9
LOGO and the Development of Thinking Skills

ROY D. PEA

D. MIDIAN KURLAND

JAN HAWKINGS

With the growing presence of computers in educational settings, questions about their importance and likely effects for children's learning have focal concern. Studies that draw conclusions about the impact of computers on children's development and thinking are beginning to emerge. It is important that we take a critical look at the contexts in which these studies are being carried out and at the assumptions that underlie them. Understanding the effects of any learning experience is a complex, multileveled enterprise. Ideally, studying how and what children learn in school contexts should allow for revisionary cycles in which variations in the, important features of learning experiences and methods of measurement can be explored and improvements made. Too often this is not done.

For the past several years we have been carrying out a series of studies conducted to understand in detail one system for using computers with children that has received great attention in the educational community: Teaching children to program through LOGO. The LOGO programming language is designed to be easily accessible to children (Abelson & DiSessa, 1981), and experience with LOGO is associated with general problem-solving abilities as well as with specific skills -in programming (Byte, 1982; Coburn et al , 1982; Papert, 1980). Our research was

AUTHORS' NOTE: We would like to thank the Spencer Foundation and the National Institute of Education (Contract 400-83-0016) for supporting our research and the writing of this chapter. The opinions expressed do not necessarily reflect the position or policy of these institutions, and no official endorsement should be inferred. Our colleagues at the Center for Children and Technology have contributed to these studies in the past several years, and we appreciate their help and support. Of course our unnamed teachers and the LOGO students deserve the lion's share of gratitude for their efforts throughout the research enterprise

designed to answer questions about the cognitive and social impact of LOGO in elementary school classrooms. One major strand of this work is summarized in this chapter: whether learning to program affects the development of other cognitive skills An interwoven theme will be how our, assumptions and understandings concerning the nature of programming and its necessary cognitive requirements changed as we became increasingly familiar with the programming "culture" emerging in the classrooms we were studying.

We began with a basic framework for conducting our work. LOGO was a well-designed smbol system for programming. Many claims had been made about the power and uniqueness of this system as an environment in which children could explore through discovery learning, and develop problem-solving skills that would spontaneously transfer beyond the practices of programming (Papert, 1980). Since this learning environment was being made available on a mass scale, it was important to examine these claims in the contexts of general use-elementary school classrooms. Our intent was to investigate the effects of LOGO learning on cognitive skills (Pea & Kurland, 1984b), but we had the documenting problem of documenting the co-creation of LOGO learning practices in classrooms by teachers and children in which cognitive skills were to be used. In the LOGO discovery learning environment, how did children encounter new information? What were the problems that engaged them? flow was LOGO integrated into the work of the classroom'?

In the next section we briefly review some of the key findings from one dine of our research-the question of whether problem-solving skills were gained- through LOGO programming that transferred beyond – programming practices. However, our main purpose will be to reflect on how these studies enabled us to look more closely at the distinction between the cognitive skills that can be practiced through some uses of , formally elegant symbol systems such as LOGO and the ways that these systems evoke particular practices in classrooms.

RESEARCH SETTING

The studies took place over a _two-year period in one third/fourth-grade and one fifth/sixth-grade classroom in a private school in New York City. The children in the studies represented a variety of ethnic and socioeconomic backgrounds and a range of achievement levels. Many of the children were, however, above national norms in school achievement and came from upper-middle-class and professional families. Each

classroom had six microcomputers during the 1981-1982 school year. In each class, children were learning LOGO.

The teachers received intensive training in LOGO They had regular - contact with members of the research staff as well as members of the team who developed LOGO throughout tile two years of the study. The computer programming activities during the first year were intended by the teachers to be largely child initiated, so as to encourage the child-centered Piagetian learning "without curriculum" advocated for LOGO (Papert, 1980). While teachers in the first .year of the study gave tile children some simple instruction in LOGO during the first several weeks and occasionally held group sessions to introduce new aspects of LOGO during the year, their self-defined rule was principally that of constructively responding to students' questions and problems as they arose. Students primary activities were the creation and development of their own computer programming projects.

Teachers scheduled computer use for students in their classrooms so that everyone would have equal access-about two 45-minute work periods per week. There were additional optional times for computer use throughout the day-before school and during lunch periods-when computers were available on a first-come, first-served basis. Logs kept at each computer over the course of the year showed that, on the average, the children spent about 30 hours programming in LOGO, although several spent as many as 60 hours.

The second year differed from tile first in that both teachers decided to take a more directive role in guiding their students' explorations of LOGO (see Hawkins, 1984b, for a more detailed description of the teachers' changing views of the role of programming in their classrooms). The teacher of the younger class gave weekly group lessons to introduce key computational concepts and techniques, and to demonstrate how they function in computer programs. The older students were also given more group lessons and were required to complete specific assignments centering on LOGO concepts and programming methods, such as preplanning. In both classrooms, the focus of the work remained the development of individual programming projects

In these classrooms, we carried out a number of studies concerning both cognitive and social questions. The studies we will focus upon here concerned the effects learning to program had on students' planning skills. Before examining more closely why we chose planning as one of our key topics, we will briefly discuss the relationship of computer programming to the development of general thinking skills such as planning.

The current claims about effects of learning to program on thinking have been must extensively stated by Papert and Feurzeig (for example, Feurzeig, Papert, Bloom, Grant, & Solomon, 1969; Feurzeig, Horwitz & Nickerson 1981; Goldstein & Papert, 1977; Papert, 1972a, 1972b, 1980; Papert, Watt, DiSessa, & Weir, 1979). Such claims are not unique it) LOGO but have been alleged for programming in general (Minsky, 1970; Nickerson, 1982).

Two key catalysts appear to have contributed to the belief that programming may spontaneously discipline thinking. The first is from artificial intelligence, where constructing programs that model the complexities of human cognition is viewed as a way of understanding that behavior 1 he contention is that in explicitly teaching the computer to do something, you learn more about your own thinking. By analogy (Papert, 1972a), programming students would learn about problem solving processes by the necessarily explicit nature of programming, as they articulate assumption s and precisely specify steps to their problem solving approach. The second influence is the widespread assimilation of constructivist epistemologies of learning, most familiar through Piaget's work. Papert (1972a, 1980) has been an outspoken advocate of the Piagetian account of knowledge acquisition through self-guided problem-solving experiences, and has extensively influenced conceptions of the benefits of learning to program through "learning without curriculum" in "a process that takes place without deliberate or organized teaching" (1980, p.8; also pp. 27, 31). (It should be noted that Piaget never advocated the elimination of organized teaching in schools.)

ON PLANNING

One of the claims made about the positive effects of programming on thinking has been in the area of planning (Feurzeig et al., 1981). From framework it is believed that programming experience will result in greater facility with the art of "heuristics," explicit approaches to problems useful for solving problems in any domain, such as planning, finding a related problem, or solving the problem by decomposing it into parts.

Planning was selected as our principal reference topic because both rational analysis of programming and observations of adult programmers show that planning is manifested in programming in important ways. At the outset of our studies, there was little evidence of how

this symbol system was learned by children in classroom settings. Since there was no information about practice in this "culture," we developed our transfer measures based on a rational analysis of the cognitive requirements of writing computer programs and from examination of the problem - solving activities of expert programmers in settings other than classrooms.

Examination of expert performance reveals that once a programming problem is formulated, the programmer often maps out a program plan or design that will then be written in programming code. Expert programmers spend a good deal of their time in planning program design (Brooks, 1982), and have many planning strategies, available, such as problem decomposition, subgoal generation, retrieval of known solutions, modification of similar code from related programs, evaluative analysis and debugging of program components (for example, see Pea & Kurland, 1983). Does the effectiveness of planning become more apparent to a person learning to program? Does the development of planning skills for more general use as thinking tools become more likely when a person learns to program? And, fundamentally does programming by its inherent nature entail planning as an unavoidable constituent process? These were the questions we set out initially to examine.

PLANNING AND PROGRAMMING

The core of computer programming is that set of activities involved in developing a reusable product consisting of a series of written instructions to make a computer accomplish some task. As in the case of theories of problem solving in general, cognitive studies of programming reveal a set of distinctive mental activities that occur as computer programs are developed These activities are involved throughout the development of a program, whether the programmer is novice or expert, because they constitute recursive phases of the problem-solving process in any general theory of problem solving (see Heller & Greeno,1979; Newell & Simon,1972; Polya, 1957). They may be summarized as follows: (1) understanding/ defining the programming problem; (2) planning or designing a programming solution; (3) writing programming code that implements the plan; and (4) comprehension of the written program and program debugging. We discuss each of these cognitive subtasks in detail elsewhere (see Pea & Kurland, 1983, 1984b).

One may raise the objection that it is possible to bypass planning in program development; that is, one may first make an initial reading of the problem and then compose code at the keyboard to achieve the task.

Although such planning-in-action is certainly possible to produce some programs, it seemed likely that such a plan-in-action :night create pro blems for the inexperienced programmer. While expert programmers can draw on their knowledge of a vast range of plans when creating programs (Atwood & Jeffries, i980; Soloway, Ehrlich, Bonar, Greenspan 1982), the novice programmer has neither the sophisticated understanding of programming code nor the experience of devising succe ssful programming schemas necessary for engaging in planning-in-action.

What are we to make of these observations in terms of defining planning as a distinct cognitive subtask in programming? Is it optional? The answer to this question -certainly has consequences for thinking. about the cognitive outcomes of programming. However, in the absence of any actual observations of *how* novices, especially children (and particularly children engaged in a discovery learning approach), create programs it seemed reasonable to base our predictions about what the potential effects of programming for planning would be on a formal model of programming's, entailments built on this adult model of expert programming.

ASSESSING PLANNING SKILLS

We were guided in the design of our studies by key features of planning processes (see Pea, 1982; Pea & Hawkins, 1984, for further details). Specifically, we felt the tasks should (1) represent situations that are congruent with what is known about plan construction, especially when planning is likely to occur, and (2) externalize the planning process to allow observers to see and record processes of plan construction.

With respect to the former the planning context should (a) be one where a child might be expected to see planning as appropriate and valuable; (b) be complex enough so that the means for achieving a goal are not immediately transparent and the possibility of alternative plans !s : recognized; and (c) involve a domain where children have a Sufficient knowledge base so that action sequences can be planned and consequences of actions anticipated.

With respect to the second point above, the task should reveal (a) whether alternatives are considered; (b) whether the planner tests alter natives by simulating their execution; (c) what kinds of revisions or debuggings of a plan are made; and (d) what different types and levels of planning decisions are made.

Planning is appropriately characterized as a revisionary process. As a consequence of considering alternatives, effective planners revise their plans. They work between top-down planning strategies, -which create a plan from successively refining the a sequence of subgoals for achievement in sequence, and bottom - up ,planning strategies, which note the emergent properties of the plan or the planning environment and add data-driven decisions to the plan throughout its creation (Haves-Roth & Hayes - Roth, 1979; Pea, 1982).

We decided that a classroom chore-scheduling task, analogous to a planning scenario used by Hayes-Roth and Hayes-Roth (1979), met this series of requirements for a planning task. Nonetheless, it constituted a "far" transfer measure because it had very few surface ace similarities to programming for instance, it did not involve a computer. We found from classroom observations that all children had to carry out certain classroom chores on a regular basis (washing the blackboards, watering the plants, and the like;. The task was made novel by requiring children to organize a plan that would allow one person to accomplish all the chores. We designed a classroom map as an external representational model to support and expose planning processes.

A transparent Plexiglas map of a fictitious classroom was developed for the task (see Figure 9.1). Children were to devise a plan to carry out six major chores. The chores could be accomplished with a minimum of 39 distinct chore acts. Some of the acts are subgoals, because hey are instrumentally necessary to accomplish others (that is, the watercan is needed to water plants; the sponge is necessary for washing tables and black boards). Finding the optimal sequencing of these chore acts is thus a challenging task.

STUDYING PLANNING SKILLS: THE FAR TRANSFER TASK OF YEAR ONE

In the first year we videotaped children from the programming classrooms individually (six boys and six girls) and a matched set of same - age controls as they worked in thisplanning environment. Each child was told that the goal was to make up a plan to do a lot of classroom chores. The child was asked to devise the shortest spatial path (for doing the chores, and that he or she could make up as many plans as were needed to arrive at the shortest plan. The child was instructed to think out loud while planning, and to use a pointer to show the path taken to do the chores. The child was given a pencil and paper to make notes (rarely used), and a list of the six chores to keep track of what she

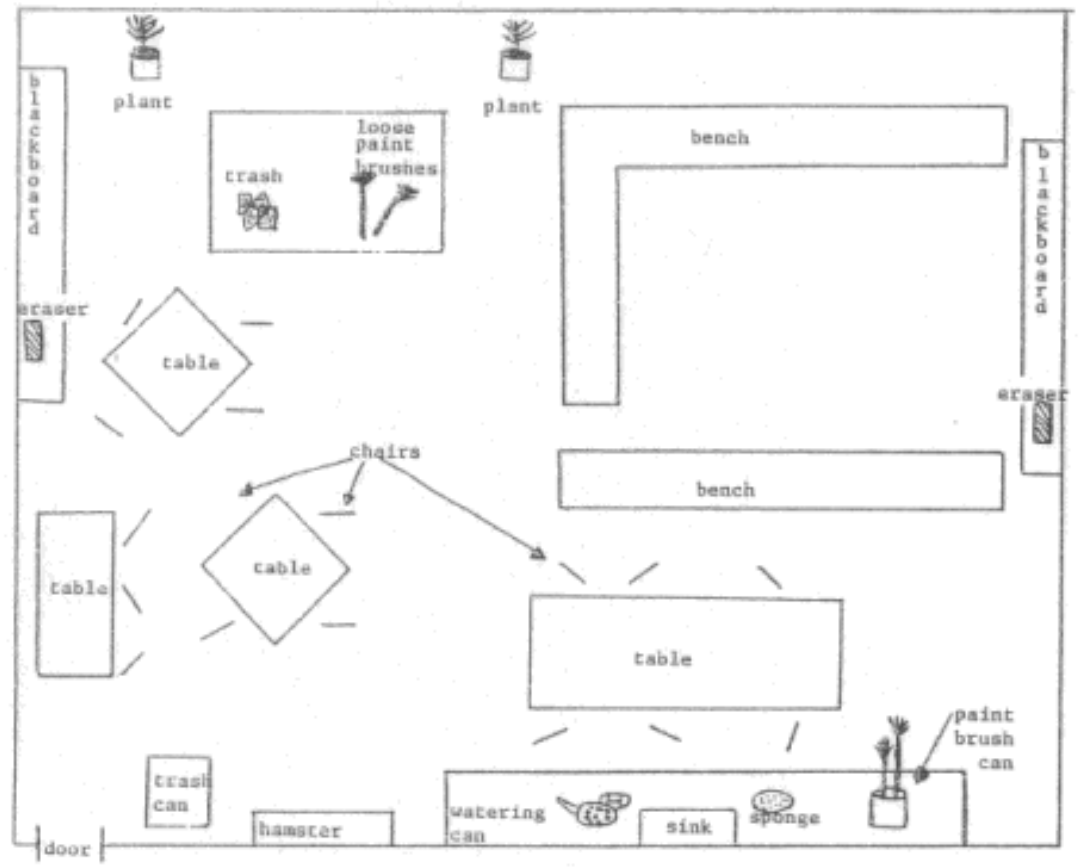

Figure 9.1. Diagram of classroom model, Study 1.

or he was doing. The same task and procedure was administered early in the school year, just as the students were beginning to learn LOGO, and again four months later.

We were interested in examining three aspects of children's plans: (1) the plans considered as products; (2) the plan revisions children made in terms of the features that contributed to plan improvement; and (3) the planning process, especially in terms of the types and levels of abstracted tion or component decisions. On the basis of what programming was assumed to be, these areas were selected because we felt they were the ones most likely to differentiate between the programming and nonprogramming students. Complete descriptions of the analyses and results are available elsewhere (Pea & Kurland, 1984a). Here we will simply review the major findings.

PRODUCT ANALYSIS

The sequence of chore acts for each plan was recorded, and the distance calculated that would be traversed if the plan were to be

executed. Route efficiency for a plan was a function of the distance covered in executing the plan relative to the optimal distance for doing the chores. There were no significant differences in the mean number of plans attempted between children of different ages or between programming and nonprogramming groups.

Route efficiency score significantly increased with age. from first to last plan within session and across age groups. The LOGO programming group, however, did not differ from controls for any plan constructed at the beginning of the school year or at the end of a school year of LOGO programming. Finally, each *age* group, regardless of programming experience improved in efficiency from first to last plan.

Our next question concerned how plan improvements were made. For the most part, we were able to characterize the children's substantive revisions of structure to improve their plans as resulting from "seeing" the chores differently over time. (for e ;ample; see Bamberger & Schon, 1982; DiSessa, 1983; Heller & Greeno, 1979).

More specifically, the initial formulation of our task as the carrying out of a set of named chores ("cleaning tables," "washing blackboards," "pushing in chairs") is a frame or set for problem understanding that must be broken for the task to be accomplished effectively. Performing each named task, in whatever order, is not an effective plan. Each chore must be decomposed into its component acts, and the parts must then be reconstructed and sequenced into an effective all-encompassing plan. 'The child's understanding of part-whole relations for the task is thus transformed during plan revision. To move toward the optimal solution of this planning problem a child must reconfigure the chore "chunks" in terms of their spatial distribution on the classroom map. Major break-throughs in plan structuring occur through discovering spatial clusters--from a list of named chores to a list of spatial clusters of chore acts.

Children's plans were analyzed in terms of these plan features. More efficient organization of chore acts into clusters was highly correlated to shorter plan distance for first and last plans in both sessions.

The mean plan cluster score significantly improved for each age group across plans and sessions, but LOGO programmers did not differ from the control groups on any of these comparisons. The children reorganized their plans into more efficient clusters during the revision process whether or not they had programmed.

PROCESS ANALYSES

We also wished to compare planning processes across children and plans. In creating their plans, did our LOGO programmers engage in

more advanced decision - making process than the nonprogrammers, even though their plans were not more efficient? We examined the process of plan construction by categorizing each segment of the children's think - aloud protocols in terms of the type of planning decision being made and its level of abstraction (as in Goldin & Hayes-Roth 1980; Hayes - Roth & Hayes - Roth 1979).

For the process analysis, we asked whether the organization of the planning process in terms of the types, levels, and sequences of planning decisions was different for the programmers than for the nonprogrammers with respect to the following: (1) frequencies of different types of planning decisions; (2) decision choice flexibility; and (3) relationships between the amount of "executive" and "metaplanning" activity during the planning process and decision-choice flexibility.

In brief, the LOGO - programming group did not differ from the control, groups on any of the comparisons for types of planning decisions. Nonetheless, we found interesting differences in when and by whom such higher-level decisions were made. Children made significantly more high-level decisions in their first plans than in their last in session 1, and older children produced more high-level decisions than did younger children. There were no age effects for the second session,,'

As a further index of planning processes, we determined the flexibility of a child's decision making during the planning process in two ways: (1) by looking at the number of transitions a child made between types of ! decision making while creating the plan, and (2) by looking at the number of transitions made between levels of decision making, irrespective of the decision type. For both sessions, the mean number of type transitions per plan is highly correlated with the mean number of level transitions per plan. The programmers did not differ from the nonprogrammers on these indices of decision-choice flexibility.

RELATIONSHIP OF PRODUCT TO PROCESS MEASURES

We also looked at how decision-making processes were related to the effectiveness of the plan as a product, and found that none of the process and product measures were significantly related. We also tested for a relationship between the frequency of high-level planning decisions and mean cluster scores. The nonsignificant relationships indicate that children revise their plans to accomplish the acts more efficiently without necessarily using (verbally explicit) metaplanning resources. Only fur the last plan of the younger children in the first session are these variables significantly correlated.

DISCUSSION

On the face of it, these results suggest that a school year of LOGO programming did not have a measurable influence on the planning. abilities of these students. While an average of 30 hours of programming is small compared with what professional programmers or college computer majors devote to such work, it is a significant amount of time by elementary school standards.

The failure of the programming students to show any advantage over nonprogrammers on the classroom planning task could have been attributed to any one of a number of possible sources. A prime concern was that our basic assumptions about programming, based on a formal analysis of its properties and expert programmer data, were inadequate for capturing what transpired in the classroom. Based on parallel ethnographic studies in LOGO classrooms (Hawkins, 1983, 1984b), we were beginning to understand that the actual classroom practice of LOGO had developed in ways that made programming activity quite different -from what had been anticipated. For example, particular pieces of students' knowledge about specific programming concepts appeared to be tightly wedded to the specific contexts in which they were learned, unlike the knowledge of expert programmers. Programming constructs for the students had local functional meaning that they did not tend to generalize, even to other closely related programming problems. Although the planning task had features that made it formally similar to the characterization of planning in programming that was available in the literature an programming, the surface structure of the task was quite different from the way programming was actually done in the classrooms. Students may have failed to recognize the task as an opportunity to apply insights from programming.

Therefore, in the second -year of the study we set out to create a new version of the planning task that resembled programming on its surface as well as in its deep structural features. Thus, for example, the new task, while not requiring any previous programming experience (therefore making it suitable for the control groups of students), consisted of a computer-based microworld environment similar to the programming environments with which the students were familiar, and provided on-line feedback on the success of planning efforts analogous to the feedback programmers get from executing their programs in the process of creating them.

In addition, most children appeared to do little preplanning in their programming work. Planning as a component of programming was introduced to the students, but not insisted upon, and possible program-

planning aids (such as worksheets) were not explicitly provided. Students tended to write and revise their code in terms of the immediate effects that commands and sequences of commands produced.

The nature of the LOGO programming environment changed during the second school year. At the end of the first year, teachers expressed disappointment with the quality of students' programming work, and decided to provide more structure to the learning environments for the second year. In addition to conducting "lessons" and group discussions on specific topics, teachers worked with children to develop more suitable individual projects, and at the beginning of the year provided some program-planning aids for the children. These aids, however, were seldom used. Students preferred to write programs interactively at the keyboard.

STUDYING PLANNING SKILLS IN A NEAR TRANSFER PROGRAMMING MICROWORLD

In the beginning of the second year, the original planning task was administered to new groups of students in the two programming classrooms and to two same-age control groups. We found again that students' last plans were better than their first plans, and that there were no differences between the programming and nonprogramming groups at the beginning of the school year.

Near the end of the year, the new planning task was given. This revised task incorporated new design features that made the task bear a far closer resemblance to programming as it was practiced in these classrooms than did the Plexiglas map task. The new task consisted of f our components: (1) a colored diagram of a classroom; (2) a set of goal cards, each depicting one of the six chores (such as wiping off the tables and watering the plant); (3) a microcomputer program that enabled students to design and check their plans with the support of the experimenter; and (4) a graphics interface that enabled students to see their plans enacted in a realistic representation of the classroom (see Figure 9.2).

The computer program created a graphics robot programming and testing environment within which children could develop their plans. The children could "program" a robot using a simple, Englishlike programming language, and then see their plan carried out.

The commands in the robot programming language consisted of a set of six actions (WALK TO, PICK UP, PUT DOWN, WIPE OFF, WATER, STRAIGHTEN UP), and the names for ail the objects in the

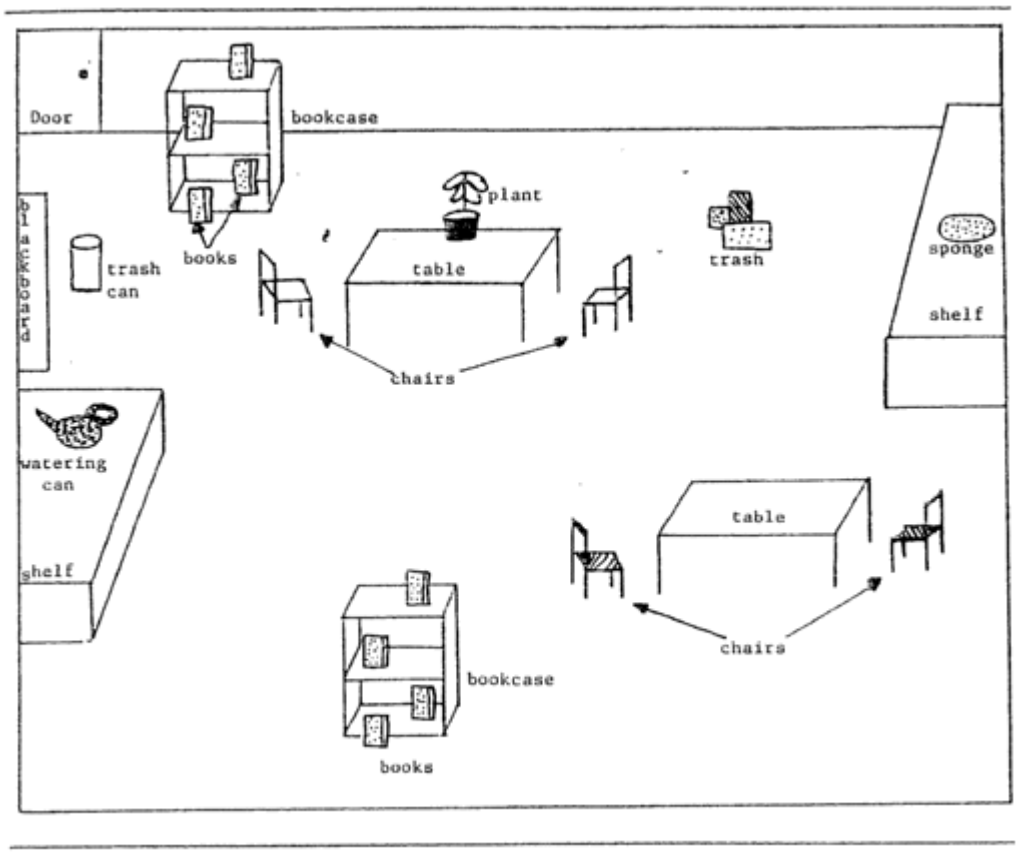

Figure 9.2. Diagram of classroom model, Study 2.

classroom. Each action-object pairing constituted a move in the plan. As the student talked through a plan while looking at the classroom diagram and goal cards, the experimenter keyed each move into the computer, which listed it for the student to see. If the student gave a command that could not be carried out at that point in the plan (for example, telling the robot to wipe off the table before telling it to go to pick up the sponge), the computer program immediately rejected the move and provided a precise context-specific error message on the screen (for example, I'M NOT CARRYING THE SPONGE). If a student indicated that his or her plan was done when there were actually one or more chores still remaining, the program provided a message to this effect, and a list of the outstanding chores appeared on the screen. A message always displayed on the screen informed students that they could at any time ask to see the list of remaining chores or review their plan by having it listed on the screen. Together, these features ensured that all the students would develop runnable, albeit not necessarily optimal, plans,

The second part of the new classroom chore-scheduling task was a graphics interface designed to provide feedback to the student on the

adequacy of his or her plan. There were four types of feedback: (1) a readout oft he total time the student's just-completed plan would take if carried out in action; (2) a representation of a classroom displayed on a high-resolution screen, on which a step-by-step enactment of the student's plan could be carried out under the student's control; (3) a step-by-step readout of each move the student had entered and the time it took tile robot, to carry out each move; and (4) a hard-copy printout of ! he student's plan that Could be referred to during subsequent planning attempts.

In individual sessions children were told to imagine that they had a robot who could understand and carry out commands to perform classroom duties Their task was to devise a plan for the robot to clean up a classroom in the least possible amount of time, covering the shortest possible Spatial path. Students were told that they would create three plans, in which they would be able to improve on their previous plans (Pea & Kurland, 1984a, for further details of the procedure). A clock inside the computer was used to record the intervals between the student's moves ("thinking time"). This enabled us to determine how reflective each student was while creating each plan, and where in the planning process the students spent time thinking.

Students were given as much time as they needed to think about what to do and to call out each individual move. The experimenter typed each move into the computer, where it was either accepted and added to the plan fist or immediately rejected and the student told what was wrong. T he computer did all the monitoring and error checking, and gave tile only feedback the child received. When all the chores were completed and the robot was directed out of the classroom door, the program calculated and then displayed how long the just-entered plan would take.

In order to determine t the effects of feedback from actual plan execution on revisions in later plans, two different task conditions were used. Half o1 the students went on to do a second and then a third plan immediately upon completion Of their first one. The other half of the students saw a representation of the classroom on the graphics screen after they had completed each plan. Simultaneously, the first move of *the* plan was printed on the text screen. 'The student was given a hand-held button that, each time it was pressed, took the program through the plan one move at a time. A line corresponding to each move wa[s] drawn to indicate tire path the robot would follow in carrying Out the plan accompanied by tile name of the move on the text screen (such *as WATER THE PLANT)* A time counter was displayed indicating the total time needed by the robot to carry out the plan up to the current move. The student's plan was printed out so that, when devising subsequent

plans, he or she could see exactly what had been done on the earlier attemts.

We hypothesized that students with programming might differ from their nonprogramming peers in four major respects:

(1) Programmers should be better planners overall Therefore, lengths of plans for the programming students should be less than those for nonprogrammers.

(2) Programmers should make more and better use of the feedback available, since programming teaches the utility of debugging partially correct procedures. This means that programmers should ask more often to see a listing of their plans *(review plan)* and refer more often to the list of remaining chores *(check list)* than nonprogrammers. In addition, in the programming group, differences on these dimensions between students in the feedback and no-feedback conditions should be greater than in the nonprogramming group.

(3) Programmers, relative to nonprogrammers, should spend more time early in their first plan thinking over alternative plans (that is, significantly more pauses and longer mean thinking tune in the first third of the first plan). On subsequent plans, their thinking time should become more evenly distributed across the plan as they concentrate on debugging different parts of it.

(4) Programmers should seek to improve or debug their first plan through successive refinements in subsequent plans s, rather than trying a different approach each time. This means that, relative to the nonprogrammers, the degree of similarity between successive plans for programmers should increase across plans.

Older students produced better (that is, shorter) plans overall than did younger students. In addition, first plans were significantly different from both second and third plans, but the second and third plans did not differ significantly from each other. Even the best group did not produce optimal plans with respect to execution time. There were no differences between the programming and nonprogramming groups in the time their plans would take to carry out. In addition, there was no difference in their use of the available feedback aids such as checking over their; sequence of moves or requesting to see a listing of the remaining chores. Students rarely used these features of the task environment, even though there was a message on the screen at all times indicating its availability. In audition, the group of students who executed their plans between each attempt tended not to spend much time watching the plan enactments nor did they refer to the printed copy of earlier plans when creating a new plan. Plans were created without much attention to the details.

When the pause data (indicating thinking time) were examined, there were again no differences between the programming and the nonprogramming groups. Students paused to think more during the first plan than during their second or third, but the amount of time spent thinking ;n their second and third plans did not differ. When thinking time was broken down into thirds (beginning, middle, and end of the plan), it was found that more thinking time occurred in the beginning third of a plan than into the middle or end third. Thus, while the pattern of thinking time for the programmers conformed to what we had hypothesized, it did not differ as predicted from the pattern for nonprogrammers

Finally, we examined the amount of overlap from plan to plan (plan similarity). The successive plans for all groups teed- fo overlap from plan to plan by 35 percent to 55 percent. Yet again there was no difference between the programming and nonprogramming students or between the students with and without benefit of feedback. Thus there was no evidence that the programmers were more likely to follow a model of plan debugging by successive refinement than nonprogrammers Additional analyses indicate that students who modified previous plans, leaving larger portions intact, did not develop appreciably better plans than students who varied their approaches from plan to plan.

DISCCUSSION

On the basis of these results we concluded that students who had spent a year programming did not differ on various developmental comparisons of the effectiveness of their plans and their processes of planning from same-age controls who had not learned to program. The results from this study are particularly striking because the computerized "near" transfer planning task was designed to have a strong resemblance to programming, including feedback in different representational media (picture of plan in execution, list of moves in plan, and so on), which, because of their planning experience, programmers might have used to greater advantage. The programming groups clearly did not use the cognitive abilities alleged to be developed through experience with LOGO in these tasks designed to tap them.

What were we to conclude from these findings? That there does not appear to be automatic improvement of planning skills from learning LOGO programming appeared clear, but why? Two major categories of potential explanations come to mind.

The first category concerns the design of the transfer tasks. There could be objections to the tasks we used and our resultant data. Perhaps these tasks do not tap planning skills. However, the tasks had greater.

surface validity, and the route efficiency measures in particular were developmentally sensitive. The developmental gap between actual performance and optimal performance could have been influenced by the greater development of planning abilities through programming Yet whether or not a student programmed did not account far the variability we found in planning task performances.

Another objection to our planning tasks was that they are not close enough to programming tasks for the transfer of planning skills from the programming domain. But according to claims made about the general value of programming for thinking, transfer at the concepts and practices of planning to other problem-solving situations should occur *spontaneously* not because of resemblances of the target task to the programming domain.

The second category of explanations concerns the nature of LOGO programming. Here we may distinguish among four different kinds of arguments. First, there are problems with the LOGO programming environment (*not* the instructional environment) as a vehicle for learning these generalizable cognitive skills. Second, the quality of learning about and developing such planning skills with the LOGO discovery learning pedagogy is insufficient for the development of generalizable planning skills. Third, perhaps the amount of time students spent in the LOGO pedagogical environment was not sufficient for us to see the effects on planning of LOGO programming experience.

On the basis of the two studies, we could not tease apart these first three alternatives. However, as we were simultaneously learning more and more about what the students were actually doing in the classrooms - what the practices of programming actually were- a fourth, and fundamentally different, interpretation of these studies became apparent.

To understand this interpretation it is useful to reflect on a sot of issues similar to those we were pursuing in programming-those that relate to the cognitive consequences of *literacy* The acquisition of literacy, like programming today, has long been claimed to promote the development of intellectual skills (Ong, 1982). Prominent historians and psychologists have argued that written language has many important properties that distinguish it from oral language, and that the use of written language leads to the development of highly general thinking abilities, such as logical reasoning and abstract thinking.

But studies bearing on this claim have traditionally been done in societies such as Senegal or Mexico, where literacy and schooling were confounded. Perhaps schooling is responsible for these changes in thinking, rather than the use of written language per se. In an extensive five-year research program, Scribner and Cole (1981) examined the cognitive effects of literacy independently of schooling. The society

studied ¯was the Vai, an African people who do not transmit literacy in the `Jai •written language through formal schooling. Their reading and writing are practiced and learned through the activities of daily life. The Vai invented their written language a mere 150 years ago, and have continued to pass literacy on to their children without schools.

Like most psychologists, Scribner and Cole brought with them standardized psychological testing instruments and stimuli for experiments on concept formation and verbal reasoning. But as Scribner and Core looked over their results from several years of work, they could see no general cognitive effects of being literate in the Vai script. For example, the literate Vai were no better than the nonliterate Vai in categorization skills or syllogistic reasoning.

Before continuing with their initial research strategy with a refined set of tasks, Scribner and Cole realized that there was a radically different way to think about their project, in terms of specific effects. They had begun by looking for general effects of literacy. But after several years of survey and ethnographic observations, they had also come to understand the tasks that Vai literates encounter in their everyday practices of literacy. The Vai use their written language primarily for letter writing, and for recording lists and making technical farming plans. New tasks were designed for assessing literacy effects that were based on those particular skills required by the literacy practices they observed.

Results from these studies demonstrated dramatic cognitive effects of literacy, but they were more local in nature. For example, letter writing, a common Vai literacy practice, requires more explicit rendering of meaning than that called for in face-to-face talk. A communication task where the rules of a novel board game had to be explained to someone unfamiliar with it revealed that performances of Vai literates was vastly superior to those of nonliterates on either version of this task.

Our results concerning the learning of programming can be examined from a similar framework (Pea, 1984b). But for programming languages unlike written language, we do not have the benefit of known historical and cultural changes that appear to result in part from centuries of use of the written language. In the absence of evidence about actual programming practices in these classrooms, we were guided by the rationale that "programming intelligence" and the kinds of programming activities carried out by adults would affect children too.

In addition to examining carefully the formal properties of programming and the planning tasks, we can also take a functional or - activity-based approach to understanding our results. We can consider "programming" not as a *given*, the features of which we know by virtue of how adults do it at its best, but as a *set of practices* that emerge in a complex goal directed cultural framework (Not readable)

(...) and complex an activity matrix as literacy. Just as one may use one's literacy in Vai society to make laundry lists rather than to analyze and reflect upon the logical structures of written arguments, so one may achieve much more modest activities in programming than dialectics concerning the processes of general problem solving, planning, precise thinking, debugging, and the discovery of powerful ideas. One may, in particular, write linear brute-force code for drawing simple pictures.

From a functional perspective we may see that powerful ideas are no more attributes inherent "in" LOGO than powerful ideas are inherent "in" written language. Each may be put to a broad range of uses. What one does with LOGO, or written language, or any symbol system is an open matter. The Vai have not spontaneously gotten into the logical features or written language, philosophy, and textual analysis that written language allows. Likewise, most of our students in these as well as others of our studies from grade school up through high school-have not spontaneously gotten into the programming practices (such as structured planful approaches to procedure composition, use of conditional or recursive structures or careful documentation and debugging) that LOGO *allows*.

For the Vai, one could imagine introducing new logical and analytic uses of their written language. Similarly, one could imagine introducing to children the LOGO programming practices many educators have taken for granted will emerge In either case, we would argue that without some functional significance to the activities for those who are learning the new practices, there is unlikely to be successful, transferable learning.

It is our hunch that wherever we see children using LOGO in the ways its designers hoped, and learning new thinking and problem-solving skills, it is because someone has provided guidance, support, and ideas for how the language could be used The teachers in our studies began to work out such a supportive approach. They found this to be a complex enterprise because they found they had to think through the problems of what Should be known about the system, and the sequence appropriate to comprehension. They also found that helping children to fund functional goals for their LOGO work was problematic throughout the two years

There are many consequences of this general account of what is involved in thinking about LOGO as potential vehicle for promoting thinking and problem-solving skills. A functional approach to programming recognizes that we need to create a *culture* in which students, peers, and teachers talk about thinking skills and display them aloud for others to share and learn from, and that builds bridges to thinking about other domains of schooling

programming projects, would come to play functional roles, not because of some abstract inherent characteristics of programming, but because of characteristics of the context in which programming gets embedded Dialogue and inquiry about thinking and learning processes would became more frequent, and the development of general problem-solving skills so important in an information age would be a more common achievement of students.

Where are we left, then? It is encouraging that there are so many positive energies in education today. The enthusiasm for LOGO as a Vehicle of cognitive change is an exhilarating part of the new processes of education one can see emerging. But we must first recognize that we are visitors in a strange world at the fringe of creating a culture of education that takes for granted the usefulness of the problem-solving tools provided by computers, and the kind of thinking and learning skills that the domain of programming makes so amenable to using, refining, and talking about together.

Learning thinking skills and how to plan well is not intrinsically guaranteed by the LOGO programming environment; it must be supported by teachers who, tacitly or explicitly, know how to foster the development of such skills through a judicious use of examples, student projects, and direct instruction. But the LOGO instructional environment that Papert (1980) currently offers to educators is devoid of curriculum, and lacks an account of how the technology can be used as a tool to stimulate students' thinking about such powerful ideas such as planning and problem decomposition. Teachers are told not to teach, but are not told what to substitute for teaching. Thinking-skills curricula are beginning to appear, but teachers cannot be expected to create them spontaneously, any more than students can be expected to induce lessons about the power of planning methods from self-generated product-oriented programming projects.