Defining programming and assessing its cognitive demands is problematic because programming is a complex configuration of activities that vary according to what is being programmed, the style of programming, and how rich and supportive the surrounding programming environment is (Kurland et al., 1984; Pea & Kurland, 1983).

One consequence of the fact that programming refers to a configuration of activities is that different combinations of activities may be involved in any specific programming project. These activities include, at a general level, problem definition, design development and organization, code writing, and debugging (Pea & Kurland, 1983). Different combinations of activities will entail different cognitive demands. For example, a large memory span may facilitate the mental simulations required in designing and comprehending programs. Or analogical reasoning skill may be important for recognizing the similarity of different programming tasks and for transferring programming methods or procedures from one context to another. An adequate assessment of the cognitive demands of programming will depend on analyses of the programming activity and examination of the demands of different component processes.

Specifying Levels of Programming Expertise

In assessing the cognitive demands of programming, specifying the intended level of expertise is essential. Different levels of expertise will entail different cognitive demands. In many Logo programming classrooms, we have observed children engaging in what we term *brute-force paragraph* programming, or what Rampy (1984) has termed *product-oriented* programming. This style is analogous to so-called spaghetti programming in BASIC. When programming, students decide on desired screen effects and then write linear programs, lining up commands that will cause the screen to show what they want in the order they want it to happen. Students do not engage in problem decomposition or use the powerful features of the language to structure a solution to the programming problem. For example, if a similar shape is required several times in a program, students will write new code each time the effect is required, rather than writing one general procedure and calling on it repeatedly. Programs thus consist of long strings of Logo primitives that are nearly impossible to read, modify, or debug, even for the students who have written them. Although students may eventually achieve their goal, or at least end up with a graphics display with which they are content, the only "demands" we can imagine for such a linear approach to programming are stamina and determination.

Thus, as a first step in determining what the cognitive demands are for learning or doing programming, we need to distinguish between *linear* and *modular* programming (or between learning to program elegantly and efficiently, and a style that emphasizes the generation of effects without any consideration of how they were generated).

The beginner's linear style of constructing programs, whether in Logo or BASIC, contrasts with modular programming (a planful process of structured problem solving). Here, component elements of a task are isolated, procedures for their execution developed, and the parts assembled into a program and debugged. This type of programming requires a relatively high-level understanding of the language. Modular programming in Logo, where programs consist of organized, reusable subprocedures, requires that students understand the flow of control of the language, such powerful control structures as recursion, and the passing of values of variables between procedures. The cognitive demands for this kind of programming are different from the demands for linear programming, as are the potential cognitive benefits that may result from the two programming styles.

Distinguishing Between Product and Process

In assessing the demands for different levels of expertise, however, it is important not to equate level of expertise with the effects the students' programs produce. We must distinguish product from process (Werner, 1937). We have seen very elaborate graphics displays created entirely with brute-force programming. One characteristic of highly interactive programming languages such as Logo and BASIC is that students can often get the effects they want simply by trial and error—without any overall plan, without fully understanding how effects are created, without the use of sophisticated programming techniques, and without recognizing that a more planful program could be used as a building block in future programs.

Furthermore, in school classrooms students borrow code from each other and then integrate the code into their programs without bothering to understand *why* the borrowed code does what it does. Students therefore can often satisfy a programming assignment by piecing together major chunks imported from other sources. Although such "code stealing" is an important and efficient technique widely employed by expert programmers, an overreliance on other people's code that is beyond the understanding of the borrower is unlikely to lead to deeper understandings of programming. Therefore, if we simply correlate students' *products* with their performance on particular demands or programming proficiency

20 Mapping the Cognitive Demands of Learning to Program

D. Midian Kurland
Catherine A. Clement
Ronald Mawby
Roy D. Pea
Bank Street College of Education

Introduction

Vociferous arguments have been offered for incorporating computer programming into the standard precollege curriculum (Luehrmann, 1981; Papert, 1980; Snyder, 1984). Many parents and educators believe that computer programming is an important skill for all children in our technological society. In addition to pragmatic considerations, there is the expectation among many educators and psychologists that learning to program can help children develop general high-level thinking skills useful in other disciplines, such as mathematics and science. However, there is little evidence that current approaches to teaching programming bring students to the level of programming competence needed to develop general problem-solving skills, or to develop a model of computer functioning that would enable them to write useful programs. Evidence of what children actually do in the early stages of learning to program (Pea & Kurland, 1983; Rampy, 1984) suggests that in current practices programming may not evoke the kinds of systematic, analytic, and reflective thought that is characteristic of expert adult programmers (Kurland, Mawby, & Cahir, 1984).

As the teaching of programming is initiated at increasingly early grade levels, questions concerning the cognitive demands for learning to program are beginning to surface. Of particular interest to both teachers and developmental psychologists is whether there are specific cognitive demands for learning to program that might inform our teaching and tell us what aspects of programming will be difficult for students at different stages in the learning process.

In the first part of this chapter, we explore factors that may determine the cognitive demands of programming. In the second part, we report on a study of these cognitive demands conducted with high school students learning Logo. The premise for the study was the belief that in order for programming to help promote the development of certain high-level thinking skills, students must attain a relatively sophisticated understanding of programming. Therefore, we developed two types of measures: measures to assess programming proficiency, and measures to assess certain key cognitive abilities that we hypothesized to be instrumental in allowing students to become proficient programmers. The relationship between these two sets of measures was then assessed.

Issues in Determining the Cognitive Demands of Programming

One of the main issues in conducting research on the cognitive demands of programming is that the term *programming* is used loosely to refer to many different activities involving the computer. These activities range from what a young child seated in front of a computer may do easily using the immediate command mode in a language such as Logo, to what college students struggle over, even after several years of programming instruction. Contrary to the popular conception that young children take to programming "naturally" whereas adults do not, what the child and the adult novice are actually doing and what is expected of them is radically different. Clearly, the cognitive demands for the activities of the young child and the college student will also differ. Thus, what is meant by programming must be clarified before a discussion of demands can be undertaken.

Compensatory Strategies

This point suggests another important factor that complicates the identification of cognitive demands of programming. Any programming problem can be solved in many ways. Different programmers can utilize a different mix of component processes to write a successful program. This allows for high levels on some abilities to compensate for low levels on others. For example, a programmer may be deficient in the planning skills needed for good initial program design but may have high levels of skills needed to easily debug programs once drafted. Thus, it will not be possible to identify the unique set of skills that are necessary for programming. Instead, different programmers may possess alternative sets of skills, each of which is sufficient for programming competence.

The Programming Environment

The features of the programming environment may also increase or decrease the need for particular cognitive abilities important for programming. We cannot separate the pure demands for using a programming language from the demands and supports provided by the instrumental, instructional, and social environments. For example, an interactive language with good trace routines can decrease the need for preplanning by reducing the difficulty of debugging. Similarly, implementations of particular languages that display both the student's program and the screen effects of the code side by side in separate "windows," such as Interlisp-D, can reduce the difficulty in understanding and following flow of control.

In learning to program, the instructional environment can reduce certain cognitive demands if it offers relevant structure, or it can increase demands if it is so unstructured that learning depends heavily on what the students themselves bring to the class. For example, understanding the operation of branching statements of the IF–THEN–ELSE type requires an appreciation of both conditional logic and the operation of truth tables. If students have not yet developed such an appreciation, doing programs that require even simple conditional structure can be very confusing. However, with appropriate instruction, an understanding of how to *use* conditional commands in some limited contexts (such as conditional stop rules to terminate the execution of a loop) can be easily picked up by students. Thus, in the absence of instruction, conditional reasoning skill can be a major factor in determining who will learn to program. However, with instructional intervention, students can pick up enough functional knowledge about conditional commands to take them quite far.

Instruction is important in other ways also. It has been our experience that students are very poor at choosing appropriate programming projects that are within their current ability, yet which will stretch their understanding and force them to think about new types of problems. They are poor at constructing for themselves what Vygotsky would describe as the *zone of proximal development* (Rogoff & Wertsch, 1984). Consequently, too little guidance on the part of the teacher can lead to inefficient or highly frustrating programming projects. On the other hand, too much teacher-imposed structure can make projects seem arbitrary and uninteresting, with the result that they are less likely to evoke students' full attention and involvement. Finding the right balance between guidance and discovery will have a major impact on the kinds of cognitive abilities students will have available to them when engaging in programming tasks.

Finally, the social context can mediate the demands placed on an individual for learning to program because programming—particularly in elementary school classrooms—is often a collaborative process (Hawkins, 1983). The varying skills of student collaborators might enable them to create programs that any one of them alone could not have produced. Although teamwork is typical of expert programmers, it raises thorny assessment problems in an educational system that stresses individual accountability.

In summary, several factors complicate the identification of general cognitive abilities that will broadly affect a child's ability to learn to program. In asking about demands, we must consider level of expertise, the impact of supportive and/or compensatory programming environments, and the role of instructional and social factors that interact with children's initial abilities for mastering programming.

ANALYSIS OF THE COGNITIVE DEMANDS OF MODULAR PROGRAMMING

Two central motivations for teaching programming to precollege students are to provide a tool for understanding mathematical concepts and to develop general problem-solving skills. But achieving these goals requires that students learn to program extremely well (Mawby, 1984). To use a language like Logo to develop an understanding of such mathematical concepts as variable and function requires that students learn to program with variables and procedures, generate code that can be reusable, and understand the control structure of the language. Students must also become reasonably good modular programmers before Logo can be

Procedural reasoning ability is one of the important skills underlying the ability to program, because programmers must make explicit the antecedents necessary for different ends and must follow all the possible consequences of different antecedent conditions. Designing and following the flow of control of a program necessitates understanding different kinds of relations between antecedent and consequent events, and organizing and interrelating the local means–end relations (modules) leading to completion of the program. Procedural reasoning thus includes understanding conditional relationships, temporal sequencing, hypothetical deduction, and planning.

Decentration also may be an important skill in programming because programmers must distinguish what they know and intend from what the computer has been instructed to execute. This is important in both program construction and debugging: In the former, the program designer must be aware of the level of explicitness required to adequately instruct the computer; in the latter, he or she must differentiate between what the program "should" do from what it in fact did. We have found that such decentering is a major hurdle in program understanding at the secondary school level (Kurland & Pea, 1985).

On the basis of this rational analysis, we designed a study to investigate the relationship of measures of procedural reasoning and decentering to the acquisition of programming skill.

METHOD

To investigate the relationship between these cognitive abilities and programming competence, we studied novice programmers learning Logo. Logo was chosen because of the high interest it has generated within the educational community, and because the Logo language has specific features that support certain important thinking skills. For example, the strategy of problem decomposition is supported by Logo's modular features. Logo procedures may be created for each subpart of a task. The procedures may be written, debugged, and saved as independent, reusable modules and then used in combination for the solution of the larger problem. Efficient, planful problem decomposition in Logo results in flexibly reusable modular procedures with variable inputs. Whereas the same can be true of languages such as BASIC, the formal properties of Logo appeared to be more likely to encourage students to use structured programming.

Participants and Instructional Setting

Participants in the study were 79 eighth- to 11th-grade female high school students enrolled in an intensive 6-week summer program designed to improve math skills and introduce programming. The goal of the program was to improve students' mathematical understanding, whereas building their sense of control and lessening their anxiety about mathematics. (See Confrey, 1984, and Confrey, Rommney, & Mundy, 1984, for details about the affective aspects of learning to program.) Those admitted to the program were generally doing very well in school and had high career aspirations, but they were relatively poor in mathematics and, in some cases, experienced a great deal of math-related anxiety.

Each day the students attended two 90-minute mathematics classes, as well as lectures and demonstrations on how mathematics is involved in many aspects of art and science. Each student also spent 90 minutes a day in a Logo programming course. The teachers hoped that the programming experience would enable students to explore mathematical principles and thus lead them to new insights into mathematics. The guiding philosophy of the program, which influenced both the mathematics and Logo instruction, was constructivist. This Piagetian-inspired philosophy of instruction holds that a person's knowledge and representation of the world is the result of his or her own cognitive activity. Learning will not occur if students simply memorize constructions presented by their teachers in the form of facts and algorithms. Thus, students were expected to construct understandings for themselves through their direct interactions with and explorations of the mathematics or programming curricula.

The Logo instruction was given in small classes, with the students working primarily in pairs, that is, two students to a computer. There was a 6:1 student–teacher ratio, and ample access to printers and resource materials. In order to provide structure for the students' explorations of Logo, the program staff created a detailed curriculum designed to provide systematic learning experiences involving the Logo turtle graphics commands and control structures. Although the curriculum itself was detailed and carefully sequenced, the style of classroom instruction was influenced by the discovery-learning model advocated by Papert (1980). Thus, students were allowed to work at their own pace and were not directly accountable for mastery of specific concepts or commands. The instructors saw their primary role as helping students to develop a positive attitude towards mathematics and programming. In this respect, the program seemed by our observations to have been very successful.

MEASURES

We were interested in how the students' level of programming proficiency would relate to the specific cognitive abilities that our earlier analysis had indicated to be potentially important. We therefore developed the following measures of cognitive performance and programming proficiency.

Cognitive Demands Tasks

Two cognitive demands tasks were developed and administered to students at the beginning of the program. The first, *procedural flow of control task,* was designed to assess students' ability to use procedural reasoning in order to follow the flow of control determined by conditional relations. In this task, students had to negotiate a maze in the form of an inverted branching tree (see Fig. 20.1). At the most distant ends of the branches were a set of labeled goals. To get to any specific goal from the top of the maze, students had to pass through "gates" at each of the branching nodes. The conditions for passage through the gates involved satisfying either simple or complex logical structures (disjunctive or conjunctive). Passage through gates was permitted by a set of geometric tokens with which the student was presented at the beginning of each problem. Each gate was marked with the type or types of tokens that were required to gain passage. For example, a circle token allowed students to pass through a circular gate, but not through a square gate. If they had both a square and a triangle token, they could pass through a joint square–triangle gate, but not through a joint square–circle gate.

The task consisted of two parts. In the first, students were presented with five problems in which they had to find paths through the maze that did not violate the conditions for passage through the gates. They were given a set of tokens and asked to discover all the possible goals that could be reached with that set.

In the second part of the task, we designed two problems, based on a more complex maze, to add further constraints and possibilities for finding the optimal legal path to the goals. Unlike part one, at a certain point in the maze students could choose to trade one kind of token for another. As they passed through each gate, they forfeited the token that enabled them to get through it. This feature introduced additional planning and hypothetical reasoning requirements because the students had to foresee the sequential implications for choosing one path over other possible paths. This task allowed for several possible solutions that met the minimum requirements of the task (i.e., reaching a specified goal). However, some solutions were more elegant than others in that they used fewer tokens. Thus, it was of interest to see whether students would choose to go beyond an adequate solution to find an elegant one.

The task was designed using non-English symbolisms so that verbal ability and comprehension of the IF–THEN connectives would not be confounding factors. In natural language, IF–THEN is often ambiguous, its interpretation depending on context. We therefore did not include standard tests of the IF–THEN connective in propositional logic because computing truth values, as these tests require, is not strictly relevant to following complex conditional structures in programming.

The procedural flow of control task, therefore, involved a system of reasonable, although arbitrary and artificial, rules, not easily influenced by the subjects' prior world knowledge. The nested conditional structure of the tree and the logical structures of the nodes were designed to be analogous to the logical structures found in computer languages.

The second cognitive demands task was designed to assess decentering as well as procedural and temporal reasoning. In this *debugging task* students were required to detect bugs in a set of driving instructions that has supposedly been written for another person to follow. Students were given the set of written directions, a map, and local driving rules. They were asked to read over the directions and then, by referring to the map, catch and correct bugs in the directions so that the driver could successfully reach the destination. In order to follow the instructions and determine their accuracy, students had to consider means–ends relationships and employ temporal reasoning. They had to decenter by making a distinction between their own and the driver's knowledge. The kinds of bugs students were asked to find and correct included:

Inaccurate information bug: Instructions were simply incorrect (e.g., telling the driver to make a righthand turn at a corner instead of a left).

Ambiguous information bug: Instructions were insufficiently explicit to enable the driver to make a correct choice between alternative routes (e.g., telling the driver to exit off a road without specifying which of two possible exits to use).

Temporal order bug: One line of instruction was given at the wrong time (e.g., telling the driver to pay a token to cross a toll bridge before indicating where to purchase tokens).

Bugs due to unusual input conditions, and *embedded bugs* in which obvious corrections failed because they introduced and/or left a bug (e.g.,

Production Task. The production task was a paper-and-pencil test designed to assess students' skills in planning, problem decomposition, and features of programming style such as the conciseness and generality of procedures. Students were shown a set of seven geometric figures, represented in Fig. 20.2.

The students were instructed to select five of the seven figures and write Logo programs to produce them. The task called for students first to indicate the five figures they would write programs for, and then to number them in the order in which the programs would be written. It was hoped that this instruction would encourage the students to plan before writing their programs. Students were free, however, to alter the choice and/or order of their figures once they began to code. For each of their five programs, they were to write the code and give the run command needed to make the program produce the figure.

The task sheet included an area labeled *workspace*, analogous to the Logo workspace, in which students could write the procedures to be called by their programs. The layout of the task sheet, two sample problems, and explicit instructions made it clear that, once written in the workspace, the procedures were available to all programs.

The task was designed to encourage planning for modular procedures that could be reused across programs. In fact, figures B, C, E, F, and G could be programmed by writing three general-purpose procedures. An optimal solution would be to write a procedure with two variable inputs to produce rectangles, a "move over" procedure with one input, a "move up" procedure with one input, and then to use those three procedures in programs to produce figures B, C, E, F, and G. Figures B and G could be most efficiently produced using recursive programs, although recursion was not necessary.

Figures A and D were included as distractor items. Unlike the other five figures, they were designed *not* to be easily decomposed and could not be easily produced with code generated for any of the other figures.

The task could be solved by planful use of flexible modules of code. It could also be solved in many other ways, such as writing low-level, inelegant "linear" code consisting of long sequences of FORWARD, LEFT, and RIGHT commands, thereby never reusing modules of code. We were particularly interested in this style dimension because a linear solution gives no evidence that the student is using the Logo constructs that support and embody high level thinking.

Comprehension Tasks. Each of the two comprehension tasks presented four procedures: one superprocedure and three subprocedures. The students were asked first to write functional descriptions of each of the procedures, thus showing their ability to grasp the meaning of commands within the context of a procedure. Then they were asked to draw on graph paper the screen effects of the superprocedure when executed with a specific input. To draw the screen effects, students had to hand-simulate the program's execution, thus providing a strong test of their ability to follow the precise sequence of instructions dictated by the program's flow of control.

In the first comprehension tasks, the superprocedure was named TWOFLAGS and the subprocedures were CENTER, FLAG, and BOX. Figure 20.3 presents the Logo code for the procedures and a correct drawing of the screen effect of TWOFLAGS 10.

The second comprehension task included procedures with two inputs and a recursive procedure with a conditional stop rule. The task was designed to make the master procedure progressively harder to follow. The superprocedure was named ROBOT, and the three subprocedures were called BOT, MID, and TOP. Figure 20.4 presents the Logo code and correct drawing of the screen effects of ROBOT 30 25.

Both programming comprehension tasks were designed as paper-and-pencil tests that did not require the use of the computer. Students were given a sheet that listed the programs, a sheet on which to write their descriptions of what each procedure would do, and graph paper on which to draw their predictions of what the program would do when executed.

PROCEDURE

The cognitive demands measures were administered to the students on the first day of the program, along with a number of mathematics, problem-solving, and attitude measures (see Confrey, 1984, for a discussion of the attitude measures). The students were tested together in a large auditorium. Instructions for each test were read by the experimenters, who monitored the testing and answered all questions. Students were given 17 minutes for the procedural reasoning task and 12 minutes for the debugging task.

In the final week of the program, the students were administered the Logo proficiency test. Testing was done in groups of approximately 30 students each. Again the experimenters gave all the instructions and were present throughout the testing to answer students' questions. Students were given 30 minutes for the production task and 15 minutes each for the comprehension tasks.

RESULTS

Programming Proficiency Tasks

To use Logo as a tool for high-level thinking, one must employ relatively sophisticated Logo constructs, such as procedures with variable inputs and superprocedures which call subprocedures. To write and understand Logo programs using these language constructs, one needs to understand something about the pragmatics of writing programs and also have a good grasp of Logo's control structure, that is, how Logo determines the order in which commands are executed. The empirical question addressed is whether students develop such an understanding as the result of 5 weeks (approximately 45 hours) of intensive Logo instruction.

Comprehension Tasks. The assessments of Logo proficiency given at the end of the course indicated that mastery of Logo was limited. On the TWOFLAGS task, 48% of the students correctly drew the first flag, which required simulating the execution of TWOFLAGS through its call to FLAG in line 2. But only 21% correctly drew the second flag, with 19% of the students correct on both flags (showing that in almost all cases performance was cumulative).

A third of the students were partially right on the second flag. Analysis of errors on this flag indicated that more students had trouble following the flow of control than had difficulty keeping track of the values of the variables. An error in *place* on the second flag suggests that the student's simulation did not execute all the positioning lines of code, especially the call to CENTER in the last line of FLAG. This reveals an error in flow of control. An error in *size* on the second flag suggests that the student did not correctly pass the variable from TWOFLAGS to FLAG to BOX.

On the ROBOT task, 65% of the students correctly drew the body of the robot, which involved simulating the execution of ROBOT through its call to MID. Thirty-seven percent correctly drew the leg, which involved *following* the execution through ROBOT's call to BOT in line 4. TOP is a recursive procedure with inputs to ROBOT of 30 25; it executes three times. The first time TOP draws the head, the second time it draws the nose, and the last time it draws the mouth and then stops. Sixteen percent of the students correctly drew the head, 13% succeeded with the nose, and only 2% were able to follow the program execution all the way through to the mouth. The cumulative percentages are within 3% of these absolute percentages.

Analysis of the errors of students who were partially correct showed that more of them correctly passed the values of variables than followed the flow of control. In partially correct drawings, the parts of the robot were more often sized correctly than placed correctly.

The students' written descriptions of the procedures in both the TWOFLAGS and ROBOT tasks showed that many had a general, albeit vague, understanding of the procedures. Often students understood the code in that they gave adequate glosses of individual lines. But when tested by the drawing task, many revealed that they did not understand Logo's control structure well enough to trace the program's execution. This was especially clear when the order of the lines in a listing of the program differed from the order in which the lines were executed.

Some students failed to grasp the fact that, because variable values are local to the procedure call, values can be passed among procedures under different names. Even more failed to understand the most basic fact of flow of control: After a called procedure is executed, control returns to the next line of the calling procedure.

Production Task. In the production task, students made very little use of variables and reusable subprocedures. Although most were able to generate the figures, many did so following the linear programming style. Only 21% of the students avoided both distractor items. An additional 35% avoided either A or D singly. Thus, 44% of the students wrote programs for both A and D. Given a low level of programming proficiency, choosing the distractors was reasonable because, by design, linear programs for the distractors were easier than linear programs for figures B and G (and comparable to C and F).

Among the possible approaches to the task are *analytic* and *synthetic* decomposition. By analytic decomposition, we mean analyzing a single figure into component parts, writing procedures for the parts, and having the program call the procedures. By synthetic decomposition, we mean decomposition of the entire problem set into components, writing procedures for the parts, and then having each of the five programs call the appropriate modules of code. Note that although the five nondistractor figures contain only rectangles, the rectangles are of different sizes. Thus, high-level synthetic decomposition, unlike analytic decomposition, requires a general procedure with variable inputs for producing the rectangles.

Students were much more likely to use analytic than synthetic decomposition. In fact, 88% wrote, used, and reused a procedure at least once, giving evidence of some analytic decomposition. However, only 20% of the students gave evidence of synthetic decomposition by using a procedure for more than one program.

Figure 20.5 and Table 20.1 provide more detail on the features used by Logo students to produce the individual figures. In the analysis represented by Fig. 20.5, we wished to know, for each figure, whether students could write code to produce it and whether they could correctly use REPEAT, variables, and recursion. The REPEAT command is the simplest modular feature in Logo. Variables go further in transforming procedures into reusable *functions*, making the procedures more general, and hence more useful. Recursion is an extremely powerful Logo construct in which a procedure can call on copies of itself from within other copies. These features of Logo make modular code possible and thus support problem decomposition strategies.

The number of commands used to produce the program is a good summary indicator of style. For these tasks, elegant programs use few commands. We counted each use of a Logo primitive as one command. Each procedure call was counted as one command and, on the first call to a procedure, the commands within the procedure were counted. On subsequent calls to that procedure, only the call itself was counted.

The graph at the top of Fig. 20.5 displays several statistics concerning the number of commands used: the range, the mean, and the region containing the middle 50% of the scores. For comparison, we also include the number of commands used in an optimal solution of the task as a whole. This particular optimal solution "synthetically" decomposes the five rectangular figures with three subprocedures and produces the programs in the order E, F, C, B, G.

The figures fall into three groups: the distractors A and D; C, E, and F; and B and G. As noted, nearly half the students chose figures A and D, and 90% of the students who chose these figures were able to write a Logo program to produce them. As expected from the design of the figures, less than 10% of these programs used variables or REPEAT. Most of the code was low-level, brute-force style, which could not be reused in other programs. Thus, whereas the students wrote programs to produce the figure, their programming style gave no indication that they were engaged in the high-level thinking that Logo can support.

The group of figures C, E, and F was chosen by more than 90% of the students, and nearly 90% of these students wrote workable programs. More than half the students correctly used REPEAT, Logo's simpler, within-procedure modular construct. Less than 15% of these programs correctly used variables. This more elegant, across-program construct was largely ignored. As a result, most students needed more than the optimal number of commands to write programs for figures F and C.

Figures B and G were chosen by the least number of students (60% and 31%, respectively) and proved to be the most difficult because only half the students wrote workable programs. These programs used REPEAT and variables relatively often (REPEAT: 49% in B, 68% in G; variables: 43% in B, 40% in G). Thus, it seems that the skilled students who chose these figures did quite well. Of the other students who chose these figures, about half did not attempt to use variables, and half used variables

What factors may have kept these students from using the powerful and elegant features of Logo? It is unlikely that students did not notice the geometrical similarity among, for instance, figures C, E, and F. But in order to do a synthetic decomposition of the task, it is necessary to write procedures with variables. Moreover, coordinating subprocedures in a superprocedure requires a good understanding of Logo flow of control. Performance on the comprehension tasks showed that students had a fair understanding of individual lines of Logo code but had difficulty in following program flow of control.

Cognitive Demands Tasks

There was a fairly broad range of performances on the cognitive demands tasks. Many students showed moderate or high levels of reasoning skills as assessed by these tasks, and a few found the tasks fairly difficult.

Procedural Flow of Control Task. The two parts of this task were examined individually. The first part included a series of problems for students to solve, each of which posed a different set of constraints and/or goals for going through the maze. Difficult problems required a more exhaustive testing of conditions than did the others (i.e., the given tokens satisfied many nodes early on). Some problems were best solved using alternate strategies, such as searching from the bottom up rather than from the top down. Performance was relatively low on the more difficult problems (30–40% correct, as opposed to 55–70% correct on the less complex problems). This indicated that when many possibilities had to be considered, and there were no easy shortcuts to reduce the number of possibilities, students had difficulty testing all conditions.

In the second part, there were three levels of efficiency among correct routes corresponding to the number of tokens required to successfully reach the goal. Only 14% of the students on the first problem and 21% on the second problem found the most efficient route, whereas 41% of the students on the first problem and 79% on the second problem were unable to reach the goal at all. Few students tested the hypotheses needed to discover the most efficient route.

Debugging Task. Table 20.2 shows the percentage of students detecting and correcting each of the four types of bugs in the task. As shown, inaccurate information and temporal bugs were easiest to detect and correct (72–91% success). Students found it more difficult to successfully correct the ambiguous instructions. Only 48% were able to write instructions that were explicit enough for a driver to choose correctly among alternate routes. For the lines with embedded bugs, only 21% fully corrected the instructions; 40% caught and corrected one bug but not the other.

Results indicate that students had little difficulty detecting first-order bugs and correcting them when the corrections were simple; for example, changing a number or a direction to turn. However, when students had to be explicit and exhaustively check for ambiguity and for additional bugs, they were less successful.

Relationship of the Cognitive Demands Measures to Programming Proficiency

Analysis of the relationship between these cognitive demands tasks and the assessments of programming proficiency yielded an interesting set of results. As can be seen in Table 20.3, the cognitive demands measures correlated moderately with composite scores on both tests of programming proficiency.

Examination of correlations with subscores on the programming production task showed that students' ability to write an adequate, runnable program was less highly correlated with cognitive demands measures than were appropriate use of variables, the use of subprocedures within programs, or the use of a minimum number of commands to write programs (one indication of program elegance).

Other subcomponents of the production task that we assumed would correlate highly with the cognitive demands measures (in particular, whether students reused procedures across several programs or used recursion) were not highly correlated. However, so few students engaged in either of these forms of programming that a floor effect may have masked this correlation. Interestingly, although few students used the more advanced programming techniques, many seemed to manifest sufficiently high levels of reasoning skills on the cognitive demands measures. Perhaps other knowledge specific to the programming domain is required in addition to the underlying cognitive capacity to reason in the ways we assessed.

In general, the correlations of the cognitive demands measure were higher with programming comprehension than with programming production. The design of the production task may have contributed to these findings. Students could write linear programs and still succeed on the task, and most did so. This was true even for those who at times in their class projects had utilized more advanced programming techniques. In contrast, the comprehension task required students to display their understanding of sophisticated programming constructs. Thus, although the comprehension task was better able to test the limits of programming novices' understanding of the language, a production task such as the one we employed may prove the better indicator of programming proficiency for students once they attain a more advanced level of ability.

We examined the relation between math achievement level (assigned on the basis of grade-point average, courses taken in school, and scores on math tests administered on the first day of the program) and Logo proficiency. Math level was as good a predictor of programming proficiency as the specific cognitive demands measures taken individually. However, when math level was partialled out of the correlations, they all remained significant at the .01 level or better, with the exception of the correlation between part two of the procedural reasoning task and program production proficiency. Thus, our cognitive demands measures appear to tap abilities that are independent of those directly tied to mathematics achievement.

When both mathematics achievement and performance on our demands measures were entered into a multiple regression analysis, with Logo proficiency as the dependent variable, the multiple correlations were .71 and .52 for programming comprehension and production, respectively. Thus, a quarter to one half of the variability in tested programming proficiency was accounted for by mathematical understanding and specific cognitive abilities bearing a rational relationship to programming.

DISCUSSION

The present study was aimed at identifying the cognitive demands for reaching a relatively sophisticated level of programming proficiency. We examined students learning Logo in an instructional environment that stressed self-discovery within a sequence of structured activities, but with no testing or grading. Given this setting and the amount of instruction, we found that for the most part students managed to master only the basic turtle graphics commands and the simpler aspects of the program control structure. Although they gained some understanding of such programming concepts as procedures and variables, most students did not develop enough understanding of Logo to go beyond the skill level of "effects generation." Thus, for example, although they used variables within procedures, they seldom passed variables between procedures, used recursion, or reused procedures across programs. There was little mastery of those aspects of programming requiring a sophisticated understanding of flow of control and the structure of the language. Without this understanding, students cannot use the powerful Logo constructs that engage and presumably encourage the development of high-level thinking skills.

Nonetheless, we did find moderate relationships between the ability to reason in ways that we had hypothesized would be critical for advanced programming, and performance on our measures of programming proficiency. The magnitude of the correlations indicated that the students who developed most in programming were also those who tended to perform better on tests of logical reasoning. However, our observations of the students during the course of their instruction and their performance on the Logo proficiency measures suggest that, for many students, the actual writing of programs does *not* require that they use formal or systematic approaches in their work. Programming can invoke high-level thinking skills, but such skills are not necessary for students to generate desired screen effects in the early stages of writing programs.

CONCLUSIONS

The field of computer education is in a period of transition. New languages and more powerful implementations of old ones are rapidly being developed, and more suitable programming environments are being engineered for both the new and established languages.

We can best assess the cognitive demands of programming when we are clear about our goals for teaching programming and about how much we expect students to learn. However, to understand the cognitive demands for achieving a particular level of expertise, we must consider the characteristics of a specific language (such as its recursive control structure), the quality of its implementation, the sophistication of the surrounding programming environment (the tools, utilities, and editors available), and the characteristics of the instructional environment in

In conclusion, we have argued that uncovering the cognitive demands of programming is far from simple. On the one hand, programming ability of one form or another is undoubtedly obtainable regardless of levels of particular cognitive skills. On the other hand, if by "learning to program" we mean developing a level of proficiency that enables programming to serve as a tool for reflecting on the thinking and problem-solving processes, then the demands are most certainly complex and will interact with particular programming activities and instructional approaches.

Programming can potentially serve as a fertile domain in which to foster the growth and development of a wide range of high-level thinking skills. However, if this potential is to be realized, studies are needed on two fronts.

First, more work is needed to discover what kinds of instructional environments and direction are best suited for achieving the many goals educators have for teaching programming to children of different ages. We are only beginning to understand how to teach programming. Indeed, many parents and educators who read *Mindstorms* (Papert, 1980) too literally are surprised that programming has to be taught at all. But the unguided, free exploration approach, although effective for some purposes, does not lead all students to a deeper understanding of the structure and operation of a programming language and thus does not lead them to see or develop high-level thinking skills such as problem decomposition, planning, or systematic elimination of errors.

Second, our ability to design effective instruction will depend in part on further experimental work to tease apart the roles various cognitive abilities play in influencing students' ability to master particular programming commands, constructs, and styles. A better understanding of the cognitive demands of using a programming language should help us to focus our instruction and identify those aspects of programming that will be difficult for students. Whereas this study demonstrated a relation between conditional and procedural reasoning ability and programming, we conjecture that, at a more fundamental level, these tasks correlated with programming proficiency because they required the ability to reason in terms of formal, systematic, rule-governed *systems*, and to operate within the limitations imposed by such systems. This may be the major factor in determining whether students will obtain expert levels of proficiency. What remains to be determined is whether extended programming, at proficiency levels below that of the expert, require and/or help to develop high-level cognitive skills and abilities.

ACKNOWLEDGMENT

The work reported here was supported by the National Institute of Education (Contract No. 400-83-0016). The opinions expressed do not necessarily reflect the position or policy of the National Institute of Education and no official endorsement should be inferred.

REFERENCES

Confrey, J. (1984, April). *An examination of the conceptions of mathematics of young women in high school.* Paper presented at the meeting of the American Educational Research Association, New Orleans.

Confrey, J., Rommney, P., & Mundy, J. (1984, April). *Mathematics anxiety: A person-context-adaptation model.* Paper presented at the meeting of the American Educational Research Association, New Orleans.

Hawkins, J. (1983). *Learning Logo together: The social context* (Tech. Rep. No. 13). New York: Bank Street College of Education, Center for Children and Technology.

Kurland, D. M., Mawby, R., & Cahir, N. (1984, April). *The development of programming expertise.* Paper presented at the meeting of the American Research Educational Association, New Orleans.

Kurland, D. M., & Pea, R. D. (1985). Children's mental models of recursive Logo programs. *Journal of Educational Computing Research.*

Luehrmann, A. (1981). Computer literacy: What should it be? *Mathematics Teacher, 74.*

Mawby, R. (1984, April). *Determining students' understanding of programming concepts.* Paper presented at the meeting of the American Educational Research Association, New Orleans.

Mawby, R. Clement, C., Pea, R. D., & Hawkins, J. (1984). *Structured interviews on children's conceptions of computers* (Tech. Rep. No. 19). New York: Bank Street College of Education, Center for Children and Technology.

Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas.* New York: Basic Books.

Pea, R. D., & Kurland, D. M. (1983). *On the cognitive prerequisites of learning computer programming* (Tech. Rep. No. 18). New York: Bank Street College of Education, Center for Children and Technology.

Rampy, L. M. (1984, April). *The problem solving style of fifth graders using Logo.* Paper presented at the meeting of the American Educational Research Association, New Orleans.

Rogoff, B., & Wertsch, J. V. (Eds.). (1984). Children's learning is the "zone of proximal development." *New Directions for Child Development* (No. 23). San Francisco: Jossey-Bass.

Snyder, T. (1984, June). Tom Snyder: Interview. *inCider* (pp. 42-48).

Werner, H. (1937). Process and achievement. *Harvard Educational Review, 7,* 353-368.

TABLE 20.1
Performance of Students on Program Production Tasks

Performance (percentages)	*Figures*						
	A	*B*	*C*	*D*	*E*	*F*	*G*
% who did it	73	60	96	51	91	91	31
workable program	86	47	85	90	91	80	48
variables used	5	43	10	2	14	12	40
repeat used	8	49	65	2	49	84	78
recursion used	0	4	0	0	0	0	8

TABLE 20.2
Debugging Task

	Task			
Bug Type	*No Change*	*Catch No Fix*	*Catch Some Fix*	*Catch & Fix*
	% of students (n = 70)			
Wrong instruction	3	6	na[a]	91
Ambiguous instruction	11	41	na[a]	48
Temporal order bug	16	11	na[a]	73
Embedded bugs	29	10	40	21

[a]not applicable

TABLE 20.3
Correlations of Demands Measures with Measures of Programming Proficiency

Demands Measures		*Measures of Programming Proficiency*						
		A	*B*	*C*	*D*	*E*	*F*	*G*
					(n = 70)			
Procedural reasoning part 1	A	—	—	—	—	—	—	—
Procedural reasoning part 2	B	.34[b]	—	—	—	—	—	—
Debugging task	C	.38[c]	.27[b]	—	—	—	—	—
Math level	D	.51[c]	.38[c]	.42[c]	—	—	—	—
Production proficiency	E	.45[c]	.19[a]	.39[c]	—	—	—	—
Comprehension proficiency	F	.54[c]	.50[c]	.45[c]	.59[c]	.26[b]	—	—
Teacher rating	G	.30[b]	.20[a]	.22[a]	.37[c]	.26[b]	.54[c]	—

[a] $p < .05$
[b] $p < .01$
[c] $p < .001$

FIG. 20.1 Procedural flow of control task to assess students' ability to use procedural reasoning.

FIG. 20.2 Program production task to assess students' skills in planning, problem decomposition, and features of programming styles.

FIG. 20.3 First Logo comprehension task with correct drawing of the resulting screen effects.

FIG. 20.4 Second Logo comprehension task with correct drawing of the resulting screen effects.

FIG. 20.5 Performance of students on program production task.