Printed in Great Britain

ON THE COGNITIVE EFFECTS OF LEARNING COMPUTER PROGRAMMING

ROY D. PEA and D. MIDIAN KURLAND

Center for Children and Technology Bank Street College of Education, 610 West 112th Street, New York, NY 10025, U.S.A.

Abstract—This paper critically examines current thinking about whether learning computer programming promotes the development of general higher mental functions. We show how the available evidence, and the underlying assumptions about the process of learning to program, fail to address this issue adequately. Our analysis is based on a developmental cognitive science perspective on learning to program, incorporating developmental and cognitive science considerations of the mental activities involved in programming. It highlights the importance for future research of investigating students' interactions with instructional and programming contexts, developmental transformations of their programming skills, and their background knowledge and reasoning abilities.

There are revolutionary changes afoot in education, in its contents as well as its methods. Widespread computer access by schools is at the heart of these changes. Throughout the world, but particularly in the U.S.A., educators are using computers for learning activities across the curriculum, even designing their own software. But virtually all educators are as anxious and uncertain about these changes and the directions to take as they are optimistic about their ultimate effects. "Now that this admittedly powerful symbolic device is in our schools," they ask, "what should we do with it?"

We believe that educators and social scientists are at an important watershed in American education. Important new opportunities abound for research and development work that can influence directly the quality of education. Hard questions are emerging about the *design* of educational activities that integrate the computer with other media. The volatile atmosphere of choices for schools (and parents), as new hardware and software appear daily, calls for principles and knowledge that educators can use, derived from systematic empirical studies, in laboratories and classrooms, of how children learn with these new information technologies. We also need theoretical debates on the aims and priorities for education in an information age. We believe that a developmental

We would like to acknowledge with thanks the Spencer Foundation and the National Institute of Education (Contract 400-83-0016) for supporting the research reported here, and for providing the opportunity to write this essay. The opinions expressed do not necessarily reflect the position or policy of these institutions and no official endorsement should be inferred. Jan Hawkins, Karen Sheingold, Ben Shneiderman and a group of anonymous reviewers provided very useful critical discussions of the data and issues covered in this report. Requests for reprints should be sent to Roy Pea at the address given above.

approach to the understanding of information technologies will be required, one that incorporates the new insights of cognitive science, and that will guide both research on, and design of, computer-based learning environments. Such a discipline of developmental cognitive science would merge theory and practice to dovetail the symbolic powers of human thinking with those of the computer in the service of human development.

In this essay our goals are considerably more modest, but nonetheless a timely subtask of the larger enterprise. Our aim is to examine two widespread beliefs about the mental activities engaged by programming a computer and their expected cognitive and educational benefits. The two beliefs are polar opposites and neither is acceptable. Together, they express the two predominant tendencies in thinking about learning to program today.

The first belief is linked to an atomistic, behaviorist tradition that views learning narrowly. This is the traditional and deeply-engrained idea that learning is simply an accumulation of relatively autonomous "facts". On this view, what one learns when learning to program is the vocabulary of commands (primitives) and syntactic rules for constructing acceptable arrangements of commands. This belief underlies most programming instruction. Its other facet is that what one learns when learning programming is just a programming language.

The contrasting belief, in part a reaction to the first belief, is that through learning to program, children are learning much more than programming, far more than programming "facts". It is said that children will acquire powerfully general higher cognitive skills such as planning abilities, problem-solving heuristics, and reflectiveness on the revisionary character of the problem solving process itself. This belief, although new in its application to this domain, is an old idea in a new costume which has been worn often before. In its common extreme form, it is based on an assumption about learning – that spontaneous experience with a powerful symbolic system will have beneficial cognitive consequences, especially for higher order cognitive skills. Similar arguments have been offered in centuries past for mathematics, logic, writing systems, and Latin (e.g. see Bruner, 1966; Cole & Griffin, 1980; Goody, 1977; Olson, 1976; Ong, 1982; Vygotsky, 1978).

The intuitively plausible claims for the cognitive benefits of programming have broadened in scope and in public attention. Although evidence does not support these claims as yet, their presumed validity is nonetheless affecting important decisions in public education, and leading to high expectations for outcomes of programming in the school and home. In the current climate of uncritical optimism about the potential cognitive benefits of learning to program, we run the risk of having naive "technoromantic" ideas become entrenched in the school curriculum by affirmation, rather than by empirical verification through a cyclical process of research and development. Already at the pre-high school level, programming is taught primarily because of its assumed impacts on higher cognitive skills, not because proficiency in programming is itself an educational goal. This assumption takes on added significance since several million pre-college age children in the U.S.A. are already

receiving instruction in computer programming each year, and France has recently made programming compulsory in their precollege curriculum, on a par with mathematics and native language studies.

With the rapid rise in the teaching of programming it has become critical for decision-makers in education to understand how programming is learned, what may be the cognitive outcomes of learning to program, what levels of programming skill may be required to obtain different types of outcomes, and what the relationships are between the cognitive constraints on learning to program and its cognitive consequences. Research directly addressing these questions is only beginning.

Throughout our paper we will highlight major issues and fundamental complexities for researchers in designing studies responsive to these critical questions. We discuss these issues in terms of a hybrid developmental framework, incorporating cognitive science and developmental psychology, and review relevant research in cognitive science and its cognate disciplines. This synthesis recognizes the inadequacies of either an extreme knowledge-building account of learning to program, or the naive technoromanticism that postulates spontaneous higher order cognitive skills as outcomes from programming experiences. Although claims about the spontaneous cognitive impacts of programming have an intuitive appeal, we show them to be mitigated by considerations of factors involved in learning and development. We also demonstrate how, embodied in practice, the fact-learning approach to programming often leads to incomplete programming skills. Cognitive studies of what expert programmers know, the level of the student's programming skills, the goals and purposes of those learning to program, the general difficulty of transferring "powerful ideas" across domains of knowledge, all contribute to our rejection of these two views. Programming in the classroom *may* fundamentally alter the ways in which learning and cognitive development proceed. But we must examine whether such bold claims find, or are likely to find, empirical support.

We have felt throughout our analysis of these issues that a developmental perspective that incorporates the seminal work in the last decade of the interdisciplinary field of cognitive science will illuminate our understanding of the potentialities of information technologies for advancing human cognition. Fundamental contributions to thinking about and concretely establishing the educational roles of information technologies could be gained from the synthesis of these two important theoretical traditions.

Developmental theorists such as Piaget and Inhelder (1969), Werner (1957) and Vygotsky (1978) have provided accounts of developmental processes with profound implications for the roles of technologies in education. On all these views, cognitive development consists not of an accumulation of facts, but of a series of progressive reorganizations of knowledge driven by the child's active engagements with physical and social environments. In these views, learning (i.e. the accumulation of new knowledge) is important for driving the developmental process, but at the same time is mediated by the current developmental capabilities of the learner.

In the field of cognitive science during the last decade, researchers in the

constituent disciplines of cognitive psychology, computer science, linguistics, anthropology, and philosophy have begun intensive collaborative research projects (e.g. Gentner & Stevens, 1983; Greeno, Glaser & Newell, 1983; Norman, 1981). The combination of careful analysis of cognitive processes and the techniques of computer simulation has led to important new insights into the nature of mental representations, problem solving processes, self knowledge, and cognitive change. Cognitive science has revealed the enormous importance of extensive, highly structured domain-specific knowledge and the difficulty of developing general purpose problem solving strategies that cut across different knowledge domains. Also, within particular domains, cognitive science research has been able to specify in great detail the naive "mental models" held by novices, such as Aristotelian beliefs about objects in motion, which are often very resistant to change through spontaneous world experience (Gentner & Stevens, 1983).

Cognitive science shares with the older tradition of developmental psychology a concern with how new learning must be integrated with prior knowledge, but it transcends earlier work in analyzing problem solving and learning processes for specific knowledge domains, and finds little role for general structural principles invoking "stages".

For a student interacting with a programming environment, for example, a developmental perspective would indicate the importance of studying how these students' current knowledge of the computer system is organized, how they regulate and monitor their interactions with it, and how their knowledge and executive routines affect the ease or pace of acquisition of abilities to use new programming constructs. Also, it would investigate the students' exploration of the system, and the ways that they are able to assimilate it to their current level of understanding and to appropriate it in terms of their own purposes, including play and competition. Learning to use the programming language may require successive developmental reorganizations not only of the students' naive understanding of the language being learned, but also of the computer system as a whole. Complex cognitive changes are unlikely to occur through either spontaneous exploration or explicit instruction alone, since students must be engaged in the task in order to interpret the new concepts. This perspective suggests that rather than arguing, as many currently are, over global questions such as which computer language is "best" for children, we would do better in asking: how can we organize learning experiences so that in the course of learning to program students are confronted with new ideas and have opportunities to build them into their own understanding of the computer system and computational concepts?

In complementary terms, cognitive science raises such important questions as: How can common systematic misconceptions in particular domains of knowledge be diagnosed and remediated through either informal or formal learning activities? For example, what does a student specifically need to know in order to comprehend and use expert strategies in designing a computer program? What component mental processes are engaged in programming activities?

The synthesis of developmental cognitive science focuses on diagnosing the mental models and mental processes that children as well as adult novices bring to understanding computer programming, since these models and processes serve as the basis for understanding transformations of their systems of knowledge as they learn. Beyond the typically agenetic cognitive science, a developmental cognitive science would ask: How are the various component mental processes involved in expert programming constructed and reconfigured throughout ontogenesis, and accessed and organized during problem solving episodes? Through what processes of reorganization does an existing system of thought become more highly developed? Through what learning activities in what kinds of environments does the novice programmer develop into an expert? Developmental cognitive science asks how the mind and its ways of knowing are shaped, not only by biological constraints or physical objects, but by the available cultural interpretive systems of social and educational interaction. As we shall see, the currently available research is impoverished in response to these questions, but current progress in understanding the development of mathematical and scientific thinking (reviewed, for example, in Siegler, 1983) leads us to be optimistic about the prospects for comparable work on the psychology of programming.

The critique of the literature on learning to program that we present below has been strongly influenced by this developmental cognitive science perspective. We do not adopt the usual computer programming perspective assuming that all programming students are adults or have the same goals as mature learners. Instead, the perspective is geared to the learning experiences and developmental transformations of the child or novice adult in interactive environments. The kinds of preliminary questions that we ask from this perspective in addressing the question: "What are the cognitive effects of learning to program?" lead us to draw on studies from diverse fields that we see as relevant to a developmental cognitive science of programming, and we have categorized them according to the topics of "What are the developmental roles of contexts in learning to program?", "What is skilled programming?", "What are the levels of programming skill development?", and "What are the cognitive constraints on learning to program?". First, however, we will begin by examining the bold claims about the effects of learning to program.

CLAIMS FOR COGNITIVE EFFECTS OF LEARNING TO PROGRAM

Current claims for the effects of learning programming upon thinking are best exemplified in the writings of Papert and Feurzeig (e.g. Feurzeig, Papert, Bloom, Grant & Solomon, 1969; Feurzeig, Horwitz & Nickerson, 1981; Goldstein & Papert, 1977; Papert, 1972a, 1972b, 1980; Papert, Watt, DiSessa & Weir, 1979) concerning the Logo programming language, although such claims are not unique to Logo (cf. Minsky, 1970).

Early claims

Two key catalysts underlie beliefs that programming will 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. In explicitly teaching the computer to do something, it is contended that 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 assumptions 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 programming, through "a process that takes place without deliberate or organized teaching" (Papert, 1980, p. 8).

Ross and Howe (1981, p. 143) have summarized Feurzeig *et al.*'s (1969) four claims for the expected cognitive benefits of learning programming. Initially, most outcomes were postulated for the development of *mathematical* thought: "(1) that programming provides some justification for, and illustration of, formal mathematical rigour; (2) that programming encourages children to study mathematics through exploratory activity; (3) that programming gives key insight into certain mathematical concepts; and (4) that programming provides a context for problem solving, and a language with which the pupil may describe his own problem solving."

Papert (1972b) argued for claims (2) to (4) in noting that writing programs of Logo turtle geometry is a "new piece of mathematics with the property that it allows *clear discussion* and *simple models of heuristics* [such as debugging] that are foggy and confusing for beginners when presented in the context of more traditional elementary mathematics" (our emphasis). He provides anecdotes of children "spontaneously discovering" phenomena such as the effects that varying numerical inputs to a procedure for drawing a spiral have on the spiral's shape. He concludes that learning to make these "small discoveries" puts the child "closer to mathematics" than faultlessly learning new math concepts.

Recent claims

We find expanded claims for the cognitive benefits of programming in a new generation of theoretical writings. In *Mindstorms*, Papert (1980) discusses the pedagogy surrounding Logo, and argues that cognitive benefits will emerge from taking "powerful ideas" inherent in programming (such as recursion and variables) in "mind-size bites" (e.g. procedures). One of the more dramatic claims is that if children had the extensively different experiences in thinking about mathematics that Logo allows: "I see no reason to doubt that this difference could account for a gap of five years or more between the ages at which conservation of number and combinatorial abilities are acquired" (p. 175). Papert is referring to extensively replicated findings of a large age gap between the early conservation of number (near age 7) and later combinatorial abilities (e.g. constructing all possible pairings of a set of different colored beads, near age 12).

Feurzeig *et al.* (1981) provide the most extensive set of cognitive outcomes expected from learning to program. They argue that "the teaching of the set of concepts related to programming can be used to provide a natural foundation for the teaching of mathematics, and indeed for the notions and art of logical and rigorous thinking in general." Learning to program is expected to bring about seven fundamental changes in thought:

(1) rigorous thinking, precise expression, recognized need to make assumptions explicit (since computers run specific algorithms);

(2) understanding of general concepts such as formal procedure, variable, function, and transformation (since these are used in programming);

(3) 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, solving the problem by decomposing it into parts, etc. (since "programming provides highly motivated models for the principle heuristic concepts");

(4) the general idea that "debugging" of errors is a "constructive and plannable activity" applicable to any kind of problem solving (since it is so integral to the interactive nature of the task of getting programs to run as intended);

(5) the general idea that one can invent small procedures as building blocks for gradually constructing solutions to large problems (since programs composed of procedures are encouraged in programming);

(6) generally enhanced "self-consciousness and literacy about the process of solving problems" (due to the practice of *discussing* the process of problem solving in programming by means of the *language* of programming concepts*);

(7) enhanced recognition for domains beyond programming that there is rarely a single "best" way to do something, but different ways that have comparative costs and benefits with respect to specific goals (learning the distinction between "process" and "product", as in Werner, 1937).

Asking whether programming promotes the development of higher cognitive skills raises two central issues in developmental cognitive science. First, is it reasonable to expect transfer across knowledge domains? Even adult thinkers are notorious for their difficulty in spontaneously recognizing connections between "problem isomorphs," problems of identical logical structure but

* Hopes that learning the concepts and language that underlie programming will change the way a learner thinks of non-programming problems recalls the strong formulation of the Sapir-Whorf hypothesis: that available linguistic labels constrain available thoughts. The strong form of this hypothesis has been extensively refuted (e.g. Cromer, 1974); only a weak version is consistent with evidence on language – thought relationships. Available labels in one's language may facilitate, but are neither necessary nor sufficient for particular forms of thinking, or conceptual distinctions. Categories of thought may provide the foundation for linguistic categories, not only the reverse. The same point applies to the language of programming.

different surface form (Gick & Holyoak, 1982; Hayes & Simon, 1977; Simon & Hayes, 1976), and in applying strategies for problem solution that they have developed in one context to new problem forms. With problems of "near" transfer so acute, the possibility of spontaneous transfer must be viewed cautiously. In later discussions, we provide a tentative developmental model for thinking about relations between different types of transfer beyond programming, and different levels of programming skill.

The second and related question is whether intellectual activity is guided by general domain-independent problem solving skills or by a conjuction of idiosyncratic domain-dependent problem solving skills (Goldstein & Papert, 1977; Newell, 1980; Simon, 1980). An extensive literature on metamemory development indicates that the tasks used to measure the functioning of "abstract thinking" are inextricably linked to the specific problems used to assess metacognition (e.g. Brown, 1983a). And as Ross and Howe (1981) note, "in most problem solving tasks, it is impossible to apply the supposed context-free skills without initially having essentially domain-specific knowledge." Within domains, however, better performances by learners are commonly accompanied by reflection on the control of their own mental activities (Brown, Bransford, Ferrara & Campione, 1983).

THE DEVELOPMENTAL ROLE OF CONTEXTS IN LEARNING TO PROGRAM

For a developmentalist, there is a major problem pervading each of these characterizations of the effects on higher thinking skills expected from learning to program. Programming serves as a "black box," an unanalyzed activity, whose effects are presumed to irradiate those exposed to it. But questions about the development of programming skills require a breakdown of the skills into component abilities, and studies of how specific aspects of programming skill are acquired. They require especially serious consideration of the developmental roles played by the contexts interpenetrating the black box: the programming environment, the instructional environment, and the relevant understandings and performances of the learner.

The question of the role of contexts in learning "programming" is complex, because "programming" is not a unitary skill. Like reading, it is comprised of a large number of abilities that interrelate with the organization of the learner's knowledge base, memory and processing capacities, repertoire of comprehension strategies, and general problem-solving abilities such as comprehension monitoring, inferencing, and hypothesis generation. This lesson has been etched in high relief through intensive efforts to develop artificial intelligence systems that "understand" natural language text (e.g. Schank & Abelson, 1977; Schank, 1982). Skilled reading also requires wide experience with different genres (e.g. narrative, essays, poetry, debate) and with different goals of reading (e.g. reading for gist, content, style). As reading is often equated with skill in decoding, "learning to program" in schools is often equated with learning the vocabulary and syntax of a programming language. But skilled programming, like reading, is complex and context-dependent, so we must begin to unpack the contexts in which programming is carried out and learned.

Environments in which children learn to read are usually overlooked because adequate environments (e.g. plenty of books, good lighting, picture dictionaries, good readers to help with hard words, vocabulary cards, phonics charts) are taken for granted. By contrast, good programming environments are not generally available to schools. Determining how children develop programming skills will not be possible without due consideration of the programming environment in which learning and development takes place, and of how learning activities are organized.

Programming environment

The distinction between a programming language and a programming environment is crucial. A programming language is a set of commands and rules for command combinations that are used to instruct the computer to perform specified operations. The programming environment, on the other hand, is the larger collection of software (operating systems and programming tools) and hardware (memory, disk storage, hard copy capability) available to the programmer. It can include an editor program to facilitate program writing, code revising, and copying useful lines of code from one program to another; debugging aids; elaborate trace routines for following the program's flow of control; automatic documenters; cross-reference utilities for keeping track of variables; and subroutine libraries.

Good programming environments (for example, those most extensively developed for working on large computers in Lisp and PL/I) make the coding aspect of programming far more efficient, allowing the programmer to concentrate on higher level issues of program design, efficiency, and elegance. In contrast, the programming environments provided for today's school microcomputers are so impoverished (typically consisting of only a crude editor and limited trace functions) that entering the code for a program and just getting it to execute correctly is the central problem.

Finally, despite vigorous arguments about the educational superiority of different programming languages, there are no data on whether different languages lead to significant differences in what children need to know prior to programming, or what cognitive benefits they derive from it. Although such differences between languages may exist, they do not affect our point, since these differences can be manipulated radically by restructuring the programming environment. Attention is best directed to general issues about programming, rather than those that are programming language specific.

Instructional environment

While features of the *programming environment* are important for learning to program, how successfully a child will master programming also depends on the *instructional environment* and the way in which resources such as computer access time and file storage are allocated. Each of these points concerns the context of cognitive activities, which we know from cognitive science and developmental psychology to be critical to the level of performance achieved in cognitive tasks

(e.g. for reviews, see Brown *et al.*, 1983; Laboratory of Comparative Human Cognition, 1983).

Deciding how to introduce programming and assist students in learning to program is hampered today by the paucity of pedagogical theory. That current "fact learning" approaches to programming instruction are inadequate has become apparent from studies of the kinds of conceptual errors made by novice programmers instructed in that way. For example, novice adult programmers reveal deep misunderstandings of programming concepts, and of how different lines of programming code relate to one another in program organization (Bonar & Soloway, 1982; Jeffries, 1982; Sheil, 1980, 1981a; Soloway, Bonar & Ehrlich, 1983; Soloway, Ehrlich, Bonar & Greenspan, 1982). As expected from what they are taught, they know the vocabulary and syntax of their programming language. Their misunderstandings are much deeper (Jeffries, 1982), such as assuming that all variables are global (when some may be specific to one procedure), and expecting that observing one pass through a loop allows them to predict what will happen on all subsequent passes (although the outputs of programming statements which *test* for certain conditions may change what will happen during any specific loop). Research by Mayer (1976), Miller (1974), and Sime, Arblaster and Green (1977) has revealed that adult novice programmers have a difficult time generally with the flow of control concepts expressed by conditionals (for a review of these findings, see duBoulay, O'Shea & Monk, 1981). These conceptual difficulties, even among professional programmers, have been lamented by such programming polymaths and visionaries as Minsky (1970) and Floyd (1979) as due to problems with how programming is taught. Too much focus is placed on low level form such as grammar, semantic rules, and some pre-established algorithms for solving classes of problems, while the *pragmatics** of program design are left for students to discover for themselves. Interestingly, these complaints about writing

---

* One many distinguish for (artificial) programming languages, just as in the case of natural languages, between three major divisions of *semiotics*, or the scientific study of properties of such signalling systems (Crystal, 1980). These three divisions, rooted in the philosophical studies of Peirce, Carnap, and Morris, are "*Semantics*, the study of the relations between linguistic expressions and the objects in the world which they refer to or describe; *syntactics*, the study of the relation of these expressions to each other; and *pragmatics*, the study of the dependence of the meaning of these expressions on their users (including the social situation in which they are used)" (ibid., p. 316). Studies of natural language pragmatics have focused on the "study of the *language* from the point of view of the user, especially of the choices he makes, the *constraints* he encounters in using language in social interaction, and the effects his use of language has on the other participants in an act of communication" (ibid., p. 278).

Although there are important disanalogies to natural language, a pragmatics of programming languages concerns at least the study of programming language(s) from the viewpoint of the user, especially of the (design) choices that he or she makes in the organization of lines of programming code within programs (or software systems), the constraints that he or she encounters (such as the requirements of a debuggable program that is well-documented for future comprehension and modification) in using programming language in social contexts, and the effects that his or her uses of programming language have on the other participants (such as the computer, as ideal interpreter, or other humans) in an act of communication involving the use of the programming language.

programs are similar to those voiced about how writing in general is taught (e.g. Scardamalia & Bereiter, 1983).

What do we know about conceptual problems of children learning to program? Problems similar to those of adult novices are apparent. To take one example, in our research with 8- to 12-year-old Logo programmers (Kurland & Pea, 1983), we find through their think-aloud protocols and manual simulation of programs that children frequently adopt a systematic but misguided conception of how control is passed* between Logo procedures. Many children believe that placing the name of the executing procedure within that procedure causes execution to "loop" back through the procedure, when in fact what happens is that control is passed to a *copy* of the executing procedure. This procedure is then executed, and when that process is complete, passes control back to the procedure that last called it. Children adopted mental models of flow of control which worked for simple cases, such as programs consisting of only one procedure, or tail recursive procedures, but which proved inadequate when the programming goal required more complex programming constructions.

In other developmental studies of Logo programming skills (Pea, 1983), even among the 25% of the children (8- and 9-year-olds; 11- and 12-year-olds) who were extremely interested in learning programming, the programs that they wrote reached but a moderate level of sophistication after approximately 30 hours of on-line programming experience during the year. Children's grasp of fundamental programming concepts such as variables, tests, and recursion, and of specific Logo primitive commands such as "REPEAT," was highly context-specific. For example, a child who had written a procedure using REPEAT which repeatedly printed her name on the screen did not recognize the applicability of REPEAT in a program to draw a square. Instead, the child redundantly wrote the same line-drawing procedure four different times. We expect that carefully planned sequences of instruction will be important to ensure that programming knowledge is not "rigid" (Werner, 1957), or "welded" (Shif, 1969) to its contexts of first learning or predominant use. Such rigidity is a common finding for early developmental levels in diverse domains (Brown *et al.*, 1983).

More broadly, in the National Assessment of Educational Progress survey of 2500 13-year-olds and 2500 17-year-olds during the 1977 – 1978 school year (National Assessment of Educational Progress, 1980), even among the small percentage who claimed to be able to program, "performance on flowchart reading exercises and simple BASIC programs revealed very poor under-

* The concept of "flow of control" refers to the sequence of operations that a computer program specifies. The need for the term emerges because not all control is linear. In linear control, lines of programming instructions would be executed in strict linear order: first, second, third, and so on. But in virtually all programming languages, various "control structures" are used to allow nonlinear control. For example, one may "GOTO" other lines in the program than the next one in BASIC, in which case flow of control passes to the line of programming code referred to in the GOTO statement. Because a program's "flow of control" may be complex, programmers often utilize programming flowcharts, either to serve as a high level plan for creating their program, or to document the flow of control in their program.

standing of algorithmic processes involving conditional branching" (cited by Anderson, 1982, p. 14).

Educators often assume that adult programmers are not beleaguered by conceptual problems in their programming, but we have seen that they are. Once we recognize that programming by "intellectually mature" adults is not characterized by error-free, routine performances, we might better understand difficulties of children learning to program, who devote only small amounts of their school time to learning to program.

These findings lead us to two central questions about programming instruction, which we define broadly to include the direct teaching provided by educators as well as the individual advice, modelling, and use of metaphors with which they support instruction and learning. How much instruction, and what types of instruction, should be offered? How much direct instruction is best for children to learn programming is a controversial question (e.g. Howe, 1981; Papert, 1980). At one extreme schools teach programming as any other subject with "fact sheets" and tests; at the other, they provide minimal instruction, encouraging children to explore possibilities, experiment, and create their own problems to solve. This second approach, popularized by Papert (1980), argues that little overt instruction is necessary if the programming language is sufficiently engaging and simple to use, while at the same time powerful enough for children to do projects that they find meaningful. Though this discovery learning perspective is not universally shared, even by Logo devotees (Howe, 1981), it has had a pervasive influence over uses of Logo by schools.

What *type* of instruction should be offered, and *when* in the course of programming skill development specific concepts, methods, and advice should be introduced are also critical questions. Two central factors are implicated by cognitive science studies. One is the current mental model or system of knowledge that the student has available at the time of instruction. A second is the goal-relevance of the problem solving activity required of the student. On the first point, there are no careful studies of the success of different instructional acts as a function of a student's level of understanding for programming akin to those carried out by Siegler (1983) for such concepts as time, speed, and velocity. At a more general level, Mayer (1979, 1981) has shown that a concrete conceptual model of a programming system aids college students in learning BASIC by acting as an advance organizer of the details of the language. With the conceptual model, learners were able to assimilate the details of the programming language to the model rather than needing to induce the model from the details.

On the second point, we would ask how compatible are the teacher's instructional goals with children's goals and purposes in learning programming? Recent developmental cognitive science and cross-cultural studies of cognition (e.g. Brown, 1982; Laboratory of Comparative Human Cognition, 1983), have shown that assessing task performance within a goal structure familiar to the person is necessary for determining the highest developmental level of an individual's performances. For learning to program, goals of the programming activity need to be contexted for the child in terms of other meaningful and goal-

directed activities, connecting either to everyday world affairs, to other aspects of the curriculum, or to both. Papert (1980) has described this as "syntonic" learning. For example, in our studies Logo classroom children found two contexts especially motivating: creating videogames and simulating conversations. The most intensive and advanced programming efforts were in the service of children's goals such as these. Dewey's (1900) point about the importance for *any* learning that developments in the new skill serve as more adequate means for desired ends thus again receives new support. A similar emphasis underlies the successful use of electronic message and publishing systems in classrooms (e.g. Black, Levin, Mehan & Quinn, 1983; Laboratory of Comparative Human Cognition, 1982). Embedding computer programming activities of increasing cognitive complexity in children's goal structures may promote learning to program and support the transfer of what is learned in programming to problem solving activities in other domains.

Our point throughout this section has been that programming is not taught by computers or by programming languages but by teachers, with the aid of the supports of a programming environment. How effectively children of different ages and with different background knowledge learn programming will be contingent upon the capabilities of their teachers, the appropriateness of their learning activities to their current level of understanding in programming, and the features available in their programming environment. Studies to date have not incorporated these considerations that a developmental cognitive science perspective recognizes as central.

WHAT IS SKILLED PROGRAMMING?

How to define and assess the constellation of skills which comprise programming has long been a major problem for industry (Pea & Kurland, 1983b), and is becoming so for schools. We define the core sense of "programming" as the set of activities involved in developing a reusable product consisting of a series of written instructions that make a computer accomplish some task. But in order to move from definition to instruction, one must begin to unpack "programming skill", in contrast to the black box approach to programming prevalent in schools. Promising moves in this direction have already been provided by careful analyses of what expert programmers *do*, and what types and organizations of knowledge they appear to have in memory that they access during programming. This research strategy, characteristic of cognitive science, has revealed significant general features of expert problem solving skills for diverse domains, such as algebra (Lewis, 1981), chess (Chase & Simon, 1973), geometry (Anderson, Greeno, Kline & Neves, 1981), physics (Chi, Feltovich & Glaser, 1981; Larkin, McDermott, Simon & Simon, 1980), physical reasoning (deKleer & Brown, 1981), and writing (Bereiter & Scardamalia, 1982), and it is providing new insights into components of programming skill. In terms of what a programmer *does*, a set of activities is involved in programming for either novices or experts, which constitutes phases of the problem solving process (e.g. Newell & Simon, 1972; Polya, 1957). These activities, which may be invoked at any time and recursively during the development of a program, are: (1)

understanding the programming problem, (2) designing or planning a programming solution, (3) writing the programming code that implements the plan, and (4) comprehension of the written program and program debugging. An extensive review of these cognitive subtasks of programming may be found in Pea and Kurland (1983b).

In terms of what an expert programmer *knows*, findings on the knowledge schemas, memory organizations and debugging strategies which expert programmers possess are of particular interest. Recent studies of programmers characterize high-level programming skill as a giant assemblage of highly specific, low-level knowledge fragments (Atwood & Ramsey, 1978; Brooks, 1977). The design of functional "programmer's apprentices" such as Barstow's (1979) *Knowledge Based Program Construction*, and Rich and Shrobe's "Lisp programmer's apprentice" (Rich & Shrobe, 1978; Shrobe, Waters & Sussman, 1979; Waters, 1982), and the MENO Programming Tutor (Soloway, Rubin, Woolf, Bonar & Johnson, 1982) has involved compiling a "plan library" of the basic programming "schemas," or recurrent functional chunks of programming code that programmers are alleged to use. Observations of programmers support these introspective analyses of "chunks" of programming knowledge. Eisenstadt, Laubsch and Kahney (1981) found that most novice student programs were constructed from a small set of program schemas, and Jeffries (1982), in comparing the debugging strategies of novice programmers and graduate computer science students, found that experts saw whole blocks of code as instantiations of well-known problems such as calculating change. Soloway and colleagues (Bonar, 1982; Ehrlich & Soloway, 1983; Johnson, Draper & Soloway, 1983; Soloway & Ehrlich, 1982; Soloway, Ehrlich, Bonar & Greenspan, 1982; also see Kahney & Eisenstadt, 1982) postulate a model in which programmers use recurrent plans as "chunks" in program composition, and identified such plans in programs written by Pascal novices (e.g. the "counter variable plan"). But for developmental cognitive science we will need studies of how students mentally construct such plan schemas from programming instruction, experience, and prior knowledge.

A related aspect of programming skill is the set of rules that experts use to solve programming problems, but again we lack genetic studies. In an analysis of a programmer's think-aloud work on 23 different problems, Brooks (1977) demonstrated that approximately 104 rules were necessary to generate the protocol behavior. Similarly, Green and Barstow (1978) note that over a hundred rules for mechanically generating simple sorting and searching algorithms (e.g. Quicksort) are familiar to most programmers.

A third aspect of programming skill is the ability to build detailed "mental models" of what the computer will do when a program runs. An expert programmer can build dynamic mental representations, or "runnable mental models" (Collins & Gentner, 1982) and simulate computer operations in response to specific problem inputs. The complexities of such dynamic mental models are revealed when skilled programmers gather evidence for program bugs and simulate the program's actions by hand (Jeffries, 1982). Not all program understanding is mediated by hand simulation; experts engage in

global searches for program organizational structure, guided by adequate program documentation, a strategy akin to what expert readers do (Brown, 1983b, Brown & Smiley, 1978; Spiro, Bruce & Brewer, 1980). How individuals develop such rich procedural understandings is currently unknown.

Expert programmers not only have available more knowledge schemas, strategies, and rules applicable to solving programming problems, but they perceive and remember larger "chunks" of information than novices. The classic Chase and Simon (1973) finding of short-term memory span advantages for chess experts over novices for meaningful chessboard configurations but not for random configurations has been replicated for programming (Curtis, Sheppard, Milliman, Borst & Love, 1979; McKeithen, Reitman, Rueter & Hirtle, 1981; Sheppard, Curtis, Milliman & Love, 1979; Schneiderman, 1977). For example, McKeithen *et al.* (1981) found that experts clustered keyword commands according to meaning (e.g. those functioning in loop statements), whereas novices clustered according to a variety of surface ordinary language associations (such as orthographic similarity and word length), intermediates falling between the two. Similarly, Adelson (1981) found that recall clusters for experts were functionally or "deeply" based; those of novices were based on "surface" features of programming code. This is a major developmental transformation, but we do not understand how it occurs. DiPersio, Isbister and Shneiderman (1980) extended this research by demonstrating that performance by college students on a program memorization/reconstruction task provides a useful predictor of programming test performances.

It is also a widely replicated finding that expert programmers debug programs in different ways than novices (Atwood & Ramsey, 1978; Gould, 1975; Gould & Drongowski, 1974; Youngs, 1974). Jeffries (1982) found that program debugging involves comprehension processes analogous to those for reading ordinary language prose. Experts read programs for flow of control (execution), rather than line-by-line (as text). But how do programmers shift from surface to deep readings of programs as they develop debugging skills?

In conclusion, we make one important observation. Expert programmers know much more than the facts of programming language semantics and syntax. However, the rich knowledge schemas, strategies, rules, and memory organizations that expert programmers reveal are directly taught only rarely. Many students appear to run aground in programming for lack of such understandings. This does not mean that they could not be taught, but for this to take place effectively will require considerable rethinking of the traditional computer science curriculum. These cognitive qualities appear instead to be a consequence of an active constructive process of capturing the lessons of program writing experience for later use.

LEVELS OF PROGRAMMING SKILL DEVELOPMENT

To date, observations of levels of programming skill development (cf. Howe, 1980) have been extremely general and more rationally than empirically derived. Accounts of novice – expert differences in programming ability among

adults coupled with observations of children learning to program provide a starting point for developing a taxonomy of levels of programming proficiency. This taxonomy can guide our research by providing a developmental framework within which to assess a student's programming expertise and make predictions for types of transfer beyond programming as a function of a student's level of expertise.

We believe that at least four distinct levels of programming ability can be identified that have distinct implications for what type of skills might transfer as the result of their achievement. These levels represent pure types and may not be characteristic of an individual, but they capture some complexities in what it means to develop programming skills. We view these levels only as guides toward more adequate characterizations of the development of programming abilities. Further differentiation will inevitably be required, in terms of the cognitive subtasks involved in the levels, and refined sublevels.

Level I Program user

A student typically learns to execute already written programs such as games, demonstrations, or computer-assisted instruction lessons before beginning instruction in how to program. What is learned here is important (i.e. what specific keys do, how to boot a disk, how to use screen menus), but does not reveal how the program works or that a program controls what happens on the screen. For many computer users this level is sufficient for effective computer use (e.g. for word processing, game playing, electronic mail). But to be more in control of the computer and able to tailor its capabilities to one's own goals, some type of programming is required.

From this level we would expect relatively little transfer beyond computer use, but some transfer on computer literacy issues. For example, given sufficiently wide exposure to different types of programs, a student would be expected to know what computers are capable of doing, what they cannot do, and fundamental aspects of how they function in their everyday lives. As users, then, children might learn when computers are appropriate tools to apply to a problem.

Level II Code generator

At this level the student knows the syntax and semantics of the more common commands in a language. He or she can read someone else's program and explain what each line accomplishes. The student can locate "bugs" preventing commands from being executed (e.g. syntax errors), can load and save program files to and from an external storage device, and can write simple programs of the type he or she has seen previously. When programming, the student does very little preplanning and does not bother to document his or her programs. There is no effort to optimize the coding, use error traps, or make the program usable by others. A program created at this level might just print the student's name repeatedly on the screen or draw the same shape again and again in different colors. The student operates at the level of the individual command and does not use subroutines or procedures created as part of other programs. This level of understanding of the programming process is sufficient for creating

short programs. But to create more widely useful and flexible programs, the student needs to progress to at least the next level.

At level II, more specific types of computer literacy related transfer would be expected. Students should develop better skills for dealing with more sophisticated software tools of the type which are rapidly permeating the business world. Computer-naive users of office information systems, even calculators, have many problems (e.g. Mann, 1975; Nickerson, 1981) and construct naive, error-ridden mental models of how they work (Mayer & Bayman, 1981; Newman & Sproull, 1979; Young, 1981). Knowledge characteristic of this level may be required to attenuate these problems. Sheil (1980, 1981a, b) provides compelling arguments that most systems require low level programming if the user wishes to take advantage of system options, a basic competency he has designated as "procedural literacy."

While potential computer literacy transfer from low level programming exposure seems a reasonable expectation, what types of cognitive transfer should ocur from this level of programming expertise is disputable. Our observations of children programming at this level suggest that some appreciation of the distinction between bugs and errors, degrees of correctness, and the value of decomposing program goals into manageable subparts may develop and transfer to other domains, but that a student's attention is typically so riveted to simply getting a program to work that any appreciation for more general cognitive strategies is lost.

Level III: Program generator

At this level the student has mastered the basic commands and is beginning to think in terms of higher level units. He or she knows sequences of commands accomplish program goals (e.g. locate and verify a keyboard input; sort a list of names or numbers; or read data into a program from a separate text file). The student can read a program and explain its purpose, what functions different parts of the program serve, and how the different parts are linked together. The student can locate bugs that cause the program to fail to function properly (e.g. a sort routine that fails to correctly place the last item in a list) or bugs that cause the program to crash as a result of unanticipated conditions or inputs (e.g. a division by zero error when the program is instructed to find the mean of a null list). The student can load, save, and merge files and can do simple calls to and from files from inside the main program. The student may be writing fairly lengthy programs for personal use, but the programs tend not to be user-friendly. While the student sees the need for documentation, he or she does not plan programs around the need for careful documentation or clear coding so that the program may be maintained by others. For this general level, one can expect to identify many sublevels of programming skill.

Within this level of expertise, students should develop some appreciation for the process of designing a successful program. Such understanding has potentially powerful implications for their work in other domains, particularly if such relationships are explicitly drawn by the teacher for students, or exemplified in other domains. However, it appears from our classroom observations and inter-

views with teachers that for students to spontaneously transfer computational concepts or language constructs used in one area of programming to other programming projects is a major accomplishment. Ideas about when to use variables, or the value of planning, as in designing program components so that they can be reused in the future, and following systematic conventions (such as beginning all graphics designs at their lower left corner) to make merging components into programs easier are all important accomplishments at this level that should not be taken for granted.

Level IV: Software developer

Finally, at this level the student is ready to write programs that are not only complex and take full advantage of the capabilities of the computer, but are intended to be used by others. The student now has a full understanding of all the features of a language and how the language interacts with the host computer (e.g. how memory is allocated or how graphics buffers may be protected from being overwritten). When given programs to read, the student can scan the code and simulate mentally what the program is doing, see how the goals are achieved and how the programs could be better written or adapted for other purposes. Programs are now written with sophisticated error traps and built-in tests to aid in the debugging process and to ensure the program is crash-proof. Beyond writing code accomplishing the program's objective, the student can optimize coding to increase speed and minimize the memory required to run a program. To decrease the time needed to write programs, he or she draws heavily on software libraries and programming utilities. Finally, he or she often crafts a design for the program before generating the code, documents the program fully, and writes the program in a structured, modular fashion so that others can easily read and modify it. Major issues in software engineering at high sublevels within this level of expertise are discussed by Thayer, Pyster and Wood (1981).

It is at this level of programming sophistication that we would expect to see most extensive evidence for cognitive transfer. The student can distance himself or herself sufficiently from the low level coding aspects of program generation to reflect on the phases and processes of problem solving involved. The issues of programming which the student is concerned with at this level — issues of elegance, optimalization, efficiency, verification, provability, and style — begin to transcend low level concerns with program execution, and may lead him or her to consider wider issues. The need at this level to be conscious of the range of intended users of programs forces the student to take the audience fully into account, a skill that has wide applicability in many other domains, such as writing.

Implicit in these distinctions between levels of programming skill and their linking to predictions about types of transfer is a theory of programming at odds with the "naive technoromanticism" prevalent in educational computing. While it is conceivable that even low levels of programming skill are sufficient to produce measurable cognitive transfer to non-programming domains, we contend that on the limited evidence available, this would be unlikely. Students who can barely decode or comprehend text are not expected to be proficient

writers. Similarly, we doubt that students with a low level understanding of programming and the skills that programming entails will write functional programs or gain insights into other domains on the basis of their limited programming skill.

COGNITIVE CONSTRAINTS ON LEARNING TO PROGRAM

Beyond asking what general cognitive characteristics may be prerequisite to or substantively influence a child's learning to program, some ask what "developmental level" children must be "at" in order to learn from programming experiences? The concept of "developmental level" at the abstract theoretical planes of preoperational, concrete operational, and formal operational intellectual functioning has proved to be useful for instructional psychology in understanding children's ability to benefit from certain types of learning experiences (e.g. Inhelder, Sinclair & Bovet, 1974). But the very generality of these stage descriptions is not suitably applied to the development of specific domains of knowledge such as programming skills.

We have two reasons for not pursuing the development of programming skills in terms of Piagetian "developmental levels". First, there is strong evidence that the development and display of the logical abilities defined by Piaget is importantly linked to content domain (Feldman, 1980; Gardner, 1983; Piaget, 1972), to the eliciting context (Laboratory of Comparative Human Cognition, 1983), and to the particular experiences of individuals (Price-Williams, Gordon & Ramirez, 1969). Since it is not apparent why and how different materials affect the "developmental level" of children's performances within Piagetian experimental tasks, it is not feasible to predict relationships between learning to program and performances on the Piagetian tasks. Our second objection is that learning to program has neither been subjected to developmental analysis nor characterized in terms of its component skills that may develop, although such analyses are necessary for articulating measures that indicate the availability and developmental status of these skills for particular learners.

While no research has been directly aimed at defining the cognitive prerequisites for learning programming, at least six factors are frequently mentioned: mathematical ability, memory capacity, analogical reasoning skills, conditional reasoning skills, procedural thinking skills, and temporal reasoning skills. These cognitive abilities, each of which have complex and well-researched developmental histories, are presumed to impact on learning to program, and could be promising directions for research.

Mathematical ability

Beyond "general intelligence", programming skill is said to be linked to general mathematical ability. Computers were first developed to help solve difficult mathematical problems. Although many computer uses today are non-mathematical (e.g. data base management, word processing), the notion persists that to program one must be mathematically sophisticated. Media accounts of children using computers in schools have perpetuated the belief that programming is the province of math whizzes. Although we doubt that math and programming abilities are related once general intelligence is factored out,

mathematical ability cannot be ruled out as a prerequisite to the mastery of certain levels of programming skills.

Processing capacity

Programming is often a memory-intensive enterprise requiring great concentration and the ability to juggle values of a number of parameters at a time. Individual differences in processing capacity are thus a likely candidate for influencing who becomes a good programmer. Forward and backward span tasks, and more recently developed transformational span measures (cf. Case & Kurland 1980; Case, Kurland & Goldberg, 1982) assess how much information one can coordinate at a given moment, and appear to index processes basic to learning. Performances on such tasks have reliably correlated with general intelligence, Piagetian developmental level, and ability to learn and use problem solving strategies (e.g. Hunt, 1978).

Analogical reasoning

A student may have background knowledge and capacities relevant to programming and yet neither connect them to the programming domain, nor transfer knowledge acquired in programming to other domains. This "access" of knowledge is absolutely fundamental to learning and problem solving throughout life (e.g. Brown, 1982). Transfers of knowledge and strategies, both "into" and "out of" learning to program may depend on analogical thinking skills. Tasks designed to measure abilities for engaging in analogical thinking (e.g. Gick & Holyoak, 1980; Sternberg & Rifkin, 1979) may predict level of programming development and transfer outcomes. Mayer (1975, 1981) argues that students learn programming by comparing the flow of control intrinsic to computational devices to that of physico-mechanical models that they already possess. Also, duBoulay and O'Shea (1976) and du Boulay *et al.* (1981) have successfully used extensive analogical modelling to explain computer functioning to novice 12-year-old programming students.

Conditional reasoning

Working with conditional statements is a major part of programming, since they guide the operation of loops, tests, input checking, and other programming functions. It is thus reasonable to predict that a student who has sufficient understanding of conditional logic, the various "if . . . then" control structures and the predicate logical connectives of negation, conjunction, and disjunction, will be a more successful programmer than a student who has trouble monitoring the flow of control through conditional statements.

Procedural thinking

Several kinds of quasi-procedural* everyday thought may influence how easily

* What is "quasi-procedural" rather than "procedural" about giving and following task instructions, directions, and recipes, is that unlike procedural instructions in a computer program, there is often *ambiguity* in the everyday examples, such that the instructions, directions, and recipes are not always unequivocal in meaning. They are also not constrained by strict *sequentiality*. One many often choose to bypass steps in a recipe or set of instructions, or reorder the steps. Neither option is available in the strict procedurality of programmed instructions. Yet similarities between the everyday cases and programming instructions are compelling enough to make their designation as "quasi-procedural" understandable.

a learner masters the "flow of control" procedural metaphor central to understanding programming, including giving and following complex instructions (as in building a model), writing or following recipes, and concocting or carrying out directions for travel. Presumably, learners more familiar with these linear procedures, analogous to the flow of control for computer operations expressed as instructions in a computer program, will more readily come to grips with the "procedural thinking" touted as a central facet of programming expertise (Papert, 1980; Sheil, 1980). However, the development of procedural thinking has been little studied to date.

Temporal reasoning

The activity of temporal reasoning is related to procedural thinking, but with a distinct emphasis. Creating and comprehending programs requires an understanding of the temporal logic of sequential instructions: "it is the intellectual heart of learning how to program" (Galanter, 1983, p. 150). In teaching programming, Galanter says: "The central theoretical concept that guided this effort was that classical forms of spacial – geometric – pictorial thinking must be augmented, and occasionally replaced, by temporal – imaginative – memorial logic. The child must learn to substitute an inner temporal eye for the outer spacial eye" (p. 163). Going somewhere in the program *next*, running one subroutine or procedure *before* another, ensuring one counter does not exceed a certain value *until* another operation is performed — these fundamental operations all require temporal understanding. Yet understanding temporal terms is a major developmental achievement, a challenge for children younger than 7 to 8 years (e.g. Friedman, 1982; Piaget, 1969). Futurity also presents complex conceptual problems for the planning activities involved in programming, such as imagining outcomes of the possible worlds generated by program design options (Atwood, Jeffries & Polson, 1980), or the "symbolic executions" while writing programming code (Brooks, 1977).

In sum, the cognitive constraints on developing programming skills are currently unknown. Although a developmental cognitive science perspective predicts that a student's attainable level of programming skill may be constrained by cognitive abilities required in programming, no studies relate level of programming skill to the abilities that we have described. Children may have conceptual and representational difficulties in constructing dynamic mental models of ongoing events when the computer is executing program lines that constrain their level of programming skill. Also, systematic but "naive" mental models or intuitive epistemologies of computer procedural functioning may initially mislead children's understanding of programming, as with adult novices. Since learning to program is difficult for many students, there is a serious need for research findings that will guide decisions about tailoring programming instruction according to a student's relevant knowledge prior to learning to program.

EVIDENCE FOR COGNITIVE EFFECTS OF PROGRAMMING

We now return to evidence for the claims for broad cognitive impacts of pro-

gramming experience, with greater awareness of the complexities of learning to program and issues of transfer. In sum, there is little evidence for these claims.

Dramatic accounts have been offered of how some school-aged children's thinking about their own abilities to solve problems is transformed through learning to program (e.g. Papert *et al.*, 1979; Watt, 1982; Weir & Watt, 1981; Weir, 1981). Important social interactional changes have been demonstrated in classrooms where children are learning Logo programming (Hawkins, Sheingold, Gearhart & Berger, 1983), and for some children programming is an important and deeply personal intellectual activity. Similarly, many teacher reports focus on social and motivational rather than cognitive aspects of this experience (Sheingold, Kane, Endreweit & Billings, 1981; Watt, 1982). It is not yet clear what the cognitive benefits of programming for such children may be in terms of the transfer claims reviewed earlier.

On the cognitive side, Ross and Howe (1981) have reviewed ten years of relevant research to evaluate Feurzeig *et al.*'s (1969) four general claims on the cognitive impacts of programming. The relevant research has been with Logo, and in nonrepresentative private schools. Below we summarize Ross and Howe's review, and integrate summaries of other studies relevant to these claims. In terms of our account of levels of programming skill and expected transfer outcomes from them, we must caution that studies so far, including our own, have an important limitation. They have all looked at what we have designated as high level or cognitive transfer outcomes, expected to emerge only at the higher levels in our account of programming skill, whereas the levels of programming attained by the students in these studies were low because they only did six weeks to a year or so of programming. In other words, there has been a mismatch of "treatment" and transfer assessments because of a failure to appreciate the different kinds of transfer to investigate and their likely linkage to different levels of programming skill. For example, there are no studies that have assessed the low-level transfer or application of programming concepts such as "variable" in different types of programming within a language (e.g. graphics versus list processing in Logo), or from one programming language to another, or of computer literacy outcomes.

First, there are no substantial studies to support the claim that programming promotes mathematical rigor. In a widely cited study by Howe, O'Shea and Plane (1979), researchers who were highly trained programmers spent two years teaching Logo programming to eleven 11-year-old boys of average or below average math ability. The first year they studied Logo, the second math with Logo, each boy working for one hour per week in a programming classroom. After two years, when Logo students were compared to non-programmers (who on pretest had significantly better scores on the Basic Mathematics Test, but equivalent scores on the Math Attainment Test), they had improved in Basic Math enough to eliminate the original performance gap with the control group, but fell significantly behind on the Math Attainment Test. Such global math score differences do not support the "rigor" claim. The oftcited finding is that the Logo group learned to argue sensibly about mathematical issues and explain mathematical difficulties clearly, but the

finding is based only on differences in ratings of Logo and control students in teacher questionnaires (Howe *et al.*, 1979). The reliability of such ratings is questionable, since the math teachers should have been blind to which students learned Logo.

Secondly, there are no reports demonstrating that programming aids children's mathematical exploration. Reports by Dwyer (1975) for children learning BASIC, and Howe *et al.* (1979), Lawler (1980), and Papert *et al.* (1979) for those using Logo, do document children's goal-directed exploration of mathematical concepts such as "variable" on computers. Though encouraging, since math exploration and "mathland" play are likely to support math learning, studies have not shown any effects of "math exploration" during programming outside the programming environment.

Third, although Feurzeig *et al.* (1969) suggest that the twelve 7- to 9-year-old children to whom they taught Logo came to "acquire a meaningful understanding of concepts like variable, function and general procedure", they provide no evidence for the claim that programming helped the children gain insight into these mathematical concepts.

Finally, we ask whether programming has been shown to provide a context and language that promotes problem solving beyond programming. Papert *et al.* (1979) conducted a Logo project with sixth graders for six weeks, and reported anecdotes that children engage in extensive problem solving and planning activities in learning programming. Whether such activities had cognitive effects beyond programming was not studied. However, Statz (1973) carried out a study to assess this claim. Logo programming was taught to sixteen 9- to 11-year-old children for a year. Statz chose four problem solving tasks with intuitive, ill-specified connections to programming activities as transfer outcome measures. The experimental group did better on two of these tasks (word puzzle and a permutation task), but no better on the Tower of Hanoi task or a horserace problem that Statz had designed. She interprets these findings as mixed support for the claim that learning Logo programming promotes the development of more general problem solving skills.

Soloway, Lochhead and Clement (1982), in reaction to the finding (Clement, Lochhead & Monk, 1979) that many college science students have difficulty translating simple algebra word problems into equations, found that more students solve such problems correctly when they are expressed as computer programs rather than as algebraic equations. They attribute this advantage to the procedural semantics of equations in programs that many students lack in the algebraic task. This effect is much more restricted than the increments in general problem solving skill predicted by the cognitive transfer claims.

A very important idea is that not only computer programs, but one's own mental activities can lead to "buggy" performances and misunderstandings. Tools for diagnosing different types of "bugs" in such procedural skills as place-value arithmetic (Brown & Burton, 1978; Brown & VanLehn, 1980; VanLehn, 1981) have resulted from extensive programming efforts to build "bug diagnostic systems" (Burton, 1981). One may argue that the widespread recognition that systematic "bugs" may beset performances in other procedural

skills, such as high school algebra (Carry, Lewis & Bernard, 1979; Matz, 1981) reflects a kind of transfer beyond programming. No evidence indicates that programming students demonstrate such transfer.

Planning in advance of problem solving, and evaluating and checking progress in terms of goals, are important aspects of a reflective attitude to one's own mental activities (Pea, 1982). We have seen that the development of planning abilities is one major predicted cognitive benefit of learning to program. We therefore developed a transfer task for assessing children's planning (Pea & Hawkins, 1984). We reasoned that a microgenetic method (Flavell & Draguns, 1957) allowing children to develop multiple plans was comparable to the rounds of revisions carried out during programming, and would allow for a detailed study of planning processes. Children planned aloud while formulating, over several attempts, their shortest-distance plan for doing a set of familiar classroom chores, using a pointer to indicate their routes. We gave the task twice, early and late in the school year, to eight children in each of two Logo classrooms (8- and 9-year-olds; 11- and 12-year-olds), and to a control group of the same number of same-age children in the same school. There were six microcomputers in each classroom, allowing substantial involvement with programming.

As in related work on adults' planning processes by Goldin and Hayes-Roth (1980; also Hayes-Roth & Hayes-Roth, 1979; Hayes-Roth, 1980), our product analyses centered on "plan goodness" in terms of metrics of route efficiency, and our process analyses centered on the types and sequencing of planning decisions made (e.g. higher level executive and metaplanning decisions such as what strategic approach to take to the problem, versus lower level decisions of what route to take between two chore acts). Results indicated that the Logo programming experiences had *no* significant effects on planning performances, on any of the plan efficiency or planning process measures (Pea & Kurland, 1983a). Replications of this work are currently under way with children in other schools.

CONCLUSIONS

As our society comes to grips with the information revolution, the ability to deal effectively with computers becomes an increasingly important skill. How well our children learn to use computers today will have great consequences for the society of tomorrow. The competence to appropriately apply higher cognitive skills such as planning and problem solving heuristics in mental activities both with and without computers is a critical aim for education. As one contribution to these issues, at the beginning we argued for and then throughout documented the need for a new approach to the pervasive questions about the cognitive effects of computer programming. This approach, which we characterize as developmental cognitive science, is one that does not merely adopt the common perspective that computer programmers are all like adults, but is instead geared to the learning experiences and developmental transformations of the child or novice, and in its research would be attentive to the playing out

of those processes of learning and development in the instructional and programming environments in which the novice gains expertise.

So can children become effective programmers and does "learning to program" positively influence children's abilities to plan effectively, to think procedurally, or to view their flawed problem solutions as "fixable" rather than "wrong"? We have shown that answers to these questions depend on what "learning to program" is taken to mean. We reviewed cognitive science studies revealing that programming involves a complex set of skills, and argued that the development of different levels of programming skill will be highly sensitive to contexts for learning, including processes of instruction, programming environment, and the background knowledge the student brings to the task. We found few studies that could inform this new understanding, although many promising research questions were defined from this perspective.

We dismissed two prevailing myths about learning to program. The myth embodied in most programming instruction that learning to program is "learning facts" of programming language semantics and syntax is untenable, since it leads to major conceptual misunderstandings even among adult programmers, and since what is taught belies what cognitive studies show good programmers do and know. These studies have direct implications for new content and methods for programming instruction that are under development in several quarters. Studies of learning to program and of transfer outcomes are not yet available for cases where instruction has such nontraditional emphases, e.g. on task analysis and problem solving methods that take advantage of what we know expert programmers do. We also delivered arguments against the second myth, of spontaneous transfer of higher cognitive skills from learning to program. Resistance in learning to spontaneous transfer, and the predicted linkages of kinds of transfer beyond programming to the learner's level of programming skill were major points of these critical reviews.

So when thinking about children learning to program, what levels of skills can be expected? Reports of children learning to program (Howe, 1981; Levin & Kareev, 1980; Papert *et al.*, 1979; Pea, 1983), including the learning disabled, the cerebral palsied and the autistic (Watt & Weir, 1981; Weir, 1981), suggest that most children can learn to write correct lines of code (level II in our account). This is no small achievement since writing grammatically correct lines of code is all many college students of programming achieve in their first programming courses (Bonar & Soloway, 1982). This level of programming skill may depend on the same abilities necessary for learning a first language.

However, for programming skills that are functional for solving problems, "grammatical" programming alone is inadequate; the student must know how to organize code and "plan schemas" to accomplish specific goals. Development to these higher levels, where one becomes facile with the pragmatics of programming, may require strategic and planful approaches to problem solving that are traditionally considered "metacognitive," and more characteristic of adolescents (Brown *et al.*, 1983) than primary school children. Further, the experience of the child in an elementary or junior high school program who spends up to 30 to 50 hours per year programming is minuscule when compared

to the 5000 hours which Brooks (1980) estimates a programmer with only three years of experience has spent on programming. Since it appears unreasonable to expect children to become advanced programmers in the few years available to them in most school programming courses, our educational goals should be more realistic and achievable. We do not currently know what levels of programming expertise to expect, but in our experience children who are programming experts are not common. There are thus large gaps between what is meant by learning to program in the computer science literature, and what "learning programming" means to educators interested in exposing this domain to children. These discrepancies should temper expectations for the spontaneous effects of children's limited programming experiences in school on their ways of thinking, at least for how programming is taught (or not taught) today. Whether research on learning to program with richer learning experiences and instruction will lead to powerful outcomes of programming remains to be seen. In place of a naive technoromanticism, we have predicted that the level of programming abilities a student has mastered will be a predictor of the kinds of concepts and skills that the student will transfer beyond programming. Although findings to date of transfer from learning to program have not been encouraging, these studies suffer in not linking level of programming skill to specific outcomes expected, and the critical studies of "low level" transfer expected from level I and II programming skills remain to be carried out. Even more importantly, with thinking skills as educational goals, we may be best off providing direct guidance that teaches or models transfer as a general aspect of highly developed thinking processes (Segal, Chipman & Glaser, 1984; Smith & Bruce, 1981). For these purposes programming may provide one excellent domain for examples (Nickerson, 1982; Papert, 1980).

Throughout, we have emphasized how developmental research in this area is very much needed. We need empirical studies to refine our characterizations of levels of programming proficiency, extensive evaluations of the extent of transfer within and beyond programming in terms of different programming and instructional environments, and studies to help untangle the complex equation involving cognitive constraints, programming experience, and programming outcomes. We believe all of these questions could be addressed by careful longitudinal studies of the learning and development process by which individual students become proficient (or not-so proficient) programmers, and of the cognitive consequences of different levels of programming skill. Such studies would provide far more relevant information for guiding the processes of education than standard correlational studies. A focus on process and the types of interactions that students with different levels of entering skills have with programming and instructional environments is critical for understanding how developments in programming skill are related to other knowledge. We are optimistic that others will join in work on these questions, for progress must be made toward meeting the educational needs of a new society increasingly empowered by information technologies.

REFERENCES

Adelson B. Problem solving and the development of abstract categories in programming languages. *Memory and Cognition* **9**, 422 - 433 (1983).

Anderson J. R., Greeno J. G., Kline P. J. & Neves D. M. Acquisition of problem solving skill. In *Cognitive Skills and their Acquisition* (ed. Anderson J. R.). Erlbaum, Hillsdale, NJ (1981).

Anderson R. E. National computer literacy, 1980. In *Computer Literacy: Issues and Directions for 1985* (eds. Seidel R. J., Anderson R. E. & Hunter B.). Academic Press, New York (1982).

Atwood M. E., Jeffries R. & Polson P. G. *Studies in plan construction. I: Analysis of an extended protocol.* (Tech. Rep. No. SAI-80-028-DEN). Science Applications, Inc., Englewood, CO (1980).

Atwood M. E. & Ramsey H. R. *Cognitive structures in the comprehension and memory of computer programs: An investigation of computer debugging* (Tech. Rep. No. TR-78A21). U.S. Army Research Institute for the Behavioral and Social Sciences, Alexandria, VA (1978).

Barstow D. R. *Knowledge-Based Program Construction*. North-Holland, Amsterdam (1979).

Bereiter C. & Scardamalia M. From conversation to composition: Instruction in a developmental process. In *Advances in Instructional Psychology* (ed. Glaser R.), Vol. 2. Erlbaum, Hillsdale, NJ (1982).

Black S. D., Levin J. A., Mehan H. & Quinn C. N. Real and non-real time interaction: Unraveling multiple threads of discourse. *Discourse Processes*, 1983, in press.

Bonar J. Natural problem solving strategies and programming language constructs. *Proceedings of the Fourth Annual Conference of the Cognitive Science Society*, Ann Arbor, Michigan, 4 - 6 August (1982).

Bonar J. & Soloway E. Uncovering principles of novice programming. Yale University Department of Computer Science, Research Report #240, November 1982. (To appear in the Tenth SIGPLAN-SIGACT Symposium on the Principles of Programming Languages, Austin, Texas, January 1983.)

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

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

Brown A. L. Learning and development: The problems of compatibility, access, and induction. *Human Development* **25**, 89 - 115 (1982).

Brown A. L. Metacognition, executive control, self-regulation and other even more mysterious mechanisms. In *Metacognition, Motivation and Learning* (eds. Klume R. H. & Weinert F. E.). Luhlhammer, West Germany (1983a, in press).

Brown A. L. Learning to learn how to read. In *Reader Meets Author, Bridging the Gap: A Psycholinguistic and Social Linguistic Perspective* (eds. Langer J. & Smith-Burke T.). Dell, Newark, NJ (1983b).

Brown A. L., Bransford J. D., Ferrara R. A. & Campione J. C. Learning, remembering, and understanding. In *Mussen Handbook of Child Psychology* (eds. Flavell J. H. & Markman E. M.), Vol. 3. Wiley, New York (1983).

Brown A. L. & Smiley S. S. The development of strategies for studying texts. *Child Development* **49**, 1076 - 1088 (1978).

Brown J. S. & Burton R. B. Diagnostic models for procedural bugs in basic mathematical skills. *Cognitive Science* **2**, 155 - 192 (1978).

Brown J. S. & VanLehn K. Repair theory: A generative theory of bugs in procedural skills. *Cognitive Science* **4**, 379 - 426 (1980).

Bruner J. S. On cognitive growth. In *Studies in Cognitive Growth* (eds. Bruner J. S., Olver R. R. & Greenfield P. M.). Wiley, New York (1966).

Burton R. B. Debuggy: Diagnosis of errors in basic mathematics skills. In *Intelligent Tutoring Systems* (eds. Sleeman D. H. & Brown J. S.). Academic Press, London (1981).

Carry L. R., Lewis C. & Bernard J. E. *Psychology of equation solving: An information processing study.* Department of Curriculum and Instruction, University of Texas at Austin, Austin, TX (1979).

Case R. & Kurland D. M. A new measure for determining children's subjective organization of

speech. *Journal of Experimental Child Psychology* **30**, 206 - 222 (1980).
Case R., Kurland D. M. & Goldberg J. Operational efficiency and the growth of short-term memory span. *Journal of Experimental Child Psychology* **33**, 386 - 404 (1982).
Chase W. G. & Simon H. A. Perception in chess. *Cognitive Psychology* **4**, 55 - 81 (1973).
Chi M. T. H., Feltovich P. J. & Glaser R. Categorization and representation of physics problems by experts and novices. *Cognitive Science* **5**, 121 - 152 (1981).
Clement J., Lochhead J. & Monk G. *Translation difficulties in learning mathematics* (Tech. Rep.). Cognitive Development Project, Department of Physics and Astronomy, University of Massachusetts, Amherst (1979).
Cole M. & Griffin P. Cultural amplifiers reconsidered. In *The Social Foundations of Language and Thought: Essays in Honor of Jerome S. Bruner* (ed. Olson D. R.). W. W. Norton, New York (1980).
Collins A. & Gentner D. *Constructing runnable mental models*. Proceedings of the Fourth Annual Conference of the Cognitive Science Society, Ann Arbor, MI, August (1982).
Cromer R. F. The development of language and cognition: the cognition hypothesis. In *New Perspectives in Child Development* (ed. Foss B.), pp. 184 - 252. Penguin, London (1974).
Crystal D. *A First Dictionary of Linguistics and Phonetics*. Cambridge University Press, Cambridge (1980).
Curtis B., Sheppard S. B., Milliman P., Borst M. A. & Love T. Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics. *IEEE Transactions on Software Engineering* **SE-5**, 96 - 104 (1979).
DeKleer J. & Brown J. S. Mental models of physical mechanisms and their acquisition. In *Cognitive Skills and their Acquisition* (ed. Anderson J. R.). Erlbaum, Hillsdale, NJ (1981).
Dewey J. *The School and Society*. University of Chicago Press, Chicago (1900).
DiPersio T., Isbister D. & Shneiderman B. An experiment using memorization/reconstruction as a measure of programmer ability. *International Journal of Man-Machine Studies* **13**, 339 - 354 (1980).
DiSessa A. A. Unlearning Aristotelian physics: A study of knowledge-based learning. *Cognitive Science* **6**, 37 - 75 (1982).
DuBoulay J. B. H. & O'Shea T. *How to work the Logo machine: A primer for ELOGO* (D.A.I. Occasional Paper No. 4). Department of Artificial Intelligence, University of Edinburgh, Edinburgh (1976).
DuBoulay J. B. H., O'Shea T. & Monk J. The black box inside the glass box: Presenting computing concepts to novices. *International Journal of Man-Machine Studies* **14**, 237 - 249 (1981).
Dwyer T. A. Soloworks: Computer based laboratories for high school mathematics. *Science and Mathematics* **22**, 93 - 99 (1975).
Ehrlich K. & Soloway E. An empirical investigation of the tacit plan knowledge in programming. In *Human Factors in Computer Systems* (eds. Thomas J. & Schneider M.). Ablex, Norwood, NJ (1983).
Eisenstadt M., Laubsch J. H. & Kahney J. H. *Creating pleasant programming environments for cognitive science students*. Paper presented at the meeting of the Cognitive Science Society, Berkeley, CA, August (1981).
Feldman D. H. *Beyond Universals in Cognitive Development*. Ablex, Norwood, NJ (1980).
Feurzeig W., Horwitz P. & Nickerson R. S. *Microcomputers in education* (Report No. 4798). Prepared for: Department of Health, Education, and Welfare; National Institute of Education; and Ministry for the Development of Human Intelligence, Republic of Venezuela. Bolt Beranek & Newman, Cambridge, MA, October (1981).
Feurzeig W., Papert S., Bloom M., Grant R., & Solomon C. *Programming languages as a conceptual framework for teaching mathematics* (Report No. 1899). Bolt Beranek & Newman, Cambridge, MA (1969).
Flavell J., & Draguns J. A microgenetic approach to perception and thought. *Psychological Bulletin* **54**, 197 - 217 (1957).
Floyd R. W. The paradigms of programming. *Communications of the ACM* **22**, 455 - 460 (1979).
Friedman W. J. (Ed.) *The Developmental Psychology of Time*. Academic Press, New York (1982).

Galanter E. *Kids and Computers: The Parents' Microcomputer Handbook*. Putnam, New York (1983).
Gardner H. *Frames of Mind. The Theory of Multiple Intelligences*. Basic Books, New York (1983).
Gentner D. & Stevens A. L. (Eds.) *Mental Models*. Erlbaum, Hillsdale, NJ (1983).
Gick M. L. & Holyoak K. J. Analogical problem solving. *Cognitive Psychology* **12**, 306 - 355 (1980).
Gick M. L. & Holyoak K. J. Schema induction and analogical transfer. *Cognitive Psychology* **15**, 1 - 39 (1982).
Goldin S. E. & Hayes-Roth B. *Individual differences in planning processes*. N-1488-ONR: A Rand Note, Rand Corp., Santa Monica, CA, June (1980).
Goldstein I. & Papert S. Artificial intelligence, language, and the study of knowledge. *Cognitive Science* **1**, 84 - 123 (1977).
Goody J. *The Domestication of the Savage Mind*. Cambridge University Press, New York (1977).
Gould J. D. Some psychological evidence on how people debug computer programs. *International Journal of Man-Machine Studies* **7**, 151 - 182 (1977).
Gould J. D. & Drongowski P. An exploratory investigation of computer program debugging. *Human Factors* **16**, 258 - 277 (1974).
Green C. C. & Barstow D. On program synthesis knowledge. *Artificial Intelligence* **10**, 241 - 279 (1978).
Greeno J., Glaser R. & Newell A. *Research on cognition and behavior relevant to education in mathematics, science, and technology*. Report submitted to the National Science Board Commission on Pre-college Education in Mathematics, Science, and Technology by the Federation of Behavioral, Psychological and Cognitive Sciences, March (1983).
Hawkins J., Sheingold K., Gearhart M. & Berger C. The impact of computer activity on the social experience of classrooms. *Journal of Applied Developmental Psychology* **2**, 361 - 373 (1983).
Hayes J. R. & Simon H. A. Psychological differences among problem isomorphs. In *Cognitive Theory* (eds. Castellan N. J., Jr., Pisoni D. B. & Potts G. R.), Vol. 2. Erlbaum, Hillsdale, NJ (1977).
Hayes-Roth B. *Estimation of time requirements during planning: The interactions between motivation and cognition*. N-1581-ONR: A Rand Note, Rand Corp., Santa Monica, CA, November (1980).
Hayes-Roth B. & Hayes-Roth F. A cognitive model of planning. *Cognitive Science* **3**, 275 - 310 (1979).
Howe J. A. M. Developmental stages in learning to program. In *Cognition and Memory: Interdisciplinary Research of Human Memory Activities* (eds. Klix F. & Hoffman J.). North-Holland, Amsterdam (1980).
Howe J. A. M. *Learning mathematics through Logo programming* (Research Paper No. 153). Department of Artificial Intelligence, University of Edinburgh, Edinburgh (1981).
Howe J. A. M., O'Shea T. & Plane F. Teaching mathematics through Logo programming: An evaluation study. In *Computer-Assisted Learning—Scope, Progress and Limits* (eds. Lewis R. & Tagg E. D.). North-Holland, Amsterdam (1979).
Hunt E. Mechanics of verbal ability. *Psychological Review* **85**, 109 - 130 (1978).
Inhelder B., Sinclair H. & Bovet M. *Learning and the Development of Cognition*. Harvard University Press, Cambridge, MA (1974).
Jeffries R. *A comparison of the debugging behavior of expert and novice programmers*. Paper presented at the Annual Meeting of the American Educational Research Association, New York City, March (1982).
Johnson W. L., Draper S. & Solowav E. An effective bug classification scheme must take the programmer into account. *Proceedings of the Workshop on High-Level Debugging*, Palo Alto, CA (1983).
Kahney H. & Eisenstadt M. Programmers' mental models of their programming tasks: The interaction of real-world knowledge and programming knowledge. Proceedings of the Fourth Annual Conference of the Cognitive Science Society, Ann Arbor, MI, 4 - 6 August (1982).
Kurland D. M. & Pea R. D. Children's mental models of recursive Logo programs. Proceedings of the Fifth Annual Meeting of the Cognitive Science Society, Rochester, NY, May 1983 (Also Tech. Rep. 10, Center for Children & Technology, Bank Street College, New York February 1983.)

Laboratory of Comparative Human Cognition. Culture and cognitive development. In *Mussen Handbook of Child Psychology: History, Theories and Methods* (ed. Kessen W.). Vol. 1. John Wiley, New York (1983).

Laboratory of Comparative Human Cognition. Microcomputer communication networks for education. *The Quarterly Newsletter of the Laboratory of Comparative Human Cognition* **4**, April (1982).

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

Lawler R. W. *Extending a powerful idea* (Logo Memo No. 58). M.I.T. Artificial Intelligence Laboratory, Cambridge, MA, July (1980).

Levin J. A. & Kareev Y. *Personal computers and education: The challenge to schools* (CHIP Report No. 98). Center for Human Information Processing, La Jolla, CA (1980).

Lewis C. Skill in algebra. In *Cognitive Skills and their Acquisition* (ed. Anderson J. R.). Erlbaum, Hillsdale, NJ (1981).

Mann W. C. Why things are so bad for the computer-naive user. Information Sciences Institute, March (1975).

Matz M. Towards a process model of high school algebra errors. In *Intelligent Tutoring Systems* (eds. Sleeman D. H. & Brown J. S.). Academic Press, London (1981).

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

Mayer R. E. A psychology of learning BASIC. *Communications of the ACM* **22**, 589 – 593 (1979).

Mayer R. E. The psychology of learning computer programming by novices. *Computing Surveys* **13**, 121 – 141 (1981).

Mayer R. E. Some conditions of meaningful learning for computer programming: Advance organizers and subject control of frame order. *Journal of Educational Psychology* **68**, 143 – 150 (1976).

Mayer R. E. & Bayman P. Psychology of calculator languages: A framework for describing differences in users' knowledge. *Communications of the ACM* **24**, 511 – 520 (1981).

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

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

Minsky M. Form and content in computer science. *Communications of the ACM* **17**, 197 – 215 (1970).

National Assessment of Educational Progress. *Procedural Handbook: 1977 – 78 Mathematics Assessment*. Education Commission of the States, Denver, CO (1980).

Newell A. One final word. In *Problem Solving and Education* (eds. Tuma D. T. & Reif F.). Halsted Press, New York (1980).

Newell A., & Simon H. *Human Problem Solving*. Prentice-Hall, Englewood Cliffs, NJ (1972).

Newman W. M. & Sproull R. F. *Principles of Interactive Computer Graphics*, 2nd edn. McGraw-Hill, New York (1979).

Nickerson R. S. Computer programming as a vehicle for teaching thinking skills. *Thinking, The Journal of Philosophy for Children* **4**, 42 – 48 (1982).

Nickerson R. S. Why interactive computer systems are sometimes not used by people who might benefit from them. *International Journal of Man-Machine Studies* **14**, 469 – 481 (1981).

Norman D. A. (ed.) *Perspectives on Cognitive Science*. Erlbaum, Hillsdale, NJ (1981).

Olson D. R. Culture, technology and intellect. In *The Nature of Intelligence* (ed. Resnick L. B.). Erlbaum, Hillsdale, NJ (1976).

Ong W. J. *Orality and Literacy: The Technologizing of the Word*. Methuen, New York (1982).

Papert S. *Mindstorms*. Basic Books, New York (1980).

Papert S. Teaching children thinking. *Programmed Learning and Educational Technology* **9**, 245 – 255 (1972a).

Papert S. Teaching children to be mathematicians versus teaching about mathematics. *International Journal for Mathematical Education, Science and Technology* **3**, 249 – 262 (1972b).

Papert S., Watt D., diSessa A. & Weir S. *An assessment and documentation of a children's computer laboratory*. Final Report of the Brookline Logo Project, Brookline, MA (1979).

Pea R. D. *Programming and problem solving: Children's experience with Logo*. Paper presented at Annual Meetings of the American Educational Research Association, Montreal, Canada, April 1983. (Also Tech. Rep. 12. Center for Children & Technology, Bank Street College, New York, April 1983.)

Pea R. D. What is planning development the development of? In *New Directions in Child Development. Children's Planning Strategies* (eds. Forbes D. & Greenberg M.), Vol. 18. Jossey-Bass, San Francisco (1982).

Pea R. D. & Hawkins J. A microgenetic study of planning processes in a chore-scheduling task. In *Blueprints for Thinking: The Development of Social and Cognitive Planning Skills* (eds. Friedman S. L., Scholnick E. K. & Cocking R. R.). Cambridge University Press, Cambridge (1984, in press).

Pea R. D. & Kurland D. M. *Logo programming and the development of planning skills* (Tech. Rep. No. 16). Center for Children & Technology, Bank Street College, New York, April (1983a).

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

Piaget J. *The Child's Conception of Time* (ed. Pomerans A. J.). Ballantine, New York (1969).

Piaget J. Intellectual evolution from adolescence to adulthood. *Human Development* **15**, 1 - 12 (1972).

Piaget J. & Inhelder B. *The Psychology of the Child*. Basic Books, New York (1969).

Polya G. *How to Solve it*. Doubleday-Anchor, New York (1957).

Price-Williams D., Gordon W. & Ramirez M. Skill and conservation: A study of pottery-making children. *Developmental Psychology* **1**, 769 (1969).

Resnick L. B. *A new conception of mathematics and science learning*. Presentation at the National Convocation on Precollege Education in Mathematics and Science, National Academy of Sciences and National Academy of Engineering, 12 - 13 May (1982).

Rich C. & Shrobe H. Initial report on a Lisp programmer's apprentice. *IEEE Transactions on Software Engineering* **SE-4**, 456 - 467 (1978).

Ross P. & Howe J. Teaching mathematics through programming: Ten years on. In *Computers in Education* (eds. Lewis R. & Tagg D.). North-Holland, Amsterdam (1981).

Scardamalia M. & Bereiter C. The development of evaluative, diagnostic and remedial capabilities in children's composing. In *The Psychology of the Written Language* (ed. Martlew M.). John Wiley, London (1983).

Schank R. *Dynamic Memory*. Cambridge University Press, Cambridge (1982).

Schank R. & Abelson R. P. *Scripts, Plans, Goals and Understanding*. Erlbaum, Hillsdale, NJ (1977).

Segal S., Chipman J. & Glaser R. (eds.) *Thinking and Learning Skills: Current Research and Open Questions*. Erlbaum, Hillsdale, NJ (1984, in press).

Sheil B. A. Coping with complexity. *Xerox Cognitive and Instructional Sciences Series* **CIS-15**, April (1981a).

Sheil B. A. Teaching procedural literacy. *Proceedings of ACM Annual Conference* 125 - 126 (1980).

Sheil B. A. Teaching procedural literacy. *Proceedings of ACM Annual Conference* 125 - 126 (1980).

Sheingold K., Kane J., Endreweit M. & Billings K. *Study of issues related to the implementation of computer technology in schools*. Final Report, National Institute of Education (1981).

Sheppard S. B., Curtis B., Milliman P. & Love T. Modern coding practices and programmer performance. *IEEE Computer* **5**, 41 - 49 (1979).

Shif Z. I. Development of children in schools for mentally retarded. In *A Handbook of Contemporary Soviet Psychology* (eds. Cole M. & Maltzman I.). Basic Books, New York (1969).

Siegler R. S. Information processing approaches to development. In *Mussen Handbook of Child Psychology (4th edn). History, Theory, and Methods* (eds. Kessen W.), Vol. 1. John Wiley, New York (1983).

Shneiderman B. Measuring computer program quality and comprehension. *International Journal of Man-Machine Studies* **9**, 465 - 478 (1977).

Shrobe H. E., Waters R. & Sussman G. *A hypothetical monologue illustrating the knowledge of underlying program analysis* (Memo No. 507). MIT Artificial Intelligence Laboratory, Cambridge, MA

(1979).
Sime M. E., Arblaster A. T. & Green T. R. G. Reducing programming errors in nested conditionals by prescribing a writing procedure. *International Journal of Man-Machine Studies* 9, 119 - 126 (1977).
Simon H. A. Problem solving and education. In *Problem Solving and Education. Issues in Teaching and Research* (eds. Tuma D. T. & Reif F.) Halsted Press, New York (1980).
Simon H. A. & Hayes J. R. The understanding process: Problem isomorphs. *Cognitive Psychology* 8, 165 - 190 (1976).
Smith E. E. & Bruce B. C. *An outline of a conceptual framework for the teaching of thinking skills* (Report No. 4844) Prepared for National Institute of Education. Bolt Beranek & Newman, Cambridge, MA (1981).
Soloway E., Bonar J. & Ehrlich K. Cognitive strategies and looping constructs: An empirical study. *Communications of the ACM* **26**, 853 - 860 (1983).
Soloway E. & Ehrlich K. Tacit programming knowledge. *Proceedings of the Fourth Annual Conference of the Cognitive Science Society*, Ann Arbor, Michigan, 4 - 6 August (1982).
Soloway E., Ehrlich K., Bonar J. & Greenspan J. What do novices know about programming? In *Directions in Human - Computer Interactions* (eds. Shneiderman B. & Badre A.). Ablex, Hillsdale, NJ (1982).
Soloway E., Lochhead J. & Clement J. Does computer programming enhance problem solving ability? Some positive evidence on algebra word problems. In *Computer Literacy. Issues and Directions for 1985* (eds. Seidel R., Anderson R. & Hunter B.). Academic Press, New York (1982).
Soloway E., Rubin E., Woolf B., Bonar J. & Johnson W. L. MENO-II: An AI-based programming tutor. Yale University Department of Computer Science, Research Report #258, December (1982).
Spiro R. J., Bruce B. C. & Brewer W. F. (eds.). *Theoretical Issues in Reading Comprehension*. Erlbaum, Hillsdale, NJ (1980).
Statz J. *Problem solving and Logo*. Final report of Syracuse University Logo Project, Syracuse University, New York (1973).
Sternberg R. J. & Rifkin B. The development of analogical reasoning processes. *Journal of Experimental Child Psychology* 27, 195 - 232 (1979).
Thayer R. H., Pyster A. B. & Wood R. C. Major issues in software engineering project management. *IEEE Transactions on Software Engineering* **SE-7**, 333 - 342 (1981).
VanLehn K. Bugs are not enough: Empirical studies of bugs, impasses and repairs in procedural skills. *Xerox Cognitive and Instructional Sciences Series* **CIS-111**, March (1981).
Vygotsky L. S. *Mind in Society* (eds. Cole M., John-Steiner V., Scribner S. & Souberman E.). Harvard University Press, Cambridge, MA (1978).
Waters R. C. The programmer's apprentice: Knowledge based program editing. *IEEE Transactions on Software Engineering* **SE-8** (1) (1982).
Watt D. Logo in the schools. *Byte* 7, 116 - 134 (1982).
Weir S. *Logo as an information prosthetic for the handicapped* (Working paper No. WP-9). MIT, Division for Studies and Research in Education, Cambridge, MA, May (1981).
Weir S. & Watt D. Logo: A computer environment for learning-disabled students. *The Computer Teacher* 8, 11 - 17 (1981).
Werner H. The concept of development from a comparative and organismic point of view. In *The Concept of Development* (ed. Harris D. R.). University of Minnesota Press, Minnesota (1957).
Werner H. Process and achievement. *Harvard Educational Review* 7, 353 - 368 (1937).
Young R. M. The machine inside the machine: Users' models of pocket calculators. *International Journal of Man-Machine Studies* 15, 51 - 85 (1981).
Youngs E. A. Human errors in programming. *International Journal of Man-Machine Studies* 6, 361 - 376 (1974).