unique versions attempted. No significant differences between groups on types of programming problems were apparent either.

Summary information for the two treatment groups on the final solution attempts to the ascending sort problem is presented in Table 10.

TABLE 9. MINIPAS history statistics for the ascending sort problem

Groups	MINIPAS History Features						
	Initial Compilations		Total Compilations		Unique Versions		Code Modification
	Mean	S.D.	Mean	S.D.	Mean	S.D.	
Treatment (n=14)	3.14	2.25	7.36	7.07	2.79	1.76	(A) 4 (D) 3 (B) 3 (E) 3 (C) 6 (*) 1
Control (n=15)	3.13	2.53	6.53	4.10	2.93	1.94	(A) 5 (D) 3 (B) 6 (E) 0 (C) 7 (*) 2

Code Modifications:
(A) Syntax
(B) Loops added or deleted
(C) Bounds on loops
(D) If statement
(E) Swapping components
(*) Unique changes

The nine characteristics used in comparing the groups were the same as those used for the initial attempts. Nonsignificant chi-square statistics suggested that the final solution attempts of students in the MEMOPS group did not differ statistically from the final attempts of the students in the NON-MEMOPS control group. The pattern concerning algorithm implementation noted for the initial attempts was not as prominent on the final attempts.

TABLE 10. Number of students exhibiting selected solution features in their final solution attempts to the ascending sort problem

	Solution Features								
Groups	1	2	3	4	5	6	7	8	9
Treatment (n=14)	7	2	(S) 7 (B) 6	11	12	14	14	4	5
Control (n=15)	7	2	(S) 5 (B) 7	11	12	15	15	2	0

Solution Features:
1 Syntactically and logically correct solutions
2 Logically correct solutions
3 Solutions attempting to implement the selection sort (S) and bubble sort (B)
4 Solutions with code demonstrating knowledge of the principle concerning preservation of values
5 Solutions containing nested loops
6 Solutions containing IF statements
7 Solutions containing assignment statements
8 Solutions exhibiting an efficient number of passes through array
9 Solutions exhibiting an efficient number of comparisons per pass

Summary of posttest 2 findings

The second posttest was divided into two parts and each was scored separately. For both parts of the test, the mean scores of the students in the MEMOPS treatment group did not differ significantly from the scores of the students in the NON-MEMOPS control group. Protocols of the students' attempts to generate Pascal code for two of three programming tasks suggested potential differences in the way the students approached these tasks.

Twenty-five of the twenty-nine students were able to write logically correct solutions to the comparison problem. No differences between the

treatment groups were apparent from studying the student protocols of the comparison problem. It was noted, however, that students with previous FORTRAN or Pascal programming experience wrote syntactically correct solutions whereas the students with little or no programming experience wrote solutions that were logically correct, but contained minor errors in syntax.

The reversal and ascending sort problems proved to be more challenging for all of the students. Only half of the students wrote logically correct solutions to the reversal problem. Furthermore, a pattern in algorithm selection was noted for the students in the MEMOPS treatment group. For the solution attempts containing classifiable algorithms, a majority of the treatment group students attempted the single index algorithm. A similar pattern was not apparent for the students in the NON-MEMOPS control group. Bounds on the FOR loop proved to be troublesome to all students regardless of prior programming experience. In spite of the demonstrated ability to exchange values on previous exercises, seven students on both the reversal and ascending sort problems failed to use this technique.

A difference between the treatment groups in initial algorithm implementation for the ascending sort was also suggested. For the MEMOPS treatment group, the number of students initially attempting to implement the selection sort was nearly twice the number of students who attempted the bubble sort. An opposite pattern was true for the NON-MEMOPS control group as more of them chose to initially implement the bubble sort over

the selection sort. These patterns were not as evident, however, on the final solution attempts.

Summary

In this chapter, a description of students' behavior as they progressed through a series of Pascal programming experiences was presented. The difficulties students encountered as well as procedural approaches they used were explored. The students were tracked both individually and in groups.

Prior to the study, approximately half of the students had engaged in some programming activities. As an initial activity of this study, half of the experienced and half of the inexperienced programmers were exposed to a manipulative computer model (MEMOPS) which was designed to facilitate the learning of programming. Students were classified by prior programming experience as well as MEMOPS exposure and the protocols of the resulting groups were then contrasted. The subjects in this study consisted of one female and thirty-five male students.

CHAPTER V: SUMMARY, DISCUSSION, RECOMMENDATIONS AND CONCLUDING REMARKS

Summary

The goals of this study were threefold. The first was to document programming behavior in an attempt to learn more about the novice's preconceptions and intuitions about programming. The second was to evaluate the effects on student learning of a manipulative computer model used prior to formal instruction on computer programming. The third and final goal was to evaluate the use of protocols as tools in studying programming behavior.

The study was conducted using a posttest quasi-experimental design. A matching strategy based upon responses to questionnaire items was used to assign students to pairs. After the students had been matched, one member of each pair was randomly assigned to the treatment group and the other member was assigned to the control group. Next, the students in the treatment group completed a series of "programming-like" tasks using a manipulative model of computer memory operations, while students in the control group worked through a placebo lesson. Instruction and programming activities that focused on elementary memory operations and Pascal declaration, assignment, and IF statements followed. The first posttest was then administered. Later in the semester, after students had been formally introduced to array data structures and Pascal looping constructs through classroom presentations and programming activities, a second posttest was administered.

The findings of this research must be considered tentative. The sample size was small and the analyses were primarily post hoc. However, the findings should serve as directions for future investigations. Based upon the data collected and the analyses performed, the findings were:

1. Novice programmers did not intuitively apply an accurate model of computer memory operations.

2. When the novice was faced with a challenging programming task that required the creative application of programming knowledge, newly learned techniques were frequently neglected.

3. The syntax as well as the semantics of computer statements must be learned by beginning programmers. Once learned, there was an initial tendency to undergeneralize followed by a tendency to overgeneralize the functions of statements.

4. Compared to the number of changes that students made in syntax and logic, the algorithm or overall approach to a problem was changed much less frequently.

5. Novice programmers appeared to expect computers to process information in a manner similar to the way humans process information.

6. The choice of algorithms of the treatment students was significantly influenced by their MEMOPS experience.

7. Posttest scores measuring syntax accuracy, the ability to

hand-execute a Pascal program, and the ability to successfully program a solution to the given tasks were not affected by the MEMOPS experience.

8. The conflict between the left-to-right direction of the MEMOPS MOVE statement and the right-to-left direction of the Pascal assignment statement was a problem for several of the students in the treatment group.

Discussion

This discussion is divided into three subsections, one for each of the study's primary areas of investigation. In the first subsection the findings that illustrate some of the preconceptions that novices have about programming are discussed. The effects of the MEMOPS experience that were found by comparing the programming performances of the two experimental groups are discussed in the second subsection. In the third subsection the usefulness of programming histories in examining novice programming behavior is discussed. The format for these subsections will be to restate the findings and discuss each independently.

Preconceptions of novices and the learning of programming concepts

1. Novice programmers did not intuitively apply an accurate model of computer memory operations.

The MEMOPS lesson was designed to provide a programming-like environment that would give novices an early opportunity to explore their own intuitive models of memory. These models were first challenged as the students attempted to solve the MEMOPS swap task. Nearly all of the less experienced programmers attempted to exchange element values by moving the value of the first cell into the second, and then moving the value of the second cell into the first. Similarly, some of the nonexperienced programmers in the control group initially implemented a similar algorithm for the Pascal swap problem. The fact that many of the novices in both groups failed to preserve a value before performing the exchange indicates that they possessed an inadequate model of computer memory operations. This finding, although not particularly profound, does verify that the treatment in this study, the MEMOPS lesson, did force the students to test their existing models of memory, whatever those models may have been.

2. When the novice was faced with a challenging programming task that required the creative application of programming knowledge, newly learned techniques were frequently neglected.

The novices in the treatment group learned the technique for exchanging values and used it repeatedly in completing the MEMOPS sorting activities. These same students also successfully solved the Pascal swap task with only minor language translation problems (the "wrong-way" error). Yet, their initial programming efforts on the more difficult problems indicated that they did not automatically nor consistently

generate the sequence of assignment statements that correctly preserved and exchanged values. The students often temporarily lost the mechanics involved in translating this thought into actual Pascal programming statements. The programming behaviors for the control students on the Pascal swap problem and subsequent programming tasks reflected a similar inconsistency.

Two additional behaviors documented in the protocols support the finding that novices neglected to use previously learned techniques as programming tasks became more difficult. First, although many students used the "keeps best" technique to locate a single value for the two hidden selection tasks, less than half of the students used it to determine the order of the unknown values in the hidden sorts. Second, even though the treatment students had performed the MEMOPS swap using the efficient technique of preserving a single value before performing the exchange, they chose to implement the less efficient technique of copying both values into unused cells in their final Pascal solutions. These noted inconsistencies in programming behavior support Sheil's view that "the difficulty of programming is that it is a very nonlinear function of the size of the problem."

Sheil (1981) has objected to characterizing programming as a "linear aggregation of difficulties" (p. 117). The inconsistent programming behaviors of the novices across several of the different programming tasks indicate that the students not only failed to utilize previously acquired techniques, but also failed to develop algorithms even though knowledge of the necessary language statements was present. The

solutions to the swap and comparison problems were fairly straightforward in that a naive understanding of the function of one or more primitive Pascal programming statements was apparently all that was required to generate a solution for either of these tasks if one was not immediately known. In other words, the solutions to these two problems were not far removed from the underlying transactions of the coding statements themselves. For the swap problem, all that was required to solve the problem was a simple understanding of assignment and READLN statements. For the 2-array comparison problem, similar knowledge of the functions of FOR loops, IF statements, and index variables in addressing array elements was evidently enough to generate a correct solution.

In contrast, the programming solutions to the three-variable sort, reversal, and ascending sort problems were much more challenging. Although one might expect some difficulty programming solutions to these problems because they required more complex algorithms, what was unexpected was the fact that students who had successfully solved the swap or comparison problem minutes before failed to write code that indicated they recognized that these same tasks were features of the solutions to the more difficult problems. A naive understanding of primitive Pascal statements as well as just having solved a problem that was a subtask of the present problem were not enough to help the students generate algorithms that would solve the more difficult problems. These programming tasks were not slightly more difficult for just a few students, as one would predict if programming could be characterized as a

"linear aggregation of difficulties", they were substantially more difficult for many of the students in both groups.

3. The syntax as well as the semantics of computer statements must be learned by beginning programmers. Once learned, there was an initial tendency to undergeneralize followed by a tendency to overgeneralize the functions of statements.

In the MEMOPS swap, nearly all of the nonexperienced programmers used the Z location to temporarily preserve a value. This behavior was not unexpected since the students had been required to use Z in the previous two tasks to store an array's smallest or largest value. Students with prior programming experience, however, used the third element of the array to preserve the value. Unlike the novices, the more experienced programmers appeared to possess more flexible knowledge regarding language statements and used the closest available location for storing the value.

Three unique behaviors documented in the three-variable programming protocols illustrate the difficulty novices had in overgeneralizing language statements. One novice attempted to use a READLN statement in place of assignment statements to reorder values (READLN (A,C,B)). Another student creatively tried to assign the order of the values to a single variable (D := A,C,B). Several students attempted to use compound logical expressions such as IF A>B>C ... to determine the relational order of values. Each of these behaviors indicates that novices often

tend to overgeneralize the functions of language statements and that the limitations of a statement is a source of difficulty.

4. Compared to the number of changes that students made in syntax and logic, the algorithm or overall approach to a problem was changed much less frequently.

Of the students whose initial solution algorithms could be classified, only two switched to a different algorithm for the swap problem, four switched for the three-variable sort, and one switched for the ascending sort problem. This finding provides some information about what students do after they correct their syntax errors and before they get their programs to work correctly. Since very few students switched algorithms, it would appear that they spend very little time re-examining their general approach to the problem by comparing it to alternative approaches. Instead, novices seemed to spend time trying to get their approach to work and only switched algorithms as a last resort. Novices do not appear to realize that algorithm development is the key to programming. To them, the mechanics of making the computer implement the algorithm is all encompassing.

In studying some of the mental processes that underlie the ability to solve verbal analogies, Sternberg (1986) found that students who could successfully complete the analogies spent their problem-solving time differently than the students who were unable to correctly complete the analogies. Specifically, the successful students spent more time initially thinking about the problem, "taking in information in order to

ensure that they had encoded the information richly and in detail" (p. 74). The findings of this study are consistent with Sternberg's. The novice programmers who were unsuccessful in generating correct solutions to the more difficult programming tasks spent much of their time modifying syntax and logic, trying to make an algorithm that was not initially well thought-out work.

5. Novice programmers appeared to expect computers to process information in a manner similar to the way humans process information.

A common analogy that programming instructors make is that programming a computer is like giving instructions to another human being. Sheil (1982) maintains that such an analogy "encourages its users to rely much more on their expectations of the hypothetical agent (the person following the instructions) than on the instructions themselves, whereas the mechanical reality is just the opposite" (p. 85). When instructing another person on a particular task, one relies on that person's existing knowledge and his ability to make inferences about information that has been left out. Unlike humans, computers cannot yet "fill in the gaps".

In using MEMOPS, students frequently entered versions of commands that appeared to require "human" types of understanding on the part of the computer. For example, novices used words like "swap" and "sort" in an attempt to solve the swapping and sorting tasks. They also used "MOVE 5" instead of "MOVE X[3] " where 5 was the value of X[3]. While these

might be inadvertent errors, the students seemed surprised by the computer's lack of understanding.

On the MINIPAS tasks, two algorithms that were attempted were close adaptations of common human processing practices. For the three-variable sort, both the "complex shuffle" and "isolate all cases" algorithms were better suited for humans than machines. The "complex shuffle" involved remembering which actions had been previously performed. The "isolate all cases" algorithm was an extension of determining specific relationships and then specifying independent actions. Many of the algorithms that were unclassifiable also appeared to feature human processes that did not easily adapt to computers, indicating that the students had not yet modified their thinking to accommodate the computer's limited capabilities.

Whereas humans rely on memory and the ability to effortlessly process conditional information, computers rely on repeating single processes many times. Processing tasks, such as selecting and sorting that humans ordinarily take for granted, must be unnaturally broken down into a simplified, well-specified repetitive process that the computer can handle. Recognizing the differences between human and computing processing techniques and modifying one's thinking to accommodate these differences may be critical to the learning of programming.

Effects of the MEMOPS experience on programming performance

6. The choice of algorithms of the treatment students was significantly influenced by their MEMOPS experience.

The most notable impact of the MEMOPS experience was its apparent influence on algorithm implementation for the more difficult programming tasks. The algorithms attempted by the treatment students for the three-variable and ascending sort problems were similar to the ones that the students had implemented in completing the MEMOPS tasks. In the visible sorting tasks, the treatment students reordered values by visually inspecting them and shuffling those that were out of order. The "complex shuffle" that these same students tried to implement appeared to be an attempt to translate this approach into a series of Pascal instructions. Sequentially filling arrays from top to bottom and the use of a "keeps best" technique to locate desired values were two additional features of the MEMOPS solution algorithms. These techniques are also features of the selection sort that many of the treatment students initially tried to implement for the Pascal ascending sort problem.

Whereas the treatment novices attempted the "complex shuffle" algorithm for the three-variable sort, the control novices tried to implement the "isolate all cases" algorithm. A comparison of the underlying features of these two algorithms reveals that the students of the two experimental groups may have been operating at different conceptual levels of problem representation. The "isolate all cases" algorithm can be characterized by a first-level analysis of the three-variable sort, namely that the solution must account for all possible value combinations. Not only does the "complex shuffle" algorithm demonstrate an awareness of this initial analysis, but also an awareness

of a second-level analysis that goes beyond the six specific instances to a class of instances. Although unduly complex, the shuffle algorithm reflects an attempt to seek a much more elegant solution to the three-variable sort problem.

The algorithms implemented for the reversal problem also suggest different levels of procedural reasoning ability. Although not statistically different, the proportion of treatment students choosing to implement the single-index algorithm for this problem was greater than the proportion of control students attempting this algorithm. The distinguishing characteristic between the single-index and two-variable algorithms is the naming scheme used to address the elements of the array. The use of a single index variable to address all elements of an array probably reflects a higher level of procedural reasoning ability and more flexibility regarding the functions of variables in programming.

As was true for the three-variable sort problem, students in the two experimental groups initially attempted different algorithms for the ascending sort problem. Whereas a majority of the treatment students initially attempted the selection sort, the control students attempted to implement the bubble sort. Just as the "complex shuffle" demanded a level of procedural reasoning that the treatment students did not yet possess, so did the selection sort. An incorrect starting value for a looping index in the bubble sort could likely result in an "out of bound" runtime error. In contrast, an incorrect starting value in the selection sort could result in "undoing" the ordering that had just been done.

This proved to be a catastrophic problem for the treatment novices and many were unable to resolve it.

The finding that the treatment students initially attempted more complex algorithms than did the control students for the more difficult problems parallels one of Mayer's findings. Using a similar experimental design, Mayer (1981) found significantly different programming performances between groups of students who had received a computer model before instruction and those who did not receive the model. More specifically, his findings indicated that the students receiving the model excelled in solving problems requiring far transfer and students who did not receive the model did as well or better on problems of near or moderate transfer. The programming behaviors reported in this study are consistent in that no performance differences (other than the "wrong-way" assignment problem) were found for the simpler problems, but significantly different algorithms were attempted by the students in the two groups for the more difficult programming tasks.

7. Posttest scores measuring syntax accuracy, the ability to hand-execute a Pascal program, and the ability to successfully program a solution to a given task were not affected by the MEMOPS experience.

The types of questions that were presented on the two posttests were typical of those that many programming instructors use to evaluate programming knowledge. These questions required the student to generate Pascal code that would perform selected programming tasks, identify

illegal array declarations and run-time errors caused by inappropriate index values, and trace the execution of a program and state the final values stored in an array. The posttest scoring procedures that were used were also typical of those utilized by programming instructors. Students were awarded full credit for syntactically correct responses and partial credit for the presence of certain "desirable" solution features. A comparison of the posttest scores of the students indicated no differences between experimental groups for syntax accuracy, the ability to hand-execute a Pascal program, and the ability to successfully program a solution to a given task. Traditional evaluation techniques were not useful in measuring the effect of the MEMOPS experience. Two factors, however, may have had a moderating effect on posttest performance.

One of the factors that may have masked potential differences between the experimental groups was the design of the MINIPAS programming environment. Like MEMOPS, MINIPAS displayed variables and their values. It also displayed the program as it was being executed. By stepping through the program one statement at a time, the control students could have acquired an understanding of memory operations that the MEMOPS lesson was designed to promote. This factor could have raised their test scores.

The tendency of the treatment group to choose more complex algorithms may also have masked group differences. In the three-variable sort, reversal, and ascending sort problems more students in the treatment group attempted sophisticated algorithms than did students in the control group. Attempting such complex algorithms may have lessened

the treatment students' chances for producing successful solutions.

8. The conflict between the left-to-right direction of the MEMOPS MOVE statement and the right-to-left direction of the Pascal assignment statement was a problem for several of the students in the treatment group.

The finding that was most surprising to the designers of the MEMOPS lesson was the conflict that the treatment students experienced concerning assignment direction. Whereas these students failed to unconditionally transfer the efficient swap technique to the Pascal swap problem, they did impose the left-to-right direction rule of the MEMOPS MOVE statement onto the Pascal assignment statement. The implication of this finding addresses a very important issue regarding the design of models and simulations, which is the degree to which a model must remain true to the event it simulates.

All models and simulations, by their very nature, make concessions concerning reality. These concessions many times are a simulation's strengths in that by stripping away some of the noncritical, technical and superficial complexity of reality, they allow the user to focus upon what is fundamental to the object or process being modeled. The purpose of the MEMOPS lesson was to provide a manipulative model of computer memory that could be used to test the student's intuitive model of memory. Students did possess incorrect intuitions about copy operations, but the MEMOPS environment forced them to alter these models to accommodate the copy's destructive property. Furthermore, the MEMOPS

environment was consistent with the Pascal environment. As a result, this caused only minor sequencing problems for the treatment students on subsequent Pascal programming tasks.

In contrast, the direction of the Pascal assignment statement differs from the pattern of left-to-right direction which is predominant in our every-day lives. Unfortunately the MEMOPS MOVE instruction reinforced this pattern and failed to prepare students for the right-to-left direction of the Pascal assignment statement. The reinforcement of the intuitive left-to-right pattern undoubtedly hindered students' ability to write Pascal assignment statements and signaled a major design flaw in the MEMOPS lesson. In general, when designing models and simulations, one must be very careful not to unwittingly reinforce intuitive patterns that are contradictory to those of the domain under investigation that will later be encountered.

Usefulness of programming histories in studying programming behavior

The performance protocols that were developed from the initial solution attempts, online programming histories, and final solution attempts were useful in documenting aspects of programming behavior that might have otherwise been difficult to study. Had comparisons of programming performance been based on posttest scores rather than individually documented solution features, the effects of the MEMOPS lesson on algorithm implementation may not have been as transparent. Although not utilized as extensively in protocol documentation as had originally been planned, the online programming histories were beneficial in studying several aspects of programming behavior.

First, the online histories pointed out the severity of the assignment direction problem for the treatment novices on the Pascal swap problem. Recall that only four students wrote initial solution attempts that exhibited left-to-right assignment. The programming histories, however, indicated that nine students at one time or another encountered this problem.

Second, the online histories indicated where students were spending their programming time. Most notable was the time spent by some students trying to get their initial solution attempt to compile. Although the syntax of their code on the initial written effort gave some indication that compilation would present a problem, the severity of this problem for these students was even more striking in the compilation histories. In addition, decoding the compilation messages was apparently a problem as students often recompiled code time and again without making any syntax modifications. Furthermore, correcting a syntax error found at the beginning of the program did not guarantee that the correction would be extended to other parts of the code that contained the same error.

Third, the online histories provided additional data that helped the researcher clarify the algorithm that the student was attempting to implement. Since several students had never programmed a computer before, the coding errors made on the written paper solutions sometimes made it difficult to determine what the student was attempting to do. This was particularly true for the three-variable sort problem. By forgetting to insert the statement separators (;) and BEGIN/ENDs in the IF statements, it was often initially difficult to decipher the student's

algorithm. However, in studying the histories of the modifications that students made to their code, the judges were often able to confirm or dispell some of their suspicions about the student's initial intentions.

Most importantly, the online histories often pointed out the key difficulties that prevented students from getting their algorithms to work. For the three-variable sort problem, the complexity of the "shuffle" algorithm became apparent as the students time and again successfully resequenced their comparison statements in an effort to figure out the relationship of the values. In attempting to implement the selection sort for the ascending sort problem, the difficulty in resolving the "unordering" predicament caused by an incorrect beginning index value became clear.

Recommendations

Based upon the tentative findings, the following recommendations are made:

1. The MEMOPS MOVE instruction should be changed to an instruction that implements a right-to-left direction of assignment. Two suggestions are to use either a LOAD or a FILL instruction. In addition to implementing the correct direction of assignment, the copy operation that these instructions connote would be more accurate than the action connoted by the MOVE instruction.

2. If this study is replicated, a panel of more than two judges with programming instruction experience should be used to 1) establish classification guidelines regarding the solution features that

are to be documented, 2) review those instances of behavior that are particularly difficult to classify to ensure consistency and 3) determine which characteristics reflect the acquisition of critical programming knowledge.

3. The MEMOPS and MINIPAS lessons might have been more effective had they been referred to more extensively during the instructional presentations. Because scheduling demanded that students from the two experimental groups attend the same lectures and lab sessions, the instructor and researcher had to refrain from making any references to the MEMOPS experience. References to the visible memory displayed in MINIPAS were kept to a minimum, since this lesson was designed to reinforce the treatment model.

4. Before attempts are made to replicate this study, the data that are recorded by the online programming histories should be rethought. Although viewing the compilation errors that the student received and seeing how the student responded to these errors was interesting, it was a time-consuming process. Efforts could be taken to develop a program such as the BUG FINDER (Bonar et al., 1982) that could locate the modifications between versions, thus reducing analysis time and human processing errors. However, the resources to develop and test such a program would be significant.

5. Due to the exploratory nature of the present investigation and the small sample size, the findings are considered to be tenta-

tive. Further research that replicates or refutes the findings of the present investigation should be conducted. In addition, the effects of the MINIPAS programming environment on novice programming performance should be studied independently of the MEMOPS lesson.

Concluding Remarks

Learning to program a computer is a very challenging and complex activity which appears to defy current instructional methods. It involves the acquisition of meaningful and nonintuitive information as well as a high degree of problem solving skill. The computer is a new learning environment for students in which previous experiences may provide an inadequate background. It requires a new set of skills and new ways of introducing those skills in meaningful contexts. Thus, the computer presents a unique opportunity for the psychologist, educator, linguist, and computer scientist to study and improve many facets of human learning and thinking. This opportunity is being presented at a most opportune time, when old learning theories and methods are being cast aside and new ones are being sought.

Motivation for studying the learning of computer programming comes from both inside and outside the discipline. From within the discipline, societal needs for employees skilled in the various aspects of information technology are growing rapidly and show strong indications of continuing to do so. For professionals in other disciplines, knowledge of human-computer interactions is anticipated

to transfer directly to many instructional problems that they face. Since the computer is both a subject to be studied and an aid in learning other subjects, its potential as a carrier of instructional innovations is unmatched. The need for improving instruction in computer programming and the probability that the knowledge acquired would have general applicability encouraged this study.

It is hoped that this study will contribute to the improvement of computer science education as well as the use of computer-based instructional artifacts in other areas. The major conclusion of this study is that the type of learning that results from simulations such as the MEMOPS lesson is of a high level. Yet, this learning defies traditional educational measurement. If this conclusion withstands the trials of investigation, the efforts will be well rewarded.

BIBLIOGRAPHY

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

Ahmann, J. S., & Glock, M. (1975). Evaluating pupil growth. Boston: Allyn and Bacon.

Allen, R. (1982). In B. Shneiderman and A. Badre (Eds.), Directions in human/computer interaction. Norwood, NJ: Ablex.

Atwood, M., & Ramsey, H. (1978). Cognitive structures in the comprehension and memory of computer programs: An investigation of computer degubbing (Technical Report TR-78-A21). Alexandria, VA: U.S. Army Research Institute for the Behavioral and Social Sciences.

Ausubel, D. (1968). Educational psychology: A cognitive view. New York: Holt, Rinehart, and Winston.

Ausubel, D. (1977). The facilitation of meaningful verbal learning in the classroom. Educational Psychologist, 12, 162-178.

Barnes, B., & Clawson, E. (1975). Do advance organizers facilitate learning? Recommendations for further research based on an analysis of 32 studies. Review of Educational Research, 45, 637-659.

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

Bayman, P., & Mayer, R. (1983). A diagnosis of beginning programmers' misconceptions of BASIC programming statements. Communications of the ACM, 26(9), 677-679.

Bok, D. (1985). Looking into education's high-tech future. Educom Bulletin, 20(3), 2-10, 17.

Bonar, J., Ehrlich, K., Soloway, E., & Rubin, E. (1982). Collecting and analyzing on-line protocols from novice programmers. Behavior Research Methods & Instruction, 14(2), 203-209.

Bonar, J., & Soloway, E. (1983). Uncovering principles of novice programming. ACM Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages, (pp. 10-13). New York: Association of Computing Machinery.

Bonham, G. (1983). Computer mania: Academe's inadequate response to the implications of the new technology [Letter to the Editor]. Chronicle of Higher Education, 26(5), 72-73.

Branch, R. (1973). The interaction of cognitive style with the instructional variables of sequencing and manipulation to effect achievement of elementary mathematics. Dissertation Abstracts International, 34, 4857A. (University Microfilms No. 74-2244).

Bransford, J. (1979). Human cognition. Monterey, CA: Wadsworth.

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

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

Brownell, W., & Moser, H. (1949). Meaningful vs. mechanical learning: A study in grade III subtractionn. Duke University Research Studies in Education, 8, 1-207.

Bruner, J. (1960). The process of education. New York: Vintage Books.

Bruner, J. (1966). Toward a theory of instruction. Cambridge, MA: Harvard University Press.

Bruner, J. (1973). Beyond the information given. New York: Norton.

Bruner, J. (1985). Models of the learner. Educational Researcher, 14(6), 5-8.

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

Clark, R. (1985a). Confounding in educational computing research. Journal of Educational Computing Research, 1(2), 137-148.

Clark, R. (1985b). The importance of treatment explication: A reply to J. Kulik, C-L. Kulik and R. Bangert-Drowns. Journal of Educational Computing Research, 1(4), 389-394.

Coombs, M., Gibson, R., & Alty, J. (1982). Learning a first computer language: Strategies for making sense. International Journal of Man-Machine Studies, 16, 449-486.

Dalbey, J., & Linn, M. (1985). The demands and requirements of computer programming: A literature review. Journal of Educational Computing Research, 1(3), 253-274.

Dijkstra, E. (1976). A discipline of programming. Englewood Cliffs, NJ: Prentice-Hall.

Du Boulay, B. (1986). Some difficulties of learning to program. Journal of Educational Computing Research, 2(1), 57-73.

Du Boulay, B., and O'Shea, T. (1981). Teaching novices programming. In M. Coombs and J. Alty (Eds.), Computing skills and the user interface. London: Academic Press.

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

Farnham-Diggory, S. (1972). Cognitive processes in education: A psychological preparation for teaching and curriculum development. New York: Harper and Row.

Fletcher, J., & Atkinson, R. (1972). An evaluation of the Stanford CAI Program in initial reading. Journal of Educational Psychology, 63, 597-602.

Gannon, J. (1976). An experiment for the evaluation of language features. International Journal of Man-Machine Studies, 8, 61-73.

Gould, J. (1975). Some psychological evidence on how people debug computer programs. International Journal of Man-Machine Studies, 7, 151-182.

Grant, E., & Sackman, H. (1967). An exploratory investigation of programmer performance under on-line and off-line conditions. IEEE Transactions on Human Factors in Electronics, 8(1), 33-48.

Green, C., & Barstow, D. (1978). On program synthesis knowledge. Artificial Intelligence, 10, 241-279.

Green, T. (1977). Conditional program statements and their comprehensibility to professional programmers. Journal of Occupational Psychology, 50, 93-109.

Hartley, J., & Davies, I. (1976). Preinstructional strategies: The role of pretests, behavioral objectives, overviews, and advance organizers. Review of Educational Research, 46, 239-265.

Jeffries, R. (1982). A comparison of debugging behavior of expert and novice programmers. Paper presented at the Annual meeting of the American Educational Research Association, New York.

Kearsley, G. (1977). Some conceptual issues in computer-assisted instruction. Journal of Computer-Based Instruction, 4(1), 8-16.

Kulik, J., Kulik, C., & Bangert-Drowns, R. (1985). The importance of outcome studies: A reply to Clark. Journal of Educational Computing Research, 1(4), 381-387.

Larkin, J., McDermott, J., Simon, D., & Simon, H. (1980). Expert and novice performance in solving physics problems. Science, 208, 1335-1342.

Lesh, R. (1976). The influence of an advanced organizer on two types of instructional units about finite geometrics. Journal for Research in Mathematics Education, 7(2), 82-86.

Love, L. (1977). Relating individual differences in computer programming performance to human information processing abilities. Dissertation Abstracts International, 38, 1443B. (University Microfilms No. 77-18,379)

Lucas, H., & Kaplan, R. (1976). A structured programming experiment. Computing Journal, 19, 136-138.

Mayer, R. (1975). Different problem-solving competencies established in learning computer programming with and without meaningful models. Journal of Educational Psychology, 67(6), 725-734.

Mayer, R. (1979a). Can advance organizers influence meaningful learning? Review of Educational Research, 49(2), 371-383.

Mayer, R. (1979b). A psychology of learning BASIC. Communications of the ACM, 22(11), 589-593.

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

Mayer, R. (1982). Contributions of cognitive science and related research in learning to the design of computer literacy curricula. In R. Seidel, R. Anderson, and B. Hunter (Eds.), Computer literacy. New York: Academic Press.

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

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

Novak, J. (1977). A theory of education. Ithaca, NY: Cornell University Press.

Papert, S. (1981). Mindstorms: Children, computers and powerful ideas. New York: Basic Books.

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

Perkins, D., Hancock, C., Hobbs, R., Martin, F., & Simmons, R. (1986). Conditions of learning in novice programmers. Journal of Educational Computing Research, 2(1), 37-55.

Resnick, L., & Ford, S. (1980). The psychology of mathematics learning. Hillsdale, NJ: Erlbaum.

Scandura, J., & Wells, J. (1967). Advance organizers in learning abstract mathematics. American Education Research Journal, 4, 295-301.

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

Sheil, B. (1982). Coping with complexity. In R. Kasschau, R. Lachman, and K. Laughery (Eds.), Information technology and psychology: Prospects for the future. New York: Praeger.

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

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

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

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

Shneiderman, B., Mayer, R., McKay, D., & Heller, P. (1977). Experimental investigations of the utility of detailed flowcharts in programming. Communications of the ACM, 20, 373-381.

Siegler, R. (1983a). Five generalizations about cognitive development. American Psychologist, 38(3), 263-277.

Siegler, R. (1983b). How knowledge influences learning. American Scientist, 71, 631-638.

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

Smith, J., & Hehusius, L. (1986). Closing down the conversation: The end of the quantitative-qualitative debate among educational inquiries. Educational Researcher, 15(6), 4-12.

Solomon, G., & Gardner, H. (1986). The computer as educator: Lessons from television research. Educational Research, 15(6), 13-19.

Soloway, E., Ehrlich, K., Bonar, J., & Greenspan, J. (1982). What do novices know about programming? In B. Shneiderman and A. Badre (Eds.), Directions in human-computer interactions. Norwood, NJ: Ablex.

Soloway, E., Bonar, E., & Ehrlich, K. (1983). Cognitive strategies and looping constructs: An empirical study. Communications of the ACM, 26(11), 853-860.

Sternberg, R. (1986). Inside intelligence. American Scientist, 74, 137-143.

Weinberg, G. (1971). The psychology of computer programming. New York: Van Nostrand Reinhold Company.

Weiser, M. (1982). Programmers use slices when debugging. Communications of the ACM, 25(7), 446-452.

Weissman, L. (1977). A methodology for studying the psychological complexity of computer programs. Dissertation Abstracts International, 37, 6233B.

West, L., & Fensham, D. (1976). Prior knowledge or advance organizers as effective variables in chemistry learning. Journal of Research in Science Teaching, 13, 297-306.

White, B. (1984). Designing computer games to help physics students understand Newton's laws of motion. Cognition and Instruction, 1(1), 69-108.

Wiedenbeck, S. (1985). Novice/expert differences in programming skills. International Journal of Man-Machine Studies, 23, 383-390.

Wittrock, M. (1966). The learning by discovery hypothesis. In L. Shulman and E. Keislar (Eds.), Learning by discovery: A critical appraisal. Chicago: Rand McNally.

Youngs, E. (1974). Human errors in programming. International Journal of Man-Machine Studies, 6, 361-376.

ACKNOWLEDGEMENTS

This study was reviewed and approved by the Iowa State Committee on the Use of Human Subjects.

A great many people helped to make this dissertation possible, and I would now like to thank a few of them. First and foremost, Dr. Rex Thomas for his guidance and nurturing throughout the writing of this dissertation. Second, Dr. Pete Boysen for making MINIPAS a reality. Third, the other members of my committee who have provided me with encouragement and words of wisdom at various times; Dr. Maribeth Henney, Dr. Cheryl Hausafus, Dr. William Miller, and Dr. Michael Simonson.

I am also indebted to the instructor, Warner Smidt, and the students who participated in this investigation. Gratitude is also extended to Bill Nimtz and his computer science students for helping to pilot test the MEMOPS and the MINIPAS lessons.

Finally, I would like to thank my husband, Steven, whose faith in my abilities and tireless support sustained me during the frustrating times.

APPENDIX A: QUESTIONNAIRE AND MATCHING CRITERIA RESULTS

To All Industrial Education 216 Students:

This semester you will be using some new instructional computing materials to learn about computers and computer programming. These materials were developed to help alleviate some of the problems and misconceptions that previous students have encountered. From time to time we will solicit your reactions to these materials.

The information requested on the attached questionnaire will be used to learn more about the background of students enrolling in introductory programming courses such as this one. It will also help us analyze any reactions you may have to the new instructional materials that you will be using. This information and any other data that are collected from you will be kept strictly confidential.

Thank you for your cooperation.

Note: All information provided on this questionnaire will be kept in strict confidence and will have no bearing in determining your course grade.

Name _____ Social Security No. _____

Age _____ Sex _____ Year in College _____

Major _____

1. What high school computer science courses have you taken? (Please describe the major activities of each.)

2. What college computer science courses have you taken? (Please describe the major topics covered in each course.)

3. What other experience have you had with computers? (List any course-related or job-related activities such as use of a statistical package for a statistics course, word-processor for writing papers, etc.).

4. If you have computer programming experience, please check all languages in which you have written programs.

_____ BASIC _____ Pascal _____ FORTRAN _____ COBAL _____ PL/1

_____ C _____ LOGO _____ Others (Specify: _____)

5. Is there a microcomputer available for your use in your home? _____ yes ____ no

6. Place a check beside all of the mathematics courses you took in grades 9-12.

___ Algebra I ___ Algebra II ___ Geometry ___ Calculus

___ General Mathematics ___ Business Mathematics ___ Trigonometry

___ Other (Specify:_____)

7. Please list all of the mathematics courses you have taken in college.

8. Place a check beside your college GPA.

____ 3.5 to 4.0	_____ 2.0 to 2.49
____ 3.0 to 3.49	_____ 1.5 to 1.99
____ 2.5 to 2.99	_____ Below 1.5

9. What grade do you expect to receive in this course (I ED 216)? Check only one.

___ A ___ B ___ C ___ D ___ F

10. Briefly state why you are taking Industrial Education 216.

TABLE A-1. Distribution of students who took a high school computing course by experimental group

Category	Experimental Groups	
	Control (n=18)	Treatment (n=18)
No course	10	13
Took a course	8	5
Chi-square = .48	Significance = .49	

TABLE A-2. Distribution of students who had previously taken a college computing course by experimental group

Category	Experimental Groups	
	Control (n=18)	Treatment (n=18)
No previous course	8	7
Computer literacy	1	3
Programming	9	8
Chi-square = 1.13	Significance = .57	

TABLE A-3. Distribution of students by experimental group and computing experience other than programming (word processing, drafting, statistical analysis)

Category	Experimental Groups	
	Control (n=18)	Treatment (n=18)
No experience	9	13
Experience	9	5
Chi-square = 1.05	Significance = .17	

TABLE A-4. Distribution of students by experimental group and highest level programming language used in writing computer programs

Category	Experimental Groups	
	Control (n=18)	Treatment (n=18)
None	7	7
BASIC/LOGO	3	3
FORTRAN	4	6
Pascal/PL1/Cobol	4	2
Chi-square = 1.07	Significance = .79	

TABLE A-5. Distribution of students by experimental group and highest level mathematics courses taken in college

Category	Experimental Groups	
	Control (n=18)	Treatment (n=18)
Algebra/Trigonometry or Business Math.	5	4
Calculus	13	14
Chi-square = 0.00	Significance = 1.00	

TABLE A-6. Distribution of students by experimental group and college grade point average

Category	Experimental Groups	
	Control (n=18)	Treatment (n=18)
Less than 2.50	12	10
2.50 - 3.0	3	6
Greater than 3.00	3	2
Chi-square = 1.38	Significance = .51	

TABLE A-7. Distribution of students by experimental group and expected course grade

Category	Experimental Groups	
	Control (n=18)	Treatment (n=18)
A	5	3
B	10	14
C	3	1

Chi-square = 2.17 Significance = .34

APPENDIX B: MEMOPS PROTOCOLS

Explanation of Initial Problem States for MEMOPS Sorting Tasks

The computer randomly generated the original values for the arrays of the MEMOPS sorting tasks. Therefore, the number of values initially out of order was not consistent for all students. Since the different states could potentially influence the solutions generated by the students, all possible problem-states were identified and documented in the MEMOPS protocols. (See solution feature 1 for Tasks 4, 5, 8, and 9 in Tables B-1 and B-2.)

The seven possible initial states are displayed below. Each state is defined by two characteristics, the number of values that are out of order and the relationships between the initial positions of the values and their final positions.

(1)	(2) 2 cell problem	(3) 3 cell problem	(4) 2-2 cell problem
1	2	2	2
2	1	5	1
3	3	3	3
4	4	4	5
5	5	1	4
(in order)	(2 cells out of order)	(3 cells out of order)	(4 cells out of order)

(5) 4 cell problem	(6) 3-2 cell problem	(7) 5 cell problem
5	3	3
1	1	1
3	2	4
2	5	5
4	4	2
(4 cells out of order)	(5 cells out of order)	(5 cells out of order)

Table B-1. Treatment group protocols for the visible MEMOPS tasks

		Solution features					
		Restarts		Task 3		Task 4	
ID	EXP	n	Tasks	1	2	1	2
T01	0	3	3,4,5	+	Z	2	+
T02	0	1	3	+	Z	3-2	+
T03	0	4	3,4,5	+	Z	5	+
T04	0	0			Z	2-2	-
T05	0	2	3,4	+	Z	3-2	+
T06	0	2	3,5	+	Z	4	+
T07	0	3	4	+		4	+
T08	B	1	3	+	X[3]	5	+
T09	B	2	3,4	-	Z	5	+
T10	B	4	3	-	X[3]	2-2	+
T11	F	1	4		X[3],X[4]	4	-
T12	F	0			Z	5	+
T13	F	1	4		X[3]	5	+
T14	F	2	2,4		X[3]	2-2	+
T15	F	2	1,3	-	X[3]	2	+
T16	F	0			X[3]	4	-
T17	P	0			X[3]	5	+
T18	P	0			X[3]	5	-

ID Student identifier
EXP Programming experience (0 none, B BASIC/LOGO, F FORTRAN, P Pascal)
Restart Features:
  n Total number of restarts for the visible tasks
  Tasks (1 Move smallest, 2 Move largest, 3 Swap, 4 Ascending sort, 5 Descending sort
Task 3 Solution Features:
  1 Swap error (+ MOVE X[1] to X[2], MOVE X[2] to X[1]; - MOVE X[1] to X[2])
  2 Memory cells used for preserving values
Task 4 and Task 5 Solution Features:
  1 Initial problem state (see preliminary appendix material)
  2 Sequential filling technique (+ used, - not used)
  3 Swapping technique (see Figure 11)

res

Task 4		Task 5		
2	3	1	2	3
+	2	3	-	3
+	3-2	2-2	+	2-2
+	2-2-3	5	+	5
-	2-2	2-2	+	2-2
+	3-2	3-2	+	3-2
+	4	4	+	4
+	4	5	+	2-4
+	5	2-2	+	4
+	5	2-2	+	2-2
+	2-2	3-2	-	3-2
-	4	3	-	3
+	5	3-2	-	3-2
+	5	(given in order)		
+	2-2	3	+	3
+	2	4	+	4
-	4	4	+	4
+	2-2-2-2	2-2	+	2-2
-	5	4	+	4

.ng sort)

Table B-2. Treatment group protocols for the hidden MEMOPS tasks

				Solution features						
		Restarts		Task 8						
ID	EXP	n	Tasks	1	2	3	4	5	6	1
T01	0	6	6,8,9	2-2	CM	-	-	+	2-2	4
T02	0	2	8,9	4	CM	+	+	-	4	3-2
T03	0	3	9	3-2	CM	-	+	+	3-2	4
T04	0	0		2	CM	-	+	+	2	2
T05	0	1	6	2	CM	-	+	+	2	4
T06	0	0		2-2	CM	+	+	+	2-2	3-2
T07	0	1	8	4	CM	-	-	+	4	4
T08	B	0		4	CM/CM	+	+	+	3-2	4
T09	B	0		5	CM/CM	+	+	+	5	2
T10	B	5	7,8	4	CM	-	-	+	4	3-2
T11	F	3	8,9	4	CM	-	-	+	4	3
T12	F	0		3-2	CM	-	+	+	5-2	2
T13	F	0		5	CM/CM	+	+	+	2-2-2-2	3
T14	F	0		3	CM	-	+	+	3	4
T15	F	2	8,9	4	CM	-	+	+	4	4
T16	F	0		5	CM	+	+	+	5	5
T17	P	0		2	CM	+	+	+	2	2-2
T18	P	1	8	4	CM/CM	+	+	-	2-2-2	4

ID Student identifier
EXP Programming experience (0 none, B BASIC/LOGO, F FORTRAN, P Pascal)
Restart Features:
n Total number of restarts for the hidden tasks
Tasks (6 Move smallest, 7 Move largest, 8 Ascending sort, 9 Descending sort)
Task 8 and Task 9 Solution Features:
1 Initial problem state (see preliminary appendix material)
2 Sequencing of COMPAREs and MOVEs
(CM COMPAREs all, MOVEs all; CM/CM MOVEs interspersed between COMPAREs; M MOVEs
3 Keeps-best technique (+ used, - not used)
4 Closure (+ attained, - not attained)
5 Sequential filling technique (+ used, - not used)
6 Swapping technique (see Figure 11)

Task 9					
1	2	3	4	5	6
4	M	-	-	-	4
3-2	CM	+	+	+	5
4	CM	-	+	+	3-2
2	CM	-	+	+	2
4	CM	-	+	+	4
3-2	CM	+	+	+	4-2
4	CM	-	-	+	4
4	CM/CM	+	+	+	2-3
2	CM/CM	+	+	+	2
3-2	M/CM	-	+	+	5
3	CM	-	-	+	3
2	CM	+	+	+	2
3	CM/CM	+	+	+	2-2
4	CM	-	+	+	4
4	CM	-	+	+	4
5	CM	+	+	-	5
2-2	CM/CM	+	+	+	2-2-2
4	CM/CM	+	+	-	2-2-2-2-2

; M MOVEs only)

APPENDIX C: POSTTEST 1 AND POSTTEST 1 PROTOCOLS

Name _____ SS # _____

PROBLEM I. Part A:

Part of a Pascal program that swaps the values of variables A and B is shown below. Complete the program by adding the necessary Pascal statements. You should not need to declare any other variables or insert any other READLN or WRITELN statements to complete the program.

```
Program Problem1 (Input,Output);
Var  a,b,c,d : Integer;
Begin
   Writeln ('Enter a whole number:');
   Readln (a);
   Writeln ('Enter another number:');
   Readln (b);
   (* Add your code below to perform the swap.*)
```

```
  Writeln ('The new value for a is ',a);
  Writeln ('The new value for b is ',b)
End.
```

PROBLEM 1. Part B:
Once you are satisfied with your answer above, logon to a VAX. At the $ prompt type PROB1. You will automatically enter the MINIPAS program. Select the WRITE PROGRAMS option from the menu. A reasonable facsimile of the above program will appear. Insert your code (the SAME statements that you have written above) into the program. Ask a monitor to verify that you have done this and then proceed to compile, run, and edit your program as necessary until you're satisfied that you have a program that solves the given problem. Exit MINIPAS. When the $ prompt appears turn in this problem sheet and get the problem sheet for the second problem.

Name_____ SS #_____

PROBLEM 2. Part A:

Part of a Pascal program that requests the user to enter three numbers in any order and then sorts these numbers from SMALLEST to LARGEST is shown below. Complete the program by adding the necessary Pascal statements. Remember, the program should store the numbers in whatever order they are entered and then reorder them so that the number with the smallest value is in variable A and the number with the largest value is in variable C.

```
Program Problem2 (Input,Output);
Var  a,b,c,d : Integer;
Begin
   Writeln ('Enter 3 numbers:');
   Readln (a,b,c);









   Write ('The ordered values from smallest ');
   Writeln ('to largest are ',a,b,c)
End.
```

PROBLEM 2. Part B:
At the $ prompt type PROB2. You will automatically enter the MINIPAS program. Select the WRITE PROGRAMS option from the menu. A reasonable facsimile of the above program will appear. Insert your code (the SAME statements that you wrote above) into the program. Ask a monitor to verify that you have done this and then proceed to compile, run and edit your program as necessary until you are satisfied that you have a program that solves the given problem. Exit MINIPAS and logoff VAX. Turn in your username and this problem sheet.

Table C-1. Treatment group protocols for the swap problem

		Initial solution						Features MINIPAS		
ID	EXP	1	2	3	4	5	6	1	2	3
T01	0	-	-	P	0	I		1	10	10
T02	0	-	+	P	2		+	2	4	3
T03	0	-	-	P	0	I		7	7	0
T04	0	+	+	P	2			1	1	1
T05	0	-	+	M	1			1	6	4
T06	0	-	+	M	2			8	8	2
T07	0	-	+	PM	2		+	3	4	2
T08	B	-	+	P	2	W	+	3	4	2
T09	B	-	+	M	1			10	16	4
T10	B	-	-	P	2		+	3	8	6
T11	F	-	+	PM	2			10	11	2
T12	F	+	+	P	2			1	1	1
T13	F	-	+	P	2			5	5	1
T14	F	+	+	P	1			1	1	1
T15	F	-	-	P	2	R,W,I		17	17	1
T16	F	+	+	P	2			1	1	1
T17	P	+	+	P	1			1	1	1
T18	P	+	+	P	1			2	2	1

ID Student identifier
EXP Programming experience
(0 none, B BASIC/LOGO, F FORTRAN, P Pascal)
Initial Solution Features:
1 Syntax and logic (+ correct, - incorrect)
2 Logic only (+ correct, - incorrect)
3 Type of statements (P Pascal, M MEMOPS, PM both)
4 Number of memory cells used to preserve values
5 Unnecessary code (I IFs, R READLNs, W WRITELNs)
6 WRONG-WAY assignment error

MINIPAS History Fea
1 Number of init
2 Number of tota
3 Number of uniq
4 Programming Pr
(A syntax,
D ordering
5 Completion tim
Final Solution Feat
1 Syntax and log
2 Logic only (+
3 Number of memo

INIPAS history			Final solution		
3	4	5	1	2	3
10	B,D	36.37	+	+	2
3	B	13.50	+	+	2
0	A,C	-	-	-	2
1		4.27	+	+	2
4	D,B	32.70	+	+	2
2	A,B	-	-	+	2
2	B	14.25	+	+	2
2	B	16.72	+	+	2
4	A,B	28.17	+	+	2
6	B	35.38	+	+	2
2	A,B	17.68	+	+	2
1		2.80	+	+	2
1	A	11.80	+	+	2
1		2.47	+	+	1
1	A,C	33.09	+	+	2
1		5.37	+	+	2
1		6.35	+	+	1
1		2.83	+	+	1

story Features:
r of initial compilations
r of total compilations
r of unique program versions
amming Problems:
syntax, B WRONG-WAY assignment, C logic,
ordering, E error A=B, B=A)
etion time (in minutes)
tion Features:
x and logic (+ correct, - incorrect)
only (+ correct, - incorrect)
r of memory cells used to preserve values

Table C-2. Control group protocols for the swap problem

		Features									
		Initial solution						MINIPAS histo			
ID	EXP	1	2	3	4	5	6	1	2	3	
C01	0	-	+	P	2			1	1	1	
C02	0	-	-	P	0	I		20	20	0	
C03	0	-	+	P	2			5	6	1	
C04	0	-	-	P	0	I		23	23	0	A,
C05	0	-	-	P	2	R	+	4	10	5	A,E
C06	0	-	+	P	2		+	3	4	2	
C07	0	-	-	P	2	I		5	15	9	A
C08	B	-	-	P	2			16	27	6	A,
C09	B	+	+	P	2			1	1	1	
C10	B	+	+	P	2			1	1	1	
C11	F	-	+	P	2			2	2	1	
C12	F	+	+	P	2			1	1	1	
C13	F	-	+	P	2		+	2	3	2	
C14	F	+	+	P	1			2	2	1	
C15	P	+	+	P	2			2	2	1	
C16	P	+	+	P	1			2	2	1	
C17	P	+	+	P	2			6	6	1	
C18	P	+	+	P	2			1	1	1	

ID Student identifier
EXP Programming experience
(0 none, B BASIC/LOGO, F FORTRAN, P Pascal)
Initial Solution Features:
1 Syntax and logic (+ correct, - incorrect)
2 Logic only (+ correct, - incorrect)
3 Type of statements (P Pascal, M MEMOPS, PM both)
4 Number of memory cells used to preserve values
5 Unnecessary code (I IFs, R READLNs, W WRITELNs)
6 WRONG-WAY assignment error

MINIPAS History Fe
1 Number of ini
2 Number of tot
3 Number of uni
4 Programming p
(A syntax,
D orderin
5 Completion ti
Final Solution Fea
1 Syntax and lo
2 Logic only (+
3 Number of mem

S history		Final solution		
4	5	1	2	3
	9.05	+	+	2
A,C	-	-	-	0
A	12.83	+	+	2
A,C,D,E	-	-	-	0
A,B,C,D,E	47.18	+	+	2
B	6.70	+	+	2
A,B,D	-	-	-	2
A,C,D,E	-	-	-	0
	2.18	+	+	2
	3.93	+	+	2
	6.77	+	+	2
	2.85	+	+	2
B	9.25	+	+	2
	3.75	+	+	1
	4.58	+	+	2
	8.55	+	+	1
A	8.00	+	+	2
	6.10	+	+	2

istory Features:
er of initial compilations
er of total compilations
er of unique program versions
ramming problems:
A syntax, B WRONG-WAY assignment, C logic,
D ordering, E error A=B, B=A)
letion time (in minutes)
ution Features:
ax and logic (+ correct, - incorrect)
c only (+ correct, - incorrect)
er of memory cells used to preserve values

Table C-3. Treatment group protocols for the three-variable sort problem

		Features											
		Initial solution							MINIPAS history				
ID	EXP	1	2	3	4	5	6	7	1	2	3	4	5
T01	0	-	-	0	+	-	+	0	1	8	2	A	-
T02	0	-	-	3	+	-	+	CI	4	16	8	B,C,D	-
T03	0	-	-	0	0	-	-	CI	14	14	1	A	-
T04	0	-	-	3	+	-	+	CI	4	10	2	B,D	74.80
T05	0	-	-	2	0	-	-	SI	20	20	1	B,C,*	-
T06	0	-	-	0	0	-	+	SI	0	0	0		-
T07	0	-	-	3	-	-	+	SI	7	12	6	D	-
T08	B	-	-	3	+	-	+	CI	7	57	4	B,C,D	-
T09	B	-	-	3	+	-	+	SI	3	12	3	B,C	-
T10	B	-	-	1	+	+	+	CI	1	12	1	B	-
T11	F	-	-	3	+	-	+	CI	5	10	5	B,C,D	-
T12	F	-	-	1	+	+	+	CI	4	7	3	B	52.68
T13	F	-	+	1	+	+	+	CI	1	2	2	B	22.05
T14	F	-	-	3	+	-	+	CI	2	8	6	C	-
T15	F	-	-	3	+	-	+	SI	2	2	0	A	-
T16	F	-	-	1	+	+	+	CI	3	11	2	B	61.28
T17	P	+	+	1	+	+	+	CI	1	1	1		17.33
T18	P	+	+	1	+	+	+	CI	5	5	1		23.14

ID Student identifier
EXP Programming experience
(0 none, B BASIC/LOGO, F FORTRAN, P Pascal)
Initial and Final Solution Features:
1 Syntax and logic (+ correct, - incorrect)
2 Logic only (+ correct, - incorrect)
3 Algorithm (0 none, 1 efficient algorithm,
2 isolate all cases, 3 complex shuffle)
4 Knowledge that original values could be destroyed
(0 no indication, + principle was known,
- principle was unknown)
5 Swapping technique (+ exchanges completed,
- exchanges incomplete or not used, * unique behavior)
6 Assignment statements (+ used, - not used)
7 IF statements (0 none, SI single statement in IF,
CI compound statements in IF)
8 Number of test cases code processes correctly (maximum is 6)

MINIPAS History Fea
1 Number of init
2 Number of tota
3 Number of unic
4 Programming pr
(A IF synta
B IF synta
C ordering
D ordering
* unique e
5 Completion tim

	Final solution							
5	1	2	3	4	5	6	7	8
-	-	-	2	+	-	+	CI	0
-	-	-	3	+	-	+	SI	0
-	-	-	0	0	-	-	SI	0
74.80	+	+	3	+	-	+	CI	6
-	-	-	2	0	-	-	SI	0
-	-	-	0	0	-	+	SI	0
-	-	-	3	+	-	+	SI	0
-	-	-	3	+	-	+	CI	1
-	-	-	3	+	-	+	SI	0
-	-	-	1	+	+	+	CI	0
-	-	-	3	+	-	+	CI	0
52.88	+	+	1	+	+	+	CI	6
22.05	+	+	1	+	+	+	CI	6
-	-	-	1	+	-	+	CI	4
-	-	-	3	+	-	+	SI	0
61.28	+	+	1	+	+	+	CI	6
17.33	+	+	1	+	+	+	CI	6
23.14	+	+	1	+	+	+	CI	6

tory Features:
of initial compilations
of total compilations
of unique program versions
mming problems:
IF syntax - boolean expression,
IF syntax - use of BEGINs/ENDs,
ordering of swapping values,
ordering of comparisons,
nique errors)
tion time (in minutes)

Table C-4. Control group protocols for the three-variable sort problem

		Features											
		Initial solution							MINIPAS history				
ID	EXP	1	2	3	4	5	6	7	1	2	3	4	5
C01	0	-	-	2	-	-	+	CI	9	18	9	A,B,C,D	-
C02	0	-	-	0	0	-	+	SI	9	10	2	A,*	-
C03	0	-	-	2	0	*	+	SI	30	30	0	A,*	-
C04	0	-	-	0	0	-	-	SI	17	17	0	A	-
C05	0	-	-	0	-	-	+	CI	12	12	1	A	-
C06	0	-	-	2	-	-	+	CI	14	14	0	C,D	-
C07	0	-	-	3	+	-	+	SI	1	2	2	D	-
C08	B	-	-	2	-	-	+	SI	14	14	0	A,*	-
C09	B	-	-	2	-	-	+	CI	3	23	3	A,C,D	-
C10	B	-	-	3	+	-	+	CI	19	21	3	B	75.
C11	F	-	-	1	+	+	+	CI	1	2	2	B,D	-
C12	F	-	-	2	+	+	+	CI	19	19	1	B	43.
C13	F	-	-	1	+	+	+	CI	1	8	8	B	-
C14	F	+	+	1	+	+	+	CI	1	1	1		20.
C15	P	-	+	3	+	-	+	CI	1	1	1		16.
C16	P	-	+	2	+	*	+	CI	2	1	1	*	18.
C17	P	-	-	1	+	+	+	CI	1	3	3	D	50.
C18	P	-	-	3	0	-	-	SI	5	10	3	B,C	-

ID Student identifier
EXP Programming experience
(0 none, B BASIC/LOGO, F FORTRAN, P Pascal)
Initial and Final Solution Features:
1 Syntax and logic (+ correct, - incorrect)
2 Logic only (+ correct, - incorrect)
3 Algorithm (0 none, 1 efficient algorithm,
2 isolate all cases, 3 complex shuffle)
4 Knowledge that original values could be destroyed
(0 no indication, + principle was known,
- principle was unknown)
5 Swapping technique (+ exchanges completed,
- exchanges incomplete or not used, * unique behavior)
6 Assignment statements (+ used, - not used)
7 IF statements (0 none, SI single statement in IF,
CI compound statements in IF)
8 Number of test cases code processes correctly (maximum is 6)

MINIPAS History Fe
1 Number of ini
2 Number of tot
3 Number of uni
4 Programming p
(A IF synt
B IF synt
C orderin
D orderin
* unique
5 Completion ti

	Final solution							
5	1	2	3	4	5	6	7	8
-	-	+	1	+	+	+	CI	1
-	-	-	0	-	-	+	CI	0
-	-	-	2	-	-	+	SI	0
-	-	-	0	-	-	+	SI	0
-	-	-	0	-	-	+	SI	0
-	-	-	2	-	-	+	CI	0
-	-	-	2	+	+	+	CI	0
-	-	-	2	-	-	+	SI	0
-	-	-	2	+	+	+	CI	1
75.60	+	+	3	+	-	+	CI	6
-	-	+	1	+	+	+	CI	3
43.42	+	+	2	+	+	+	CI	6
-	-	-	1	+	+	+	CI	1
20.62	+	+	1	+	+	+	CI	6
16.47	+	+	2	+	+	+	CI	6
18.72	+	+	2	+	*	+	*	6
50.25	+	+	1	+	+	+	CI	6
-	-	-	3	-	-	+	SI	0

tory Features:
of initial compilations
of total compilations
of unique program versions
mming problems:
IF syntax - boolean expression,
IF syntax - use of BEGINs/ENDs,
ordering of swapping values,
ordering of comparisons,
unique errors)
tion time (in minutes)

APPENDIX D: POSTTEST 2, SCORING PROCEDURE, AND POSTTEST 2 PROTOCOLS

Name _____

1. Assume the following constant declarations have been made in a Pascal program.

```
CONST   MAX = 20.0;
        BIGGEST = 10;
```

   Circle the letters of all array declarations appearing below that are legal in Pascal.

   a. VAR Aarray: Array [0..7] of Char;

   b. VAR Barray: Array ['A'..'Z'] of Real;

   c. VAR Carray: Array [1..Max] of Integer;

   d. VAR Farray: Array [INTEGER] of 1..7;

   e. VAR Garray: Array [1..Z] of Boolean;

2. Assume the following declarations have been made for a Pascal program.

```
CONST   MAX = 10;
VAR     X: Array [1..Max] of Integer;
        Y: Array [1..Max] of Real;
        I,J: Integer;
```

   Circle the program segments below that will cause some type of RUN-TIME error and describe the error that will be caused.

   a. `For I:= 1 to Max Do X[I] := X[I] + 1;`

   b. `For I:= 0 to Max-1 Do X[I+1] := X[I+1] * I;`

   c. `For I:= 1 to Max Do Y[I] := X[I] * X[I];`

   d.
```
For I := Max Downto 0 Do
    Begin
      J := I + 1;
      Readln (X[J]);
    end;
```

3. What values will be stored in Arrays X and Z after the following program has been executed? Use the following data as needed for input data:
   4 17 3 2 8 11 10 16 13 99 0 6

```
Program XXX (input,output);
Const  Max = 8;
Var    J,N,I : Integer;
       X : Array [1..Max] of Integer;
       Z : Array [1..Max] of Integer;

Begin
  Readln (N);
  For I := 1 to N Do Readln (X[I]);
  For J := 1 to N Do Readln (Z[J];
  I := 1;
  For J := N Downto 1 Do
    Begin
    Z[J] := X[I];
    I := I + 1;
    End;
End.
```

4. What will be stored in Xarray after this program has finished execution? Use the following data as needed for input data: 4 8 7 3 2 9 10 0 16 2 1 8 3

```
Program AAAA (Input, Output);
Const   Max = 8;
Var     Xarray : Array [1..Max] of Integer;
        J,N,L : Integer;

Begin
  For J := 1 to Max Do Readln (Xarray[J]);
  N := 7;
  For J := 1 to N do
     Begin
     If J <> N
        then Xarray[J] := Xarray[J+1]
        else Xarray[J] := Xarray[1];
     End;
End.
```

5. Write Pascal code that will compare the contents of X[1] to Y[1], X[2] to Y[2], etc. and print out a message after each comparison stating which one contains the larger of the two integers stored in each array. (You may assume that the value in X[I] will never equal the value in Y[I]). Assume the following declarations have been made:

```
Const   Max = 7;
Var     X,Y : Array [1..Max] of Integer;
        I,J,K: Integer;
```

6. Part of a program that will REVERSE the order of the values stored in array X appears below. (If the values in X were 2, 4, 9, 10, 16 then the code below would reverse these values so that X would contain 16, 10, 9, 4, 2.) Fill in the bounds to the FOR statement that would be required to perform this reversal and add whatever Pascal statements are necessary to complete the reversal. You may use ONLY those constants, arrays, and variables that have been declared below. You MAY NOT declare any additional ones. Use your own input data for this problem.

```
Program Reversem (Input,Output);
Const  Max = 10;
Var    X : Array [1..10] of Integer;
       I,J,N,R: Integer;
Begin
  For I := 1 to Max do Readln (X[I]);
  For I := __________ to __________ Do
     Begin
     (* Add statements needed to complete REVERSAL below*)




     End;
End.
```

7. Write a Pascal program that will put the values stored in an integer array of size 6 in order such that the smallest value stored in the array is located in the first element of the array and the largest value is located in the last element of the array. You may not use any constants, arrays, or variables other than those that have been declared for you in the code shown below. Use your own input data for this problem.

```
Program PutInOrder (input,output);
Const  MAX = 6;
Var    X : Array [1..6] of Integer;
       I,J,Z,R: Integer;
Begin
  For I := 1 to MAX Do Readln (X[I]);
```

Turn this part of the test in and get the instructions for the last part of the test.

Logon to your VAX account. Type $CAS, select the ASSIGNMENT option, and run NEWMINI. Enter the code you wrote for problem 7. Raise your hand when you have entered the code so that either Warner or Lib can check to be sure the same code was entered. Once this has been verified, you may continue to work on your solution, testing it and making changes, until it works properly (or you run out of class time).

Scoring Procedure

Problem 1 (5 points)
- 1 point for each subitem properly marked

Problem 2 (8 points)
- 1 point for each programming segment properly marked
- 1 point for each correct error description

Problem 3 (8 points)
- 8 points if the correct values were specified for elements 1-4 of both arrays, and elements 5-8 were left blank
- 5 points if only elements 1-4 of both arrays contained values, but the values were incorrect.
- 3 points if the values were correctly read into the arrays originally

Problem 4 (8 points)
- 8 points if all eight cells contained correct values
- 6 points if the majority (5 or more) of cells contained the correct values
- 2 points if the student read the values in properly before any other processing occurred

Problem 5 (5 points)
- 5 points for a syntactically correct solution
- 4 points for a logically correct solution that contained syntax errors
- 2 points if a FOR statement was used properly
- 2 points if an IF statement and two WRITELNs were used
- -1 if output messages failed to state the cell name

Problem 6 (8 points)
- 7 points if solution was logically correct but contained syntax errors
- 5 points if exchanges were correctly performed but an "out of bound" error could occur due to an incorrect bound
- 3 points if student attempted an exchange, but the exchange contained an error
- 2 points for correctly identifying bounds, but failing to attempt an exchange

Problem 7 (15 points)
- 13 points for logically correct code containing minor syntax errors
- 10 points for solutions properly implementing two nested FOR loops, a single IF statment, value exchanges, but errors occured in value exchanges
- 8 points for solutions properly using IF statements

```
  and value exchanges, but "out of bound" error
  was present, or the student failed to use nested
  loops
5 points if student properly performed an exchange only
3 points if IF statement was properly used
1 point added to score if efficient bounds were used
```

Table D-1. Treatment group protocols for the comparison and reversal problems

		Comparison					Solution features	
ID	EXP	1	2	3	4	5	1	2
T02	0	-	+	+	+	+	-	-
T03	0	+	+	+	+	+	-	-
T04	0	-	+	+	+	+	+	+
T05	0	-	+	+	+	+	-	-
T09	B	-	-	-	+	+	-	-
T10	B	-	-	+	+	+	-	-
T11	F	+	+	+	+	+	+	+
T12	F	+	+	+	+	+	+	+
T13	F	+	+	+	+	+	-	-
T14	F	+	+	+	+	+	+	+
T15	F	-	-	+	-	+	-	-
T16	F	-	+	+	+	+	+	+
T17	P	+	+	+	+	+	+	+
T18	P	+	+	+	+	+	-	-

ID Student identifier
EXP Programming experience
(0 none, B BASIC/LOGO, F FORTRAN, P Pascal)
Comparison Features:
1 Syntax and logic (+ correct, - incorrect)
2 Logic only (+ correct, - incorrect)
3 FOR statement (+ used, - not used)
4 Same index to address both arrays (+ used, - not used)
5 IF statement (+ used, - not used)
Reversal Features:
1 Syntax and logic (+ correct, - incorrect)
2 Logic only (+ correct, - incorrect)
3 Algorithm (0 not classifiable, S single index, T two-variable)
4 Boundary expressions in FOR (+ correct, - incorrect)
5 Number of memory cells used to preserve values
6 Preservation of values (0 not classifiable, + values preserved, - values lost)

ures

Reversal				
2	3	4	5	6
-	S	-	1	+
-	S	-	1	+
+	S	+	1	+
-	S	-	2	+
-	0	0	0	0
-	S	-	1	+
+	S	+	1	+
+	S	+	1	+
-	S	-	1	+
+	T	+	1	+
-	T	-	0	-
+	S	+	1	+
+	S	+	1	+
-	T	-	1	+

ost)

Table D-2. Control group protocols for the comparison and reversal problems

		Comparison					Solution features	
ID	EXP	1	2	3	4	5	1	2
C02	0	-	+	+	+	+	-	-
C03	0	-	+	+	-	+	-	-
C05	0	+	+	+	+	+	-	+
C06	0	+	+	+	+	+	+	+
C08	B	-	-	+	+	+	-	-
C09	B	+	+	+	+	+	-	-
C10	B	-	+	+	+	+	-	+
C11	F	+	+	+	+	+	-	+
C12	F	+	+	+	+	+	-	+
C13	F	-	+	+	+	+	-	-
C14	F	+	+	+	+	+	+	+
C15	P	+	+	+	+	+	-	+
C16	P	+	+	+	+	+	-	-
C17	P	+	+	+	+	+	+	+
C18	P	-	+	+	-	+	-	-

ID Student identifier
EXP Programming experience
(0 none, B BASIC/LOGO, F FORTRAN, P Pascal)
Comparison Features:
1 Syntax and logic (+ correct, - incorrect)
2 Logic only (+ correct, - incorrect)
3 FOR statement (+ used, - not used)
4 Same index to address both arrays (+ used, - not used)
5 IF statement (+ used, - not used)
Reversal Features:
1 Syntax and logic (+ correct, - incorrect)
2 Logic only (+ correct, - incorrect)
3 Algorithm (0 not classifiable, S single index, T two-variable)
4 Boundary expressions in FOR (+ correct, - incorrect)
5 Number of memory cells used to preserve values
6 Preservation of values (0 not classifiable, + values preserved, - values lost)

cures

Reversal				
2	3	4	5	6
-	S	-	0	-
-	0	-	0	-
+	S	+	1	+
+	S	+	1	+
-	0	-	0	-
-	S	-	0	-
+	T	+	2	+
+	T	-	2	+
+	T	+	1	+
-	T	-	2	+
+	S	+	1	+
+	S	-	1	+
-	S	-	2	+
+	T	+	2	+
-	0	-	0	-

lost)

Table D-3. Treatment group protocols for the ascending sort problem

		Features													
		Initial solution									MINIPAS history				
ID	EXP	1	2	3	4	5	6	7	8	9	1	2	3	4	5
T02	0	-	+	S	+	+	+	+	+	E	2	18	5	A,C,D,E	-
T03	0	-	-	0	-	-	+	+	-	-	3	19	7	A,B,C	-
T04	0	-	+	S	+	+	+	+	E	E	1	2	2	*	30.70
T05	0	-	-	0	+	-	+	+	-	-	4	6	3	B,C	-
T09	B	-	-	B	-	-	+	+	-	-	8	10	3	D	-
T10	B	-	-	S	+	+	+	+	E	E	1	5	5	B,C	-
T11	F	-	+	B	+	+	+	+	E	E	3	3	1	A	10.52
T12	F	+	+	B	+	+	+	+	E	E	1	1	1		14.55
T13	F	-	+	B	+	+	+	+	-	-	3	3	1		-
T14	F	-	-	S	-	+	+	+	+	+	2	4	2	E	16.00
T15	F	-	-	0	-	-	+	+	-	-	8	22	2	A	-
T16	F	-	+	S	+	+	+	+	E	E	2	2	2	C	17.23
T17	P	+	+	S	+	+	+	+	+	+	3	3	2		14.31
T18	P	-	+	S	+	+	+	+	+	E	3	5	3	C,D,E	37.26

ID Student identifier
EXP Programming experience
(0 none, B BASIC/LOGO, F FORTRAN, P Pascal)
Initial and Final Solution Features:
1 Syntax and logic (+ correct, - incorrect)
2 Logic only (+ correct, - incorrect)
3 Algorithm (0 not classifiable, S selection sort, B bubble sort)
4 Preservation of values (+ values preserved, - values lost)
5 Nested loops (+ used, - not used)
6 IF statements (+ used, - not used)
7 Assignment statements (+ used, - not used)
8 Passes through array (E excessive, + efficient, - not enough)
9 Comparisons per pass (E excessive, + efficient, - not enough)

MINIPAS History F
1 Number of in
2 Number of to
3 Number of un
4 Modification
(A syntax
B additi
C bounds
D compar
E stat.em
* unique
5 Completion t

	Final solution								
5	1	2	3	4	5	6	7	8	9
-	-	+	B	+	+	+	+	+	-
-	-	-	B	-	+	+	+	E	-
30.70	+	+	S	+	+	+	+	E	+
-	-	-	B	+	-	+	+	-	-
-	-	-	0	-	-	+	+	-	-
-	-	-	S	+	+	+	+	E	E
10.52	+	+	B	+	+	+	+	E	E
14.55	+	+	B	+	+	+	+	E	E
-	-	+	B	+	+	+	+	-	-
16.00	+	+	S	+	+	+	+	+	+
-	-	-	S	-	+	+	+	E	E
17.23	+	+	S	+	+	+	+	+	+
14.31	+	+	S	+	+	+	+	+	+
37.28	+	+	S	+	+	+	+	E	+

History Features:
ber of initial compilations
ber of total compilations
ber of unique program versions
ifications to code:
(A syntax,
B addition or deletion of looping structures,
C bounds on loops,
D comparisons between cells,
E statements in swap code,
* unique changes)
pletion time (in minutes)

Table D-4. Control group protocols for the ascending sort problem

		Features													
		Initial solution									MINIPAS history				
ID	EXP	1	2	3	4	5	6	7	8	9	1	2	3	4	5
C02	0	-	-	B	-	+	+	+	E	E	2	6	5	C,D	-
C03	0	-	-	B	-	-	+	+	-	-	10	12	3	A,C,D,*	-
C05	0	-	-	B	+	-	+	+	-	-	2	5	4	C,B	32.85
C06	0	+	+	B	+	+	+	+	E	E	2	2	1		10.85
C08	B	-	-	0	-	-	+	+	-	-	4	14	1	A	-
C09	B	-	-	S	+	-	+	+	-	-	3	10	7	B	-
C10	B	-	-	0	+	-	+	+	-	-	2	7	6	B,*	-
C11	F	-	-	S	+	+	+	+	E	E	8	11	3	A,B,D	39.65
C12	F	+	+	B	+	+	+	+	+	E	2	2	1		12.43
C13	F	-	+	B	+	+	+	+	E	E	2	8	2	C	-
C14	F	-	+	S	+	+	+	+	E	E	1	1	1	C	9.67
C15	P	-	+	B	+	+	+	+	E	E	3	5	3	C	32.85
C16	P	-	+	S	+	+	+	+	+	E	3	5	2	C	20.75
C17	P	-	-	B	+	-	+	+	-	-	1	1	1	B	-
C18	P	-	-	0	+	-	+	-	-	-	2	9	4	A,B	-

ID Student identifier

EXP Programming experience
(0 none, B BASIC/LOGO, F FORTRAN, P Pascal)

Initial and Final Solution Features:
1 Syntax and logic (+ correct, - incorrect)
2 Logic only (+ correct, - incorrect)
3 Algorithm (0 not classifiable, S selection sort, B bubble sort)
4 Preservation of values (+ values preserved, - values lost)
5 Nested loops (+ used, - not used)
6 IF statements (+ used, - not used)
7 Assignment statement (+ used, - not used)
8 Passes through array (E excessive, + efficient, - not enough)

MINIPAS History Fe
1 Number of ini
2 Number of tot
3 Number of uni
4 Modifications
(A syntax,
B additio
C bounds
D compari
E stateme
* unique
5 Completion ti

	Final solution								
5	1	2	3	4	5	6	7	8	9
-	-	-	B	-	+	+	+	E	E
-	-	-	0	-	-	+	+	-	-
32.85	+	+	B	+	+	+	+	E	E
10.85	+	+	B	+	+	+	+	E	E
-	-	-	0	-	-	+	+	-	-
-	-	+	S	+	+	+	+	E	E
-	-	-	S	+	+	+	+	E	E
39.65	+	+	S	+	+	+	+	E	E
12.43	+	+	B	+	+	+	+	+	E
-	-	+	B	+	+	+	+	E	E
9.67	+	+	S	+	+	+	+	E	E
32.85	+	+	B	+	+	+	+	E	E
20.75	+	+	S	+	+	+	+	+	E
-	-	-	B	+	-	+	+	-	-
-	-	-	0	0	+	+	+	-	-

story Features:
r of initial compilations
r of total compilations
r of unique program versions
ications to code:
syntax,
addition or deletion of looping structures,
bounds on loops,
comparisons between cells,
statements in swap code,
unique changes)
etion time (in minutes)