J. EDUCATIONAL COMPUTING RESEARCH, Vol. 2(4), 1986

A STUDY OF THE DEVELOPMENT OF PROGRAMMING ABILITY AND THINKING SKILLS IN HIGH SCHOOL STUDENTS*

D. MIDIAN KURLAND
ROY D. PEA
CATHERINE CLEMENT
RONALD MAWBY
Bank Street College of Education

ABSTRACT

This article reports on a year-long study of high school students learning computer programming. The study examined three issues: 1) what is the impact of programming on particular mathematical and reasoning abilities?; 2) what cognitive skills or abilities best predict programming ability?; and 3) what do students actually understand about programming after two years of high school study? The results showed that even after two years of study, many students had only a rudimentary understanding of programming. Consequently, it was not surprising to also find that programming experience (as opposed to expertise) does not appear to transfer to other domains which share analogous formal properties. The article concludes that we need to more closely study the pedagogy of programming and how expertise can be better attained before we prematurely go looking for significant and wide reaching transfer effects from programming.

Psychologists, computer scientists, and educators have argued that computer programming can be a powerful means of enhancing thinking and the development of good problem-solving skills in children, in addition to being a powerful method for teaching students fundamental concepts in mathematics, physics, and logistics [1-3]. At first glance, the enthusiasm surrounding programming seems well-founded. Observations of expert adult programmers indicate that

* 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.

programmers explicitly employ important problem-solving strategies such as decomposing problems into modules, use analogical reasoning, and systematically plan, code, and debug their programs. Programming seems to demand complex cognitive skills such as procedural and conditional reasoning, planning, and analogical reasoning [3-7].

In addition to problem-solving skills, programming utilizes fundamental concepts such as variables and recursive structures, which are important in math and physics. It is well-known that these concepts are difficult to teach with traditional media, and their employment in the functional context of a programming language may make them more easily comprehended [1, 3].

Motivated by this enthusiasm for the potential of programming, as well as by the pressure from business and the homes to make students "computer literate," schools have instituted programming courses or related activities at all grade levels. Yet, surprisingly, there has been very little research to date which has directly addressed the many broad claims that have been made for programming. And in addition, there has been very little research examining what students are learning about programming itself as the result of school-based instruction. We know far too little about what to expect students will learn in a reasonable period of time, how they learn, what conceptual difficulties they encounter, what forms of cognitive support they may require to guide them over these difficulties, or whether individual differences in learning styles are reflected in programming and need to be guided differently in instruction. And beyond what rational analyses suggest, we cannot say with much assurance what knowledge and skills that students bring with them when they first meet programming (e.g., natural language competencies; various reasoning skills) are likely to facilitate the acquisition of programming knowledge and skills.

Addressing the issue of how well students actually learn to program in precollege courses is thus an important matter. It is particularly important because of two relations between level of expertise and transfer of learning. First, it is primarily sophisticated programming activities that demand higher level thinking skills and problem-solving techniques, and these activities require expertise. They require a good model of the stucture of the language: you cannot write a modular program and use variables unless you understand how control is passed in the language and how logical tests operate on outputs of operations. Thus, the thinking skills we hope will develop and transfer out of programming depend upon students attaining certain proficiencies in programming [8]. Simple tasks such as a four-line graphics program to draw a box do not require the full range of complex reasoning skills that programming is purported to help develop.

Second, the transfer of concepts and skills across domains depends on the detection of potential similarities between a known and a new domain [9-13]. Brown, Bransford, Ferrara and Campione note that this fact implies differences in

the abilities of novices and experts to transfer since novices and experts classify similarities between tasks differently [14]. Novices will be more limited in their abilities for recognizing problem similarity since they tend to classify tasks according to surface characteristics whereas experts focus on underlying conceptual properties or casual structures. For example, Chi *et al.* examined the categorization of physics problems by novice and experts [15]. Novices categorized problems in terms of particular objects referred to, terminology given in a problem statement, and physical configurations of problem elements. In contrast, experts ignored these superficial features of problems and categorized them in terms of physics principles relevant for problem solutions. Since the novice and expert represent domains differently, they have different information to use in classifying problems, and in accessing knowledge potentially useful in a new problem-solving situation. Similar findings have been obtained for novice and expert adult programmers [16].

Thus, in programming, even if novices begin to develop an understanding of the workings of the language and to write relatively sophisticated programs, they may represent programming in terms of the surface code, format and syntactic properties of a language, and in terms of particular tasks. Experts, on the other hand, are more likely to represent programming problems in terms of the general concepts which underlie particular programming constructs such as recursive routines and variables, the underlying structure of broad classes of problems, the solution strategies which crosscut many types of problems, or routinized plans [17] or "templates" [18] for solving common programming subproblems. Those aspects of programming problem-solving skills we hope will transfer, and that could transfer, involve the general structure of the problem-solving activity and general concepts. Further, the ability to transfer these techniques and concepts from programming will depend on recognizing problems in new domains where the techniques and concepts apply by analogical extension [11, 19].

Whether we are concerned about students learning to think better through programming, or in their learning to program, it is essential to recognize that we are in a very early state of knowledge about the psychology of programming. For this reason, any work in this area, has the nature of work in progress. The technologies available to schools, both hardware and software, are in great flux, and teachers' intuitions are being sharpened through their experiences in helping students learn programming, and to think through programming. So, as useful as any new findings in this area are likely to be for the educator, they must be treated with caution. At the same time, the influence on education of grandiose and optimistic pronouncements that have been made about the cognitive benefits of programming, and on the ease with which students can learn to program, cry out for empirical assessment, even in these early days in the field when the terrain changes faster then one's "research snapshot" develops.

THE PRESENT STUDY

To begin to examine more directly some of the many claims that are being made for and about programming we undertook a study designed to investigate the relation between thinking skills and programming, and to investigate the programming skills acquired by precollege students. We were interested in the development of programming skill among well taught precollege students with significantly more experience programming than most students who have participated in previous research studies.

Our study had three aims. The first was to document the impact of programming experience on a range of reasoning and math skills. The second was to observe the nature of the programming knowledge students attained. The third was to replicate findings from a previous study [20] that certain of these cognitive skills predict some aspects of programming skill acquisition.

Our choice of concepts and skills to investigate was based on a rational analysis of the cognitive components of programming and on correlations, found in previous research, between particular skills and programming mastery [6]. The tasks chosen involved procedural reasoning, decentering, planning, math understanding, and algorithm design.

Our particular task designs were based on an analysis of how the target skills are manifested in programming. Many of the skills we were interested in could not be assessed by standard cognitive ability measures, either because no measures of these skills exist or because existing measures demand skills in a form inappropriate to programming. For instance, standard tests of conditional reasoning examine comprehension of material implication as defined in standard logic. This is not particularly relevant to the use of conditionals in programming; rather the conditional reasoning skill practiced in programming involves reasoning through a complex chain of nested condition-action structures.

METHOD

Design

Three groups of high school students were tested at the beginning and end of the school year. One group of students, the *Experimental* group, was enrolled in their second year of programming. A second group, the *Some-CP* group, had taken one year of programming but had elected not to continue. A third group, the *No-CP* group, had no experience programming.

A battery of *posttests* administered at the end of the year was intended to assess the cognitive benefits resulting from the programming course, and for the *Experimental* group, programming knowledge and skill. Performance on these measures was compared among the three groups of students. The *pretests* administered at the beginning of the year were selected as potential predictors of performance in the programming class. These tests also served as measures of the initial ability level of students on many of the skills were were posttested.

Table 1. Distribution of Subjects in Each Group According to Sex, Grade in School and Grade Point Average

	Sex		*Grade*			*GPA*	
Group	*Male*	*Female*	*10th*	*11th*	*12th*	*Mean*	*Range*
Experimental	11	4	9	3	3	74.3	40-93
No Prior Programming	9	7	8	2	6	78.0	68-96
Some Prior Programming	6	7	4	6	3	77.7	46-94
Total	26	18	21	11	12	76.6	40-96

Students

All students for the study were drawn from a large urban public high school with an ethnically and socio-economically mixed student body. The experimental group consisted of a full class of fifteen students who ranged widely in ability as indicated by their grade point average. Control students were selected from a pool of volunteers and matched with the experimental students on math background, overall GPA, and grade level. Students in the Some-CP group had taken an introduction to computers through the BASIC language course the previous year. Table 1 gives the breakdown of the three groups by sex, grade, and GPA.[1]

Programming Instruction

Students in the experimental group had taken the same introductory course as the Some-CP students. They were currently enrolled in a second, more intensive programming course taught by an excellent programming teacher with five years experience.[2] Class met for forty minutes, five days a week in one of the school's computer labs. Over the year students studied six programming languages. They spent nine weeks each on BASIC, COBOL, and Logo and three weeks each on FORTRAN, MACRO, and Pascal.

[1] The number of students reported for results of certain measures varies since we were unable to administer some tests to one or two students in each group.

[2] The teacher of the Experimental students had a B.A. in Mathematics from Yale University, an M.A. in Interactive Educational Technology from Harvard University, and five years of teaching experience. Her students have won the department's prize exam for first year students in each of her five years, and her AP students placed very highly in national competition and on the Advance Placement Exam.

The nine week Logo section came at the end of the year. While the programming teacher designed and taught the curriculum for the other five languages, we designed, then had the teacher teach, the Logo curriculum. Our aim in designing the Logo curriculum was to help students develop a richer mental model of Logo than students in our previous studies seemed to develop. The focus was on control structure. Work was done solely in list processing—no turtle graphics were taught. In turtle graphics it is too easy for students to continue to generate interesting screen effects without understanding the code [21]. In list processing work, to obtain interesting effects requires a deeper understanding of the language. This approach has its own costs—students need to understand more of the language before they can do much of interest in it.

In our design of the Logo curriculum, we emphasized comprehension over production. Students were given handouts covering concepts and commands, worksheets that stressed program comprehension, and a glossary of Logo primitives, written in flow-of-control language (i.e., in terms of inputs and outputs of Logo command operations). And we supplied utilities for file management to encourage a tool-kit approach.

We designed a series of weekly projects, each building on the previous ones, so that in principle each project could be a modular tool for the next project. The final project was to program a simple version of ELIZA, the program that mimics a non-directive psychotherapist [22]. Topics covered in the course included Logo commands (primitives, procedures, inputs, outputs, and outcomes, creating and editing procedures, words, lists, and list processing, input and output commands, workspace management commands, debugging, trace and error messages, subprocedures, procedures with input variables, naming and creating variables with the MAKE command, the OUTPUT command, conditionals, and tail and embedded recursion.

Measures

The specific rationale and design of each of the tasks used in the study is described fully elsewhere [23]. A brief review of the tasks is provided below.

Pretests – To assess the extent to which skills acquired in programming transfer to other domains, we developed transfer tasks in both "far" and "near" contexts. Our far transfer tasks (the majority of the tasks), demanded skills we believed to be deeply ingredient to programming, but they bore no obvious surface similarities to programming tasks. One near transfer task, in addition to bearing deep structural similarities to programming, resembled programming tasks in several surface features. The pretests were divided into three types: procedural reasoning, planning, and mathematics.

Procedural Reasoning Tests – Rational analysis suggests that programming requires instrumental reasoning, particularly procedural reasoning. Designing, comprehending, and debugging programs requires this type of means-ends reasoning.

Programmers must make explicit the antecedents necessary for different ends, and must follow through the consequences of different antecedent conditions. Designing and following the flow of control of a program requires understanding different kinds of relations between antecedent and consequent events, and organizing and interrelating the local means-end relations leading to the final end. Therefore we designed a set of tasks to measure procedural/conditional reasoning with conditional structures in complex contexts. One task was non-verbal and two were verbal. The tasks involved following the flow of control in systems having logical structures analogous to the logical structures in computer languages. The systems involved reasonable though arbitrary and artificial rules to make them analogous to a programming language and to prohibit students' use of prior world knowledge.

Nonverbal Reasoning Task One. This task was designed using non-English symbolisms so that verbal ability and comprehension of the "if-then" connective would not be an inhibiting factor for students.

Students had to negotiate passage through tree diagrams having an embedded conditional structure. The task tapped ability to discover which goals could be legally reached given that a set of antecedents were true, and ability to determine the antecedents necessary to reach given goals.

Passage through the trees required satisfaction of conditions set by nodes in the tree. Each node required a differing logical combination of various shaped "tokens." Nodes with a *disjunctive* structure offered a choice of tokens to be used, and nodes with a *conjunctive* structure required a combination of tokens. Some nodes were combinations of disjuncts and conjuncts.

The task had two parts. In the first part (Part A), for each question students were given a set of tokens and were asked to determine all goals that could be legally reached with that set. The second part (Part B) included additional components aimed at hypothetical reasoning and planning abilities. In some instances many routes were legal but students were encouraged to find the most efficient route. Here we were interested in the student's sense for elegant problem solutions. In other cases students were required to test a large number of possibilities to discover the one legal path.

Verbal Reasoning Task One. The first verbal procedural reasoning task was analogous to the Non-verbal Procedural Reasoning tasks, but given in verbal form. This task used the "if . . . then . . . else" structure often found in programming. The task assessed ability to follow complex verbal instructions consisting of nested conditionals. Students had to understand the hierarchical relations between instructions, e.g., that some condition was only relevant given the outcome of a prior condition.

The task involved following instructions within a precisely defined set of rules (determining a student's tuition from a complex set of rules based on the student's background and current academic level). Like the non-verbal task, students were given different types of questions to test their mastery of the complex

logical structure. In some questions (Type A) students were given a set of antecedents and were asked for the consequence. In other questions (Type B), the goal or answer was given and the student had to determine what antecedent conditions must have been satisfied. Finally, other questions (Type C) asked what are all and only the decisions that must be made in order to determine a particular outcome given partial knowledge about the conditions. These questions required a good understanding of the structure of the instructions. Students had to separate irrelevant from relevant conditions and understand the hierarchical relation among conditions.

Verbal Reasoning Task Two. This task had a complex conditional structure with a number of goals and conditions for satisfaction. The problem had two conditional structures, in addition to the "if . . . then . . . else" structure, that were isomorphic to programming conditionals. There was a "do-until" loop structure, and a structure isomorphic to an "on gosub" or "Jump match" structure where a match between variables determines what you do.

Planning Task. Several analyses of the cognitive components of programming isolate planning as a central activity [4-7, 24]. After defining the problem to be solved, the programmer develops a plan or "structured description" of the processes required to solve the problem [5], that will then be written in programming code. Observations of expert programmers reveal that a major portion of their time is devoted to planning and that they have available many general plan-design strategies. Pea and Kurland provide an indepth discussion of the nature of planning as it is manifested in programming [24].

The task used to assess planning skill was a slightly modified version of that described in Pea, Hawkins and Kurland [21] (also see [24]). The task involved scheduling a set of classroom chores: students had to design a plan which specified the sequence in which chores should be completed in order to clean-up a classroom in as little time as possible. The chores were to be executed by a hypothetical "robot" who responded to a minimum set of commands, and required a specified amount of time to perform specific actions.

This was a computer-based task. A graphics interface depicted a classroom in which the chores were to be done. Students gave English commands to instruct the robot how to clean up the room and the experimenter typed the commands into the computer. Students designed three plans. After each plan, students were told how much time the Robot would take to actually execute the plan.

The programming and nonprogramming students were each further divided into two subgroups. One subgroup received "feedback" after each plan and the other subgroup did not. Although all students were told of the time that would be required to complete their plans, "feedback" students also received a paper print-out of their plan listing each action and the amount of time it required. They were also shown a screen display of the classroom, in which a step by step enactment of the student's plan (the path of the robot as he completed each chore) was carried out under the student's control. We proposed that there may

be group differences in the extent to which students benefited from the feedback information.

The planning task was administrated for both the pretests and posttest. Two types of data yielded by this task were used in the analyses to be reported. One was the time required to execute the students' plans. The second was the planning behavior of the students. This was assessed by their use of the plan monitoring aids, which was recorded by the computer, and the amount of time they spent thinking about their plan, also recorded automatically by the computer.

Math Test. Math ability has been hypothesized to be both a cognitive demand and an outcome of programming experience [6, 25]. Similarities between high school math and programming exist at several levels. Math and programming languages are both formal systems with well-defined syntactic rules. Both employ the concepts of variable and algorithm. At the procedural level, both may demand representing relatively rich situations in relatively abstract formalisms, and then operating on these formalisms to compute an outcome. Math word problems require extracting essential relations from a prose description of a situation and representing them in mathematical terms. Programming involves giving an explicit procedural representation of a desired output.

Thus we included a math task that we felt would be relevant to programming. Since the math backgrounds of our students varied, and we did not want the task to demand special knowledge, we considered the most basic algebraic concept—the use of letters to represent variables. All students had enough math so that this notation was familiar. The task was designed to depend more on the ability to systematically operate with variables, and on insight during mathematical thinking, than on domain-specific mathematical knowledge.

These salient similarities guided our task design. We gathered a set of math problems that tested either grasp of variables, especially evaluating functions, which is analogous to keeping track of a variable in programming, or ability to relate a symbolic expression to a prose description.

We wanted the variables task to reflect the use of variables in programming. Since values of variables are often passed, modified and printed in programming, we chose problems in which students had to determine the values of variables which were defined in terms of other variables. They thus had to evaluate nested functions, following a chain of several variables to reach a constant. To follow the calculation through would be analogous to tracing the value of a variable through the execution of a program.

Posttests

The battery of posttests included mesaures of procedural reasoning, decentering, planning, math ability, and algorithm design and comprehension. All but the algorithm test can be seen as measures of "far transfer:" the tests demanded skills and concepts we believed to be ingredient to programming, but the tests

bore no obvious surface similarities to a programming task. The algorithm task was our measure of "near transfer;" in addition to deep structural similarities to programming, the task also resembled a programming task in several surface features.

Non-Verbal Procedural Reasoning Task Two – This was a slight modification of the non-verbal procedural reasoning pretest. The rationale was the same–to test procedural and conditional reasoning ability in a non-verbal situation in which reasonable though arbitrary rules must be followed.

This task, like the pretest, had two parts. Rules were similar to Part B of the original non-verbal task. However, unlike the previous task, there was no question designed to assess elegance. In Part A of the posttest students were given a set of passes and were asked to find all the goals they could reach with the passes. These questions assessed ability to exhaustively test conditions in order to discover all legal routes through the tree with the given passes.

In Part B, students were given a set of passes and were asked to find the correct path leading to a particular goal. There was only one legal path for each question. Again, students had to plan, and evaluate several possible routes in order to arrive at the legal route. For the first problem, possibilities could be easily reduced if students compared the number of passes they were given with the number required to reach the goal. The second problem was more difficult since more possibilities had to be tested.

Debugging Task – Programming, especially debugging, demands decentering–programmers must differentiate between their knowledge and intentions and the actual programming code the computer reads. This is a common problem for novice programmers [26]. In program construction the programmer must realize the level of explicitness required to adequately instruct the computer, and in debugging must distinguish expectations from what the computer actually executed. We hypothesized that after learning to program, students might be better at writing and debugging instructions in general.

The debugging task required both instrumental reasoning and decentering. Students were required to detect bugs in a set of driving instructions written for another person to follow. Students had to follow the given instructions, assess their accuracy, and correct "buggy" instructions. This required them to use means-ends analysis and temporal reasoning to assess the consequences and connections among temporally ordered actions. Students had to decenter, making a distinction between the subject's and the driver's knowledge, in order to tell whether instructions were sufficiently explicit and accurate. Bugs included were:

1. *Ambiguous information bug* – instructions not sufficiently explicit to enable the driver to correctly make a choice between alternative routes.
2. *Temporal order bug* – one instruction was stated at the wrong time.
3 *Insufficient or missing information bug.*
4. *Complex bugs* – problems due to unusual input conditions, and embedding, in which obvious corrections fail because they introduce and/or leave a bug.

For each line of instructions with a bug, students were scored for whether they caught the bug, and whether they correctly rewrote the instruction (fixed the bug). For lines of instruction not containing a bug, students were scored for whether they left the line unchanged, or instead inserted information which resulted in a new bug.

Math Test – The math posttest focused on calculating values of variables and translating prose descriptions into symbolic expressions. The rationale was that by programming in six different languages students would have explicit knowledge of variables and considerable practice in setting up equations with variables and tracing the calculation of values of variables.

We used three symbolic expression problems that have been used by Erhlich, Abbot, Salter, and Soloway in studying the transfer of programming abilities to math tasks [27]. The tasks gave prose descriptions and asked for an equation that expressed the same information. For one of the problems we gave students a partial equation to be completed. Ehrlich *et al.* gave programmers and nonprogrammers partial equations of different forms, and found that the advantage of a programming background was most evident when the equation was written with a single variable on one side, e.g., $R = 3/4 \times D$, rather than when written as a multiple expression, e.g., $4R = 3D$. Ehrlich *et al.* suggested that programmers benefited from the single variable expression because in programming one thinks of an active generation of a value, rather than a static description of a relationship.

Two of the three variable problems were the same as given on the math pretest. The third was a simpler problem, based directly on the sort of functional evaluation one finds in Logo list processing, i.e., "$A = B + 1; B = C + 10; C = D + 100; D = 0$; What is the value of A?" Because of poor average performance on the pretest we sought to reduce the difficulty of the easiest problems.

Algorithm Design and Analysis Task

This task assessed comprehension and production of an algorithm within a task designed to closely resemble a programming task. An *Analysis* part asked students to understand a program-like algorithm or "plan;" a *Design* part asked them then to develop an improved algorithm of their own. The task employed a meaningful rather than an abstract programming-language, but its structure resembled the structure of a computer program with sub-routines. The steps of the algorithms were functionally equivalent to programming language commands, as the task description will make clear. Thus, the task served both as: 1) a measure of general algorithmic concepts and skills employed in programming, which might develop through programming, and 2) a measure of "near" transfer to test whether skills employed in programming transferred more readily when the task structure is more transparently analogous to a program.

Students were presented with a goal and a series of legal operators (e.g., count, increment, decrement, test). The algorithms consisted of organizing the

operations in such a way so as to achieve the goal efficiently. Efficient correct algorithms had to have a looping structure. Students were given one algorithm, a looping structure with two flaws that made it inefficient. They were asked to calculate the time required to achieve the goal if the algorithm were executed (assuming 10 seconds per operations). This required students to understand the algorithm. Students were then asked to devise a better algorithm. The students' algorithms were scored for the overall design, use of an iterative structure, accuracy, and conformity to the rules of the system.

Programming Skill Measures

Measures of the programming skills of students in the experimental group included both final scores on regular class tests given for each language, and a specially constructed test administered at the time of the posttest. The teacher designed the regular class tests with the exception of a "Conditional-Iteration" question designed by us and included on the final test for four of the languages.

The Conditional-Iteration question was designed to assess procedural reasoning and understanding of variables within each of the languages taught. For this question, students were asked to show the output (values of variables) of a short program which was structurally analogous, across the four languages tested (BASIC, FORTRAN, COBOL, Logo). Success required ability to follow the flow of control and keep track of variable values. Each program had an iterative (looping or recursive) structure, multiple variables, and conditional tests. The antecedent of each conditional test evaluated the value of a variable; the consequent either incremented a variable, stopped an iteration, or printed a statement. Like the problems on the math tests which asked students to evaluate variables, many variables in these programs were defined in terms of each other, rather than in terms of constants. To fully test students' understanding of how control is passed and of the order of execution of statements, each program contained a line of code designed to detect students' "order bugs," misconceptions of flow of control. This was a conditional test whose consequent prints a statement, but whose antecedent is never satisfied given the order of execution of commands. If students included the statement as output in their answer, they did not correctly follow the flow of the program. A correct answer to each problem displayed the correct values of three variables printed after three iterations.

Logo Test – The second programming measure was a comprehensive Logo test designed by us and administered by the classroom teacher as the students' final exam. This test assessed program comprehension and program production. The program comprehension questions included:

1. A *matching task*: examples of Logo expressions must be identified as expressions of a certain kind. For example, given the expression: "*A*, does Logo read this as a word, number, list, procedure, variable or primitive?

2. A *flow of control task*: students must show the screen effects for a procedure containing three subprocedures each of which prints a certain word.
3. Four *program comprehension tasks* which focused on list processing primitives, the MAKE and OUTPUT commands, tail recursion, and embedded recursion, respectively. Students needed to show the screen effects for four, 2- to 4-line programs, written with these structures. Each program contained local variables, and students were given particular input values.

The program production part of the task required students to write programs (on paper) to generate three given screen effects. They were told to use separate super-procedures to generate each display, but that the super-procedures could share subprocedures. They were also to give the appropriate run commands. An example task was given. The first screen effect the students were to generate was a display of two short English sentences; the second was identical to the first with two sentences added; the third screen effect was identical to the second with the exception that the subject and object of the added sentences was different. Thus, an ideal approach to this task involved the creation of two subprocedures. One would produce the first screen effect. The second would produce the remaining two sentences for the other two effects, by using variables for the subject and object of the sentence.

PROCEDURE

All groups of students received all the pre- and posttest measures, with the exception of the measures of planning skill, and programming skill. The planning task was only administered to the experimental group and to the *No-CS* control group.[3] The programming tests were only given to the *Experimental* group.

Pretests were given during the first month of classes in the fall and the posttests were given during the last month of classes in the spring. We were able to give the Experimental group and most control students the math and procedural reasoning tasks during class time; other students were given the math and reasoning tasks individually. The planning task was always individually administered. All tasks, with the exception of the Planning Task, were administered under time constrainted conditions (5 to 17 minutes per task).

RESULTS

The study was designed to address three questions:

1. Did learning programming for two years occasion advances in reasoning and math skills? Did these second-year programming students perform better, at

[3] The planning task was individually administered. Consequently, logistics did not permit administration of this task to both control groups.

the end of the year, on tasks assessing reasoning and math skills, than students who had only one introductory course?
2. Were certain math and reasoning skills good predictors of success in a programming course? What were the correlations between performance on reasoning, math and programming tasks?
3. Were students able to program at an advanced level after their second year of programming?

Performance of Programming and Nonprogramming Students on Pretest Measures of Reasoning, Math and Planning Skills

To make meaningful *posttest* comparisons between programmers and nonprogrammers, we first examined the comparability of the groups in terms of the skills measured. One purpose of our pretest battery was to make this assessment. The pretests were designed to measure many of the same skills as the posttests, and in two instances the pre and post measures were identical. We compared the three groups on the pretests using analyses-of-variance. Also, correlations between pre- and posttests were examined to provide evidence for the underlying equivalence of the measures.

To conduct these analyses, composite scores were computed for each pretest measure. The analyses-of-variance on each composite showed there were no significant differences between groups for any measures.

The means and standard deviations for the math pretest scores are shown in Table 2. One score consisted of the combination of the two variables problems. Another score consisted of performance on the remaining three questions. As shown in the table, performance was generally low and highly variable. Students had difficulty computing the values of variables except in the simplest cases. They were also generally unable to create the symbolic expression for a word problem.

Table 2. Performance on the Math Prestest: Mean Number of Points in Each Group

	Group					
	Experimental (N = 15)		*Control (Some-CP) (N = 12)*		*Control (No-CP) (N = 16)*	
	Mean	*SD*	*Mean*	*SD*	*Mean*	*SD*
Variables Questions (max = 8)	3.67	2.66	2.50	2.71	2.31	2.85
Other Questions (Max = 8)	1.60	1.12	2.00	1.86	3.06	2.69

Table 3. Performance on theNon-Verbal Procedural Reasoning Pretest: Mean Number of Points in Each Group

	Group					
	Experimental (N = 15)		*Control (Some-CS) (N = 13)*		*Control (No-CS) (N = 16)*	
	Mean	*SD*	*Mean*	*SD*	*Mean*	*SD*
Part A (max = 60)	19.47	19.38	32.39	16.34	28.63	17.85
Part B (max = 6)	.73	1.10	.92	1.71	1.69	2.15

Table 4. Performance on the Verbal Procedural Reasoning Pretests: Mean Number of Points in Each Group

	Group					
	Experimental (N = 15)		*Control (Some-CS) (N = 13)*		*Control (No-CS) (N = 16)*	
	Mean	*SD*	*Mean*	*SD*	*Mean*	*SD*
Verbal Task 1						
Type A and B Questions (max = 18)	6.40	4.37	9.54	5.09	7.88	5.33
Type C Questions (max = 7)	1.00	.93	2.15	1.14	1.38	1.89
Verbal Task 2 (max = 6)	1.33	1.80	1.31	1.65	1.31	1.92

Table 3 shows performance on composite scores for Part A and Part B of the nonverbal reasoning task. Again performance was fairly low for each group. Students could discover some of the correct goals in Part A (which asked them to discover all possible legal goals given a set of tokens), but were often not exhaustive. For Part B, students were usually able to find a legal, but not the best, path to a goal.

Results for Verbal Reasoning Task One are shown in Table 4. For all of the verbal tasks, performance indicated that all groups of students had difficulty following the complex nested conditionals given in the verbal instructions.

Table 5 shows performance by each group on the planning task. There were no group differences due to feeeback condition, so scores were collapsed for this factor. Two general measures of performance are shown: the amount of time it

Table 5. Performance on the Planning Pretest

	Group			
	Experimental (N = 15)		Control (No-CS) (N = 16)	
	Mean	SD	Mean	SD
Plan Execution Time–				
In Minutes[a]				
Mean Plan Time (across 3 plans)	21.42	1.93	21.73	2.65
Best Plan Time	19.57	1.58	19.67	2.12
Planning Behavior				
Mean "Think" Time–				
In Minutes	33.34	13.12	30.02	9.34
Mean Number of Pauses	4.44	2.61	4.13	1.83
Mean Number of Reviews	1.02	1.19	.56	.69
Mean Number of Checks	.27	.38	.56	.71

[a] Optimal time = approximately 17 minutes.

would take to execute their plans (lower times indicate more efficient plans) and the amount of "planning behavior" on the part of the students. Measures of planning behavior include the amount of time students spent thinking about their plans while creating them, the number of pauses between commands (where a pause was defined as any time a student waited five seconds or more between two consecutive steps in their plan) and the extent to which they took advantage of the plan monitoring aids available: the number of times they *reviewed* a listing of their plan so far, and the number of times they *checked* a list of remaining chores. As shown in the table, there are no differences between groups on any of these measures. This allowed us to compare groups directly on the posttest.

Performance of Programming and Nonprogramming Students on Posttests of Reasoning and Math Skills

Non-Verbal Procedural Reasoning Posttest – Composite scores were developed for Part A and for Part B of the nonverbal reasoning test. Performance on these measures for each group is shown in Table 6. There were no significant between-group differences. As on the pretest, students were often able to discover some of the correct goals in Part A, but tended not to be exhaustive. For Part B, many students were unable to find the one legal path for either one or both of the questions asked.

Table 6. Performance on the Non-Verbal Procedural Reasoning Posttest: Mean Number of Points in Each Group

	Group					
	Experimental (N = 15)		Control (Some CS) (N = 13)		Control (No CS) (N = 16)	
	Mean	SD	Mean	SD	Mean	SD
Part A (max = 90)	59.27	24.14	55.08	23.13	58.56	21.23
Part B (max = 2)	.73	.88	.85	.80	.88	.96

Debugging Posttest – Table 7 shows students' performance on the four specific types of bugs. The groups did not differ in their ability to detect or correct any of the classes of bugs. For all groups the temporal order bug was relatively easy to detect. For the remaining types of bugs, students in each group, on the average, were able to detect half of the bugs present. For these bugs, once a bug was detected, most students could successfully correct it. Few students were able to completely detect and correct the complex, embedded bugs.

Planning Posttest – Table 8 shows performance on the planning task. Again, there were no significant differences between groups on any of the measures of plan execution time or planning behavior.

It was of particular interest to compare the groups performance on this task to their performance on it at the beginning of the year. A repeated-measure ANOVA was carried out with Group and Feedback Condition as between-subject variables and Session (pre/post) as a within-subject variable. Mean plan-time (the average of the three plans) was the dependent measure. This analysis revealed that there was a main effect for session–mean plan-times improved slightly overall from the pre- to posttests–but there was no effect for Group, or Feedback Condition, and no interactions. Thus, there were improvements on the planning task over the year but the programming students did not improve any more than the non-programming students, nor did they respond differently to the feedback.

Math Test – As shown in Table 9, no significant differences between groups were found on either the variables problems or the symbolic expressions problems. Thus our findings were not consistent with previous results [27] in which college-level programming appeared to provide advantages for solving word problems given partial equations of the form used here.

A second analysis of performance on the math test involved comparing performance on the subset of those problems which were identical to problems on the pretest. There were eight variable value calculation questions in common between the two tests and the composite scores for these were compared. A

Table 7. Performance by Each Group on the Debugging Posttest: Mean Number of Bugs Detected and Corrected in Each Category

	Group					
	Experimental (N = 15)		Control (Some CS) (N = 13)		Control (No CS) (N = 16)	
	Mean	SD	Mean	SD	Mean	SD
Bug Types						
Ambiguous Information (max = 2)						
Detect	1.13	.83	1.15	.80	1.06	.85
Correct	1.07	.80	1.00	.82	.75	.93
Insufficient Information (max = 4)						
Detect	1.67	1.11	2.00	1.15	1.94	1.18
Correct	1.27	.96	1.77	1.17	1.75	1.24
Temporal Order (max = 1)						
Detect	.80	.41	.77	.44	.69	.48
Correct	.60	.51	.62	.51	.56	.51
Complex (max = 2)						
Detect	.73	.59	.92	.64	.88	.72
Correct	.40	.63	.62	.65	.56	.73

Table 8. Performance on the Planning Posttest

	Group			
	Experimental (N = 15)		Control (No CS) (N = 16)	
	Mean	SD	Mean	SD
Plan Execution Time—In Minutes[a]				
Mean Plan Time (across 3 plans)	20.22	1.69	21.04	1.78
Best Plan Time	18.85	1.23	19.17	1.23
Planning Behavior				
Mean "Think" Time—In Minutes	23.17	12.30	23.73	9.32
Mean Number of Pauses	2.87	2.11	2.79	1.34
Mean Number of Reviews	.40	.46	.56	.51
Mean Number of Checks	.49	.55	.31	.45

[a] Optimal time = approximately 17 minutes.

Table 9. Performance on the Math Posttest: Mean Number of Points in Each Group

	Group					
	Experimental (N = 14)		Control (Some CS) (N = 13)		Control (No CS) (N = 15)	
	Mean	SD	Mean	SD	Mean	SD
Variables Problems (max = 9)	5.64	3.05	5.77	2.31	5.00	2.04
Equation Problems (max = 3)	1.29	1.20	1.15	1.14	1.20	.94

Table 10. Performance by Each Group on the Algorithm Analysis and Design Task: Number of Subjects in Each Response Category for Algorithm Analysis

	Group		
	Experimental (N = 14)	Control (Some CS) (N = 13)	Control (No CS) (N = 15)
Gave approximately correct time	5	2	2
Understood but calculated incorrectly	4	4	9
Response indicates no understanding	3	6	4
No answer	2	1	0

repeated-measures ANOVA (group by session) indicated that posttest performance was significantly better ($F(1,38) = 26.25$; $p < .00$). However, there was no main effect for group nor an interaction. This result was surprising given the degree to which students in the programming course had to work with variables, and the number of different ways they encountered them in their programming tasks.

Algorithm Design and Analysis Test – The two parts of this task—analysis and design of an algorithm—were analyzed separately. Students' ability to analyze an existing algorithm is shown in Table 10. No significant differences between groups were found.

Groups were compared for the style and adequacy of the algorithm they generated. Although there were no between-group differences on an overall composite

Table 11. Algorithm Analysis and Design Task: Number of Algorithms in Each Group Receiving Each Score

	Group		
	Experimental (N = 14)	*Control (Some CS) (N = 13)*	*Control (No CS) (N = 15)*
Scoring Dimensions			
Scope of Intended Design[a]			
No design apparent	5	4	6
Specific to given input	5	4	2
Specific to input of a multiple of 4 coins	2	1	3
General Solution	2	4	4
Used Programming Structures			
Loop	5	2	1
Repeat	2	1	3
Conditional Test[b]	7	3	1
Counter[c]	10	3	2
Structural Errors Present			
In Counter/Counting	12	11	13
In Sequencing	7	5	9
Quality of Design[d]			
No design apparent	5	4	6
Many flaws	1	7	6
Few flaws	7	0	1
Working design	1	2	2

[a] Few algorithms would actually run if executed, but we assessed whether the attempted design was intended to be general or specific.

[b] Chi Square test on number of students using a conditional test = 7.13, $p < .05$.

[c] Chi Square on number of students using a counter = 11.95, $p < .05$.

[d] Chi Square on number of students falling into each quality of plan category = 16.04, $p < .01$.

score, there were differences on some subscores. As shown in Table 11, programming students were more likely to use three of the four programming structures possible: a loop, a conditional test, and a counter (differences in the frequency of use of the latter two structures were significant).

There was also a significant difference in the score for overall algorithm quality. While only one programming student wrote an algorithm that would actually work successfully, many more programming students than nonprogramming students wrote algorithms with only a few flaws. Only one programming student wrote an algorithm with many flaws, although six students in the nonprogramming groups wrote such algorithms.

The picture that emerges from these results is that programming students recognized this task as analogous to programming and could employ some of their knowledge from that domain to construct an algorithm. In comparison to nonprogramming students, they were better able to develop an algorithm which used efficient programming-like constructs, and which could be fairly easily debugged. However, their work was not flawless; there was usually at least one error either in the sequencing, in the use of the counter, or due to violation of the complex task constraints, which prevented their algorithms from actually working. They also did not usually write a general algorithm which would work for any number of input values.

Correlation of Math and Reasoning Pretests with Posttests

The math and reasoning pre- and posttests were almost all significantly correlated, even with grade point average partialed out. Results are shown in Table 12. (Math pretest scores are presented in Table 9.)

Correlation of Math and Reasoning Pre- and Posttests with Programming

We correlated performance on pre- and posttests with a composite of the test scores for each language and with subscores on the Logo tests. Table 13 shows correlations with the composite test scores. The procedural reasoning pretest scores and the math variables pretest score correlated significantly with the

Table 12. Correlations Between Pretests and Posttests for All Subjects; Grade Point Average is Partialed Out ($N = 44$)

	PRETESTS		
	Procedural Reasoning		
	Non-Verbal	*Verbal*	*Math*
Posttests			
Procedural Reasoning Non-Verbal 2	.45*	.64*	.72*
Debugging Test	.56*	.60*	.61*
Math	.39*	.69*	.74*
Algorithm Analysis	.35*	.50*	.56*
Design	.19	.39*	.23

* $p < .01$

Table 13. Correlation of Performance on Programming Tests with Performance on Pre- and Posttests (Experimental Group) ($N = 15$)[a]

	Composite Programming Tests Score
Pretests	
Non-Verbal Procedural Reasoning	.48*
Verbal Procedural Reasoning	.66**
Math	.77**
Planning (mean time)	.09
Posttests	
Non-Verbal Procedural Reasoning	.65**
Debugging	.63**
Algorithm (analysis)	.85**
Algorithm (design)	.74**
Math	.77**
Planning (mean time)	−.65**

[a] $N = 15$ for correlations with all pretests, and the procedural reasoning and debugging posttests. $N = 14$ for correlations with the math and algorithm posttests.

* $p < .05$

** $p < .01$

programming score. Of the posttests, the procedural reasoning, debugging, algorithm, and math scores correlated significantly with the programming tests. The math-variables pre- and posttest scores, and the algorithm task scores correlated particularly well with the programming measures.

Table 14 shows the correlation of the comprehension and production parts of the Logo Test with the pre- and posttests. Almost all correlations are significant. With grade point average partialed out, the pattern of significance remains essentially unchanged.

The correlations found between procedural reasoning, decentering and programming replicate findings of an earlier study of high school students learning Logo [20]. These skills, as well as the ability to evaluate the values of variables, to translate word problems into symbolic equations, and to design and comprehend an algorithm, appear to be centrally related to the development of programming skill.

Programming Ability in the Experimental Group

All measures of programming skill showed that most students had gained only a modest understanding of any of the languages taught. Only performance on the Logo test will be reported here along with the conditional flow of control question which we included on each of the tests for three other languages. Performance on the Logo test is representative of understanding of other languages

Table 14. Correlation of Performance on Logo Test with Performance on Pre- and Posttests (Experimental Group) ($N = 13$)

	Logo Scores		
	Production	*Comprehension*	*Total*
Pretests			
Non-Verbal Procedural Reasoning	.59*	.68**	.69**
Verbal Procedural Reasoning	.62**	.67**	.66**
Math	.68**	.69**	.70**
Planning (mean time)	.02	−.11	.01
Posttests			
Non-Verbal Procedural Reasoning	.73**	.58*	.70**
Debugging	.71**	.69**	.72**
Algorithm			
Analysis	.84**	.80**	.84**
Design	.66**	.63**	.68**
Math	.57*	.65**	.61**
Planning (mean time)	−.34	−.59*	−.44

* $p < .05$

** $p < .01$

Table 15. Performance of the Experimental Group on the Final Programming Tests in Each Language

	Mean Scores on Each Test	
	Mean	*SD*
BASIC	69.80	18.60
COBOL	64.07	22.91
FORTRAN	57.40	22.26
Logo	53.47	25.08

	Correlations Among Test Scores		
	BASIC	*COBOL*	*FORTRAN*
BASIC			
COBOL	.80*		
FORTRAN	.88*	.74*	
Logo	.88*	.91*	.72*

* $p < .01$

by the students; as shown in Table 15, performance in each language is highly correlated.

Logo Proficiency Test – In general, students exhibited a somewhat confused overall understanding of Logo. For example, when asked to identify Logo expressions as variables, procedure names, words, lists, or numbers, on the average only half of the students correctly identified the expressions. They had greatest difficulty recognizing a variable. On following the flow of control through a short Logo program only five of the students were successful. Several students showed understanding of the passage of control among subprocedures, but they failed to exhaustively follow the passing of control. The remaining students demonstrated no understanding of how the order of execution of the lines in a program is determined by particular flow of control commands and the current value of variables.

On the program comprehension problems, students were successful on a program which required understanding of a simple use of list processing primitives and an input variable. When the problem, requiring understanding of *Make* and *Output*, several understood that the OUTPUT command in a subprocedure passes information to a calling procedure, but they did not understand that the OUTPUT command ends the execution of a procedure. Only four students showed a good understanding of tail recursion and none could follow a program involving embedded recursion (also see [28]).

On the program production part of the Logo Test, approximately half of the students could produce the correct screen effects. Only three students used variables and only five students used subprocedures, even though much of the content for the three screen effects was similar or identical, and effectively created a demand for use of subprocedures and variables. The reamining students wrote linear "brute force" programs (lists of outputs preceded by the print command) or were unable to approach a successful program.

Overall, more students evidenced comprehension of variables and flow of control when given short simple programs than when asked to produce their own programs. This may indicate both the fragility of their understanding, and a lack of appreciation of the utility of variables and subprocedures.

Conditional Iteration Problems in Four Languages – Few students could successfully produce the output of short programs given in each language which had an iterative structure with conditional tests and multiple variables to be incremented. Many did understand lines of code which contained a conditional stop rule, incremented a variable, or created a global variable (in Logo). However, they often seemed to evaluate these lines out of context; many could not follow the flow of control and did not demonstrate understanding of the order of execution. A few students followed the order correctly but stopped the programs too soon (wrong number of iterations); they seemed to interpret the comparatives in the stop rule incorrectly, or to have some other difficulty with flow of control.

With the exception of the Logo version of this task, most students could not systematically keep track of values of variables. This was easier in Logo, where the program did not compare variables to each other (i.e., IF $X = Y$) but instead compared them to constants (i.e., IF $X = 3$). Also, the Logo task had only three variables whereas the other tasks had a fourth, counter variable. It is important to note that the "variables" problems on the math pre- and posttests were analogous to this programming problem. There were four variables, most of which had values defined in terms of other variables. Each of these tasks demanded a systematic approach in order to reduce working memory load. Students did not demonstrate skills for such systematic reasoning in their programming nor in their solving of the math problem.

DISCUSSION

Programming students were found to range greatly in their understanding of even basic programming principles and commands. For the most part they exhibited a weak understanding of flow of control or of the structure of the languages in which they worked. Observations of the students as they worked in their programming class indicated that students frequently shared ideas and code (compare [29]). While exploiting pre-written code is an honorable programming technique among professionals, in schools it is a double-edged sword. Some students relied on the understanding of a few good students and never bothered to learn the material themselves. Many students used a trial-and-error approach to a task, or immediately asked for help when stuck. Though several were concerned with understanding what to do, they did not seem to have techniques or rules for systematically analyzing buggy programs and for developing corrections.

Given the generally low level of programming understanding, even after two years of instruction, it did not come as a major surprise that there were no significant differences between the experimental and control groups for any of our measures of "far" transfer. This was the case even though our reasoning and math measures correlated with programming mastery indicating, as expected, that programming taps a number of specific complex cognitive skills.

The transfer tasks all proved to be fairly difficult for most students regardless of programming experience. Students had difficulty with exhaustive and accurate procedural reasoning, evaluating variables defined in terms of other variables, setting up an equation for a word problem expressing proportional relations, decentering sufficiently to exhaustively detect "buggy" instructions and constructing and monitoring a plan for efficient chore execution.

One reason postulated for previous failures to find far transfer still reigns central: students did not attain a very high level of expertise in programming. The significant gains we did find for programming students on the near transfer task—the Algorithm Design and Analysis Task—highlight the important relationship between the nature of knowledge transferred and the acquisition of expertise.

The algorithm task, to which programming students did apply some of their skills, was different from our other transfer measures in two ways: First, the possible knowledge to be transferred included specific programming concepts such as a "counter" and a conditional stop rule, as well as cognitive *operations* used in programming (such as procedural reasoning, or the systematic evaluation of variables). Second, it bore relatively obvious similarities to a programming task (the goal was to perform numerical computations given a set of functions analogous to programming commands or subprocedures). We found that students recognized the conditions for application of some of their programming concepts to the task. Also, to some extent they showed superior procedural reasoning ability; their overall plan quality was better than the nonprogramming students though many still made procedural errors.

These positive results exemplify the tight relation between transfer and what has been learned. The *concepts* transferred—the use of a counter, a loop and a conditional stop rule—are salient features of programming, explicitly represented in the code, and presented early in programming instruction. Thus they are familiar to, even if not fully mastered by, most novices. Given the transfer of the operational skills here and not to our other tasks, it is apparent that relatively context specific rather than general operations were learned.

Because most students' knowledge of the fundamental aspects of programming was quite limited, we do not conclude that development and *far* transfer of skills from programming cannot in principle occur. We can conclude, however, that such far transfer is unlikely to occur given the type of programming curriculum and amount of experience provided for these students, which if anything is misrepresentatively rigorous and unrealistically more intensive than that found in most schools today. Such experience is insufficient for mastery of the programming concepts and practices that engage and make more probable the far transfer of high level thinking skills. Until a population with greater programming expertise is studied longitudinally, the far transfer question remains open.

In conclusion, two things seem clear. First, mastery of at least basic programming skills appears to be essential for transfer, but is hard to achieve within the constraints imposed by the organization of schools. And second, if programming is to continue to play such a major role in the school curriculum, we need to develop much more effective ways of teaching children to program. Explicit devices for helping students see how flow of control structures work appear promising [30-33]. But better programming environments are not enough by themselves. Instruction must explicitly focus on helping the student build a model of how the programming language works. If the operation of the language is a mystery [26], then students cannot write complex and cognitively demanding programs. Early on and throughout instruction, understanding the control and data structures should be stressed. Trial-and-error creation of screen effects typical of pure discovery learning environments common in precollege programming should be tempered with directed teaching of the principles which underlie the effects.

Trial-and-error generation of screen effects neither engages high level thinking skills nor supports increased mastery of the language.

Another possibility often proposed is that, since transfer of thinking skills may involve representation of knowledge at a high level of abstraction, divorced from particular contexts, one might teach thinking skills at this general level of abstraction. Perhaps in this way the need for domain expertise can be bypassed. Unfortunately, we know from previous research that "methods without content are blind," that students have great difficulty deducing examples to which general thinking skills or rules they are taught will apply if they are presented with abstractions alone [34-38]. Insofar as instruction in general thinking skills programs has been effective in promoting transfer, it appears that there have been *explicit conditions for transfer designed into the instructional programs*, including multiple examples of skill application, links to real-world problem solving situations, content area instruction, abstract descriptions of thinking skill methods, and so on [39, 40]. These issues are too complex for treatment here, but will be important to systematically consider in future instruction and research with the aim of helping students learn generalizable thinking skills such as planning and problem solving methods through computer programming activities.

From our perspective, based on data from the present study and others [20, 21, 41-43], we do not believe that the current hope for *incidental* learning of generalizable thinking skills through programming is realistic, and would take these broader lessons about conditions for transfer of learning from the psychological literature into account in designing for transfer in the future. Whether with better programming environments, better instruction, and more explicit attention to designing instruction for transfer, programming will begin to more fully live up to its potentials and promises remains to be seen.

REFERENCES

1. S. Papert, *Mindstorms*, Basic Books, New York, 1980.
2. W. Feurzeig, S. Papert, M. Bloom, R. Grant, and C. Solomon, *Programming Languages as a Conceptual Framework for Teaching Mathematics* (BBN Report No. 1889), Bolt, Beranek, and Newman, Cambridge, Massachusetts, 1969.
3. R. S. Nickerson, Computer Programming as a Vehicle for Teaching Thinking Skills, *Thinking: The Journal of Philosophy for Children*, *4*, pp. 42-48, 1982.
4. R. E. Brooks, Towards a Theory of the Cognitive Processes in Computer Programming, *International Journal of Man-Machine Studies*, *9*, pp. 737-751, 1977.
5. R. Jeffries, A. A. Turner, P. G. Polson, and M. E. Atwood, The Processes Involved in Designing Software, in *Cognitive Skills and Their Acquisition*, J. R. Anderson (ed.), Erlbaum, Hillsdale, New Jersey, pp. 255-283, 1981.

6. R. D. Pea and D. M. Kurland, *On the Cognitive Prerequisites of Learning Computer Programming*, (Technical Report No. 18), Center for Children and Technology, Bank Street College of Education, New York, 1983.
7. N. Pennington, *Cognitive Components of Expertise in Computer Programming: A Review of the Literature*, (Technical Report No. 46), University of Michigan, Center for Cognitive Science, Ann Arbor, 1982.
8. R. Mawby, *Proficiency Conditions for the Development of Programming Skill*, paper presented at the International Conference on Thinking, Harvard University, Cambridge, Massachusetts, August, 1984.
9. J. G. Carbonell, *Derivational Analogy: A Theory of Reconstructive Problem Solving and Expertise Acquisition*, Technical Report CMU-CS-85-115, Carnegie-Mellon University, Computer Science Department, Pittsburgh, March 1985.
10. D. Gentner, Are Scientific Analogies Metaphors? in *Metaphor: Problems and Perspectives*, D. Miall (ed.), Harvester Press Ltd., Brighton, England, 1982.
11. ——, Structure-Mapping: A Theoretical Framework for Analogy and Similarity, *Cognitive Science*, 1983.
12. M. B. Hesse, *Models and Analogies in Science*, University of Notre Dame Press, Notre Dame, 1966.
13. K. J. Holyoak, Analogical Thinking and Human Intelligence, in *Advances in the Psychology of Human Intelligence*, Vol. 2, R. J. Sternberg (ed.), Erlbaum, Hillsdale, New Jersey, 1983.
14. A. L. Brown, J. D. Bransford, R. A. Ferrara, and J. C. Campione, Learning, Remembering, and Understanding, in *Cognitive Development* (Vol. III), J. H. Flavell and E. M. Markman (eds.), of P. H. Mussen (ed.), *Handbook of Child Psychology* (4th edition), Wiley, New York, 1983.
15. M. T. H. Chi, P. J. Feltovich, and R. Glaser, Categorization and Representation of Physics Problems by Experts and Novices, *Cognitive Science*, *5*, pp. 121-152, 1981.
16. B. Adelson, Problem Solving and the Development of Abstract Categories in Programming Languages, *Memory and Cognition*, *9*, pp. 422-433, 1981.
17. E. Soloway, *From Problems to Problems via Plans: The Content and Structure of Knowledge for Introductory LISP Programming*, Technical Report No. 21, Cognition and Programming Project, Yale University, Department of Computer Science, New Haven, Connecticut, 1984.
18. M. C. Linn, The Cognitive Consequences of Programming Instruction in Classrooms, *Educational Researcher*, *14*, pp. 14-29, 1985.
19. D. N. Perkins and G. Salomon, Transfer and Teaching Thinking, in *Thinking: Progress in Research and Teaching*, J. Bishop, J. Lochhead, and D. Perkins (eds.), Erlbaum, Hillsdale, New Jersey, in press.
20. D. M. Kurland, C. A. Clement, R. Mawby, and R. D. Pea, Mapping the Cognitive Demands of Learning to Program, in *Thinking: Progress in Research and Teaching*, J. Bishop, J. Lochhead, and D. Perkins (eds.), Erlbaum, Hillsdale, New Jersey, in press.
21. R. D. Pea, J. Hawkins, and D. M. Kurland, LOGO and the Development of Thinking Skills, in *Children and Microcomputers: Research on the Newest Medium*, M. Chen and W. Paisley (eds.), Sage, Beverly Hills, California, 1985.

22. J. Weizenbaum, *Computer Power and Human Reason: From Judgment to Calculation*, W. H. Freeman, San Francisco, 1976.
23. D. M. Kurland, R. D. Pea, C. Clement, and R. Mawby, *A Study of the Development of Programming Ability and Thinking Skills in High School Students*, Technical Report, with Appendices, Center for Children and Technology, Bank Street College of Education, New York, 1985.
24. R. D. Pea and D. M. Kurland, *Logo Programming and the Development of Planning Skills*, (Technical Report No. 16), The Center for Children and Technology, Bank Street College of Education, New York, 1984.
25. ——, On the Cognitive Effects of Learning Computer Programming, *New Ideas in Psychology*, *2*:1, pp. 137-168, 1984.
26. R. D. Pea, Language-Independent Conceptual Bugs in Program Understanding, *Journal of Educational Computing Research*, *2*:1, pp. 25-36, 1986.
27. K. Ehrlich, V. Abbot, W. Salter, and E. Soloway, Issues and Problems in Studying Transfer Effects of Programming, in *Developmental Studies of Computer Programming Skills*, D. M. Kurland (ed.), (Technical Report No. 29), The Center for Children and Technology, Bank Street College of Education, New York, 1984.
28. D. M. Kurland and R. D. Pea, Children's Mental Models of Recursive Logo Programs, *Journal of Educational Computing Research*, *1*:2, pp. 235-243, 1985.
29. N. M. Webb, Microcomputer Learning in Small Groups: Cognitive Requirements and Group Processes, *Journal of Educational Psychology*, *76*:6, pp. 1076-1088, 1984.
30. D. duBoulay, T. O'Shea, and J. Monk, Presenting Computing Concepts to Novices, *International Journal of Man-Machine Studies*, *14*, pp. 237-249, 1981.
31. B. duBoulay, Children Learning Programming, *Journal of Educational Computing Research*, in press.
32. D. Mioduser, R. Nachmias, and D. Chen, *Teaching Programming Literacy to Non-Programmers: The Use of Computerized Simulation*, (Technical Report No. 15), The Computers in Education Research Lab, Center for Curriculum Research and Development, School of Education, Tel Aviv University, Tel Aviv, Israel, 1985.
33. R. Nachmias, D. Mioduser, and D. Chen, A Cognitive Curriculum Model for Teaching Computer Programming to Children, in *Computers in Education*, K. Duncan and D. Harris (eds.), Elsevier Science Publishers, B. V. North Holland IFIP, 1985.
34. N. Frederiksen, Implications of Cognitive Theory for Instruction in Problem Solving, *Review of Educational Research*, *54*, pp. 363-407, 1984.
35. R. M. Gagne, Learnable Aspects of Problem Solving, *Educational Psychologist*, *15*, pp. 84-92, 1980.
36. R. Glaser, Education and Thinking: The Role of Knowledge, *American Psychologist*, *39*, pp. 93-104, 1984.
37. R. E. Mayer, The Elusive Search for Teachable Aspects of Problem Solving, in *A History of Educational Psychology*, J. A. Glover and R. R. Ronning (eds.), Plenum, New York, in press.
38. A. Schoenfeld, *Mathematical Problem Solving*, Academic Press, New York, 1985.

39. R. D. Pea, *Transfer of Thinking Skills: Issues for Software Use and Design*, paper presented at a national conference on "Computers and Complex Thinking," National Academy of Sciences, Washington, D.C., 1985.
40. L. B. Resnick, *Education and Learning to Think: Subcommittee Report*, National Research Council Commission on Behavioral and Social Sciences and Education, Washington, D.C., NRC, 1985.
41. D. H. Clements and D. F. Gullo, Effects of Computer Programming on Young Children's Cognition, *Journal of Educational Psychology*, *76*, pp. 1051-1058, 1984.
42. J. D. Milojkovic, "Children Learning Computer Programming: Cognitive and Motivational Consequences," Doctoral dissertation, Department of Psychology, Stanford University, 1983.
43. C. Clement, D. M. Kurland, R. Mawby, and R. D. Pea, *Analogical Reasoning and Computer Programming*, paper presented at the Conference on Thinking, Cambridge, Massachusetts, 1984.

Direct reprint requests to:

D. Midian Kurland
Center for Children and Technology
Bank Street College of Education
610 West 112th Street
New York, NY 10025

J. EDUCATIONAL COMPUTING RESEARCH, Vol. 2(4), 1986

A SUMMARY OF MISCONCEPTIONS OF HIGH SCHOOL BASIC PROGRAMMERS

RALPH T. PUTNAM
University of Pittsburgh

D. SLEEMAN
JULIET A. BAXTER
LAIANI K. KUSPA
Stanford University

ABSTRACT

This study examined high school students' knowledge about constructs in the BASIC programming language. A screening test was administered to ninety-six students, fifty-six of whom were interviewed. Students were asked to trace simple programs and predict their output. Errors in virtually all BASIC constructs we examined were observed, with many of the misconceptions arising from the application of knowledge and reasoning from informal domains to programming. It is argued that a lack of knowledge of basic features of programming language will prevent students from developing the higher-level cognitive skills that much programming instruction is intended to foster.

Computer programming courses are often offered by high schools on the grounds that learning programming is a powerful way to develop problem solving and reasoning skills. Linn has suggested that such problem solving skills are the culmination of a chain of cognitive consequences of programming instruction [1]. This chain includes competence with specific features of the programming language being learned, skills for designing programs within the language, and general problem solving skills applicable to other formal systems. While knowledge of specific features of the language being studied is only the first link in this chain, it is a prerequisite to the learning of more general design and problem solving skills. For students to engage in tasks such as debugging programs or designing algorithms by analyzing complex tasks, they must have a certain amount of knowledge about the syntax and semantics of a programming language. A student