On the Cognitive Prerequisites of Learning Computer Programming

Roy D. Pea
D. Midian Kurland

Technical Report No. 18

ON THE COGNITIVE PREREQUISITES OF LEARNING COMPUTER PROGRAMMING*

Roy D. Pea and D. Midian Kurland

Introduction

Training in computer literacy of some form, much of which will consist of training in computer programming, is likely to involve $3 billion of the $14 billion to be spent on personal computers by 1986 (Harmon, 1983). Who will do the training? "hardware and software manufacturers, management consultants, retailers, independent computer instruction centers, corporations' in-house training programs, public and private schools and universities, and a variety of consultants" (ibid., p. 27). To date, very little is known about what one needs to know in order to learn to program, and the ways in which educators might provide optimal learning conditions. The ultimate success of these vast training programs in programming--especially toward the goal of providing a basic computer programming competency for all individuals--will depend to a great degree on an adequate understanding of the developmental psychology of programming skills, a field currently in its infancy. In the absence of such a theory, training will continue, guided--or to express it more aptly, misguided--by the tacit "folk theories" of programming development that until now have served as the underpinnings of programming instruction. Our paper begins to explore the complex agenda of issues, promise, and problems that building a developmental science of programming entails.

Microcomputer Use in Schools

The National Center for Education Statistics has recently released figures revealing that the use of micros in schools tripled from Fall 1980 to Spring 1983. The outcome of that leap is that there are now 120,000 microcomputers for students in 35% of the country's public

---

*The work upon which this publication is based was performed pursuant to Contract No. 400-83-0016 of the National Institute of Education. It does not, however, necessarily reflect the views of that agency.

schools: 22% of these are in elementary schools, and 64% are located in secondary schools (Reed, 1983). A staggering 4.7 million precollege students worked at computer terminals during the 1981-1982 school year. Yet the National Education Association reported in March 1983 that just 11.2% of the country's public school teachers use computers in teaching.

Problems with Questions of the "Cognitive Demands" of Programming

A common set of questions voiced by those wishing to learn computer programming (but perhaps as commonly voiced by those offering programming training or courses) is: "What do I need to know in order to learn to program?" "Do I need to be good at mathematics? If I'm not, can I still learn to program?" "Do I need to be a highly developed logical thinker?" In the academic community, these questions get expressed in the jargon of the social sciences: "What are the cognitive demands of programming?" or "What mental abilities are cognitive prerequisites for learning to program?" or "Are there individual differences in programming skill development?" The common fear for the individual who would like to learn programming, and the concern of educators and employers (frequently motivated by cost effectiveness), is that there are some persons who are either not capable of being trained to program, or who are not "developmentally ready" in that they need to learn or know more fundamentally relevant things before embarking on the task of learning to program. While these questions are subject to empirical analyses, we have found in reviewing the literature that the uses of such empirical analyses are often quite pernicious. If persons do not get a high enough score on a "programming aptitude measure," they may be denied educational or employment opportunities. One study is quite explicit on the uses to which they believe such findings should be put--for student advising and placement:

> The rapid increase in the need for personnel in these areas is attracting many individuals needing education, and the number who will not succeed with this education will increase....The increase in the number of such students is already being observed by many schools, resulting in the use of relatively scarce faculty resources to educate individuals who will not successfully complete a technical course, while keeping out students who might succeed if permitted to enroll. (Wileman, Konvalina & Stephens, 1981, p. 226; our emphases)

Fowler & Glorfeld (1981, p. 96) note that "the need to identify a potentially successful student is very important for reasons such as

early counseling into an appropriate career path and formation of honors sections." In another study, Newstead (1975, p. 89) goes so far as to say that "One can conclude that programming ability (as measured in this study) may be much more innate than 'business training course spiels' would have one believe. Anyone cannot be a programmer [sic]."

The logic of these approaches is not hard to follow. Let the rich get richer, while the poor recede into the nontechnical background of society. Never mind that it may be possible to instruct "those who will not succeed" in another way (and the pedagogical alternatives for learning to program are many and diverse [Lemos, 1975, 1980; Papert, 1980]). Instead, assume that "this education" that they will not succeed with is immutable and adequate for anyone wishing to learn to program "if only they had the right stuff." While some may have felt that this attitude had some justification when programming was an optional activity (something one could, but need not do), it is a problem of great seriousness today, since at least a basic understanding of (modicum of competence in) how to write and understand computer programs is at the core of the needs for any member of a computer literate society (e.g., Sheil, 1981a, 1981b). We cannot continue to dismiss those who have difficulties in learning to program as flies in the ointment of a well-oiled training machine. Instead, we are obliged to try out or, if necessary, develop new means for instruction that allow all an equal opportunity to learn about and participate in the computational powers of a technological society, one which has an impact on and will continue to have an even greater impact on the educational, work, and family lives of members of our society.

Overview

In preparing this report, we found that when we critically considered the available literature that aims to address the relationships between "programming aptitude" and interests of individuals and their programming performance, the static nature of these studies was apparent in that the research questions asked have not varied substantially in nearly 30 years. Ever since the 1950s, when the Programmer Aptitude Battery was developed by IBM to help select programmer trainees, consistently modest correlations (at their best from 0.5 to 0.7, hence accounting for only a quarter to a half of the variance), and in many cases much lower, have existed between an individual's score on such a measure and his or her assessed programming skill. In addition, ever since the 1950s, global measures of programming skill, such as grade in a programming training course or supervisor ranking, have served as skill assessments. Studies in 1983 still take

for granted the utility of this multivariate approach, and offer no greater insights--certainly none that are instructionally relevant--into what makes a successful programmer than they did three decades ago. The same is true of interest measures and programming skill assessments. These studies always ask whether particular aptitude variables have an effect on programming success, rather than the more fundamental psychological question of how they might have such effects, in terms of component skills or knowledge representations mediating specific programming activities.

Given these insufficiencies, we must often step back during the course of our review and survey the presuppositions that have guided this line of research, asking whether they are warranted, culling from the research literature the few studies that suggest new and promising directions, and asking many more questions than we are able to answer on the basis of available evidence. While the dearth of answers to these questions is at times disconcerting, we feel that a groundclearing is in order. The available foundations are unstable, so we must uproot them in a search for new metaphors, new ways of seeing what programming is that people may learn to do it, and what it is about people that they are able to do programming. Since the available programming aptitude literature is built upon such questionable foundations, a point-by-point review of this literature in terms of subject populations, specific measures, and correlations obtained is not useful, although we will provide brief surveys and extensive citations of the hundred or so studies of this kind. On the other hand, we are heartened by some recent studies on the cognitive psychology of programming that begin to unravel the complex of mental activities essential to programming. In our intellectual travels through the development of programming skills, we become more and more compelled by the importance of one dominant motif: the role of purpose and goals in programming. What one needs to know in order to program will depend in fundamental ways on one's programming goals. This point has repercussions all the way from how one does programming instruction, to what kinds of programming instruction are selected on the basis of the educational values and goals that define one's programming pedagogy. To take a simple example: documenting a program and the mental skills that are required may be unnecessary for a single-use program for one's home computer system. But it is a central concern if one requires a program for the public domain that is understandable and maintainable by others. So we will ask not, "What are the cognitive demands of computer programming?" as if programming were a unified homogeneous activity (which we challenge below), but rather, "What are the cognitive demands of doing computer programming activity X with goal Y?"

With these provisos in mind, we will now begin to develop the ground raising which we have promised.

Background Issues

What Is Computer Programming?

We will define the core sense of "computer programming" as that set of activities involved in developing a reusable product consisting of a series of written instructions that make a computer accomplish some task. It is interesting to note that, although the sense of "computer programming" has not varied in nearly four decades, the set of activities involved in doing programming has undergone major qualitative transformations. In the early days of programming, for example, the programmer had to know the details of the computer hardware in order to write a program that worked; today this is no longer true. An important consequence of this evolution of the set of activities that constitutes programming is that the "cognitive demands" made by computer programming needs specification at the level of programming subtasks, or component activities. Therefore, we must ask about the variety of cognitive activities involved in computer programming as it is carried out today, and especially as it is carried out by children as they attempt to master this complex skill.

From our own work as well as our reading in the literature on programming, we find that at least four distinct levels of computer programming ability can be identified. While these levels represent pure types and are not characteristic of a single individual, they do capture some of the complexity of what it means to learn to program. While we will take up the issue of levels of programming skill development in more detail below, it is important in attempting to build an adequate characterization of the programming process to bear in mind the range of abilities included under the heading of "being able to program."

Level I: Program user. Before learning how to program, one typically learns to execute already written programs such as games, demonstrations, or CAI lessons. What is learned at this level is not trivial (i.e., what the return key or the reset key does, how to boot a disk, how to use a screen menu), but gives no information to the novice on how a computer program works or even that there is a program controlling what happens on the screen. Many adult computer users never advance past this level. One does not need to know about programming to be a word processor operator, make airline reservations, process payroll checks, design a budget, or do any of a growing number of computer-based activities. However,

Level I users remain at the mercy of the program they are using and are powerless to effect changes in it. While it can be argued that as programs get better, there will be less need for the average person to make programming changes, we would argue that without some familiarity with programming, the user is less likely to appreciate fully the power and potential of this technology. In addition, without some appreciation of programming, the user is not likely to take full advantage of optional parameters built into sophisticated application programs which themselves constitute a high-level form of programming.

Level II: Code generator. At this level the student knows the syntax and semantics of the more common commands in a language. The user can read someone else's program and know what each line accomplishes. The student can locate bugs that prevent 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 that he or she has seen previously. When programming, the student does very little preplanning and does not bother to document his or her program. There is no effort to optimize the coding, use error traps, or make the program user-friendly and crash resistant. A program at this level might simply consist of printing the student's name over and over on the screen or drawing the same shape repeatedly 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.

Level III: Program generator. At this level, the student has mastered the basic commands and is thinking in terms of higher level units. Sequences of commands that accomplish program goals are known (e.g., locate and verify a keyboard input, sort a list of names or numbers, read data into a program from a separate file). The student can now read a program and say what the goal of the program is, what functions different parts of the program serve, and how the different parts are linked together. The student can locate bugs that prevent the program from functioning properly (e.g., a sort routine that fails to correctly place the last item in a list) or bugs which 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 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 so that the program may be maintained by others. Within this gener-

al level, one can identify many sublevels of computer programming skill.

Level IV: Software developer. At this level, the student is ready to write programs that are both complex and are intended to be used by others. The student now knows several languages and has a full understanding of all their features and how the languages interact with the host computer (e.g., how memory is allocated, how graphic buffers can be protected from being overwritten, how peripheral devices can be controlled by the program). 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, most likely, how the programs could be better written or adapted for other purposes. The student now writes programs with sophisticated error traps and built-in tests to aid in the debugging process and to ensure that the program will be reliable, provable, and maintainable. In addition to writing code that accomplishes the program's objective, the student can optimize coding to increase speed and minimize the amount of memory required for running. To decrease the time needed to write programs, the student will draw heavily on software libraries and programming utilities. Finally, the student will often design the program before generating the code, will document the program fully, and will write the program in a structured fashion thus making it easy for others to read and/or modify. Major issues in software engineering at this level of expertise are discussed by Thayer, Pyster and Wood (1981).

There are many methodological problems with assessing computer programming abilities across these four major levels and their many sublevels. While psychological studies of expert and novice programmers have revealed some efficient measures that exploit the differences in programming-specific problem-solving strategies--specifically, debugging (Jeffries, 1982) and program memory organization (diPersio et al., 1980)--determining whether or not a person can program at some specified level of expertise remains a difficult task.

Demands of Learning to Program: The Problem of Differentiation

The question of the cognitive demands of computer programming is an enormously complex one, 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. In addition, a programmer may be highly developed in some aspects of programming but not in others. For example, it is not uncommon to find programmers who

can write highly efficient and elegant code, but not code that can be understood or maintained by other programmers. Typically, in research on the psychology of computer programming, "learning to program" has been equated with learning the syntax and the definition of commands in a programming language, just as reading is often equated with skill in decoding. However, once past the initial level of skill acquisition, what we mean by "reading" is actually reading comprehension, which entails an elaborate body of world knowledge, comprehension monitoring, inferencing, hypothesis generation, and other cognitive and metacognitive strategies that take years to develop fully. This lesson has been etched in high relief through the intensive efforts necessary to develop artificial intelligence systems that "understand" text (e.g., Anderson & Bower, 1973; Schank, 1982; Schank & Abelson, 1977). Skilled reading also requires wide experience with different types of texts (e.g., narrative, essays, plays, poetry, debate, biography) and with different goals of the reading activity (such as reading for meaning, content, style, pleasure). Skilled computer programming is similarly complex and context-dependent, which makes the problem of assessing the cognitive demands of "learning to program" all the more acute.

This issue of "cognitive demands" and the corresponding problem of selecting components of the question that are researchable has been a general one. The idea that some development may serve as a necessary prerequisite for some other development is familiar from the literature of moral reasoning (e.g., Tomlinson-Keasey & Keasey, 1974), language development (e.g., Sinclair, 1969; Slobin, 1973, 1982) and, more generally, from the "invariant" developmental stage sequence arguments offered by Piaget as central to his structuralist developmental theory. This approach to the cognitive demands of programming has recently begun to be applied to programming as well (Favaro, 1983). In each case, the empirical testing of the prerequisite character of some specific cognitive achievement for some other cognitive achievement has depended on a refinement of the general question into specific questions that are empirically researchable. Rather than asking, as in the early days of the cognitive prerequisite controversy in developmental psycholinguistics--"Does language learning require prior concept development for the ideas expressed in language?"--current language development research recognizes that such a question must be asked for specific language constructions or subparts of language, and that the answer will depend on the linguistic forms chosen (e.g., Johnston, 1982). We will urge a similar differentiation for questions about the "cognitive demands" or "prerequisites" of learning to program--cognitive prerequisites in order to do what specifically in programming?

What Programming Is as a Cognitive Activity

In this section, we first outline recent findings about the cognitive psychology of expert programming, then provide a brief account of available theories of the cognitive subtasks involved in programming, describe existing accounts of what "learning to program" involves, and then critique the popular accounts of what learning to program requires.

One can begin to analyze what programming skill is as a cognitive activity by engaging in detailed analyses of what expert programmers do, and what kinds and organizations of knowledge they appear to have stored in memory in order to do it. This research strategy has revealed significant general features of expert problem-solving competence and performance for a wide variety of other subject 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 has begun to provide some insights into the components of programming skill.

Recent studies of programmers suggest that high-level computer programming skill is characterized by a giant assembly of highly specific, low-level knowledge fragments (Brooks, 1977; Atwood & Ramsey, 1978). For example, 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 & Schrobe, 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 computer programming "schemas," or recurrent functional chunks of programming code that programmers are alleged to use.

There is some behavioral evidence from studies of programmers that supports these rational and introspective analyses of "chunks" of computer programming knowledge. Eisenstadt, Laubsch and Kahney (1981) have developed a Logo-like software programming language called SOLO and used it to introduce computer-naive college psychology students to computer programming. In an analysis of novice student programs, they found that most programs were constructed from a small set of basic program schemas comparable to the "plan library" of Schrobe et al. (1979). Jeffries (1982), in a comparison of the debugging strategies of novice PASCAL 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 his colleagues (Bonar, 1982; Ehrlich & Soloway, 1983; Johnson, Draper & Soloway, 1983; Soloway & Ehrlich, 1982; Soloway, Ehrlich, Bonar & Greenspan, 1982; also see Kahney & Eisenstadt, 1982) have begun to construct a "plan-based theory of computer programming" which also postulates the use of recurrent plans as "chunks" in program composition. Such plans were identified in preliminary analyses of programs written by PASCAL novices (e.g., the "counter variable plan"). What is missing here, for our purposes, is a genetic account of the construction of such plan schemas from programming instruction, experience, and prior knowledge that is brought to the task of learning to program. (For the interested reader, a general account of schema theory in cognitive science is provided by Rumelhart, 1980.)

A second, related characteristic of computer programming skill is the large number of component rules that underlie an expert's generation of programming problem solutions. In an analysis of one programmer's work on 23 different problems, Brooks (1977) demonstrated in a detailed model that about 104 rules were necessary to generate the protocol behavior. Further, 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 characteristic of computer programming skill is the ability to construct detailed mental models of how the computer is functioning when a computer program is running (Sheil, 1980, 1981a). The expert programmer can build dynamic mental representations, or "runnable mental models" (Collins & Gentner, 1981) in which they can simulate computer operations in response to specific problem inputs. Brooks (1977) has characterized these mental operations as "symbolic executions." 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). We should note that not all aspects of program understanding are mediated by hand simulation; often experts engage in a prior global search for program organizational structure, a strategy akin to that of expert text readers (Brown, 1983b; Brown & Smiley, 1978; Spiro, Bruce & Brewer, 1980) and guided by adequate program documentation.

Expert programmers not only have more information about the computer programming domain, but remember larger, meaningful chunks of information that enable them to perceive programs and remember them more effectively than novice programmers. The classic Chase and Simon (1973) finding of short-term memory advantages for chess

experts over novices for meaningful chessboard configurations but not for random configurations has been repeatedly replicated for the domain of computer programming (Curtis, Sheppard, Milliman, Borst & Love 1979; McKeithen, Reitman, Rueter & Hirtle, 1981; Norcio & Kerst, in press; Sheppard, Curtis, Milliman & Love, 1979; Shneiderman, 1977). For example, McKeithen et al. (1981) used the Reitman-Rueter (1980) technique for inferring individual subjects' chunking of key ALGOL programming concepts in memory from recall orders to discover the specifics of memory organization that may facilitate this performance difference. They found extensive differences in the mnemonic strategies used by beginning, intermediate, and expert ALGOL programmers to memorize ALGOL keyword stimuli. Notably, experts clustered the keyword commands according to meaning in ALGOL (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), with intermediates somewhere in between. In a related finding, Adelson (1981) presented computer programming novices and experts with a recall task in which stimuli were lines of programming code which could be organized either procedurally (into programs) or syntactically (in terms of order relationships between different control phrases of the computer language). She found that experts recalled program lines "in the order in which they would have been evaluated in a running program," whereas novices clustered by syntactic category. Recall clusters for experts were thus functionally or "deeply" based, whereas those of novices were based on "surface" features of programming code. This distinction is reminiscent of the striking developmental shift from surface structure-based, or "syntagmatic," word associations to functional category-based, or "paradigmatic," word associations during childhood (e.g., Nelson, 1977).

DiPersio, Isbister and Shneiderman (1980) have carried this line of research further by demonstrating that performance on a program memorization/reconstruction task provides a useful measure and predictor of computer programming ability. Scores on program logic reconstruction tasks and performance on college class exams in programming were significantly correlated. The authors attributed this result to the extent of subjects' "semantic" knowledge base for the programming language, that is, the functional nature of the code (which we have more appropriately designated as the "pragmatics"[1] of programming skill). Such results are encouraging insofar as they suggest the utility of such a memory task as one measure for assessing computer programming skill development. More research will be required, however, for the performance levels of individuals rather than groups to be inferred from their performance on program memory tasks.

It is also a widely replicated finding that expert programmers debug their programs in different ways than do novices (Atwood & Ramsey, 1978; Gould, 1975; Gould & Drongowski, 1974; Youngs, 1974). Perhaps most important is the recent finding (Jeffries, 1982) 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).

In terms of identifying the specific cognitive activities involved in programming (the necessity of which we argued for earlier in our discussion of the cognitive demands of computer programming), we need a more comprehensive account of the task of programming. Recent research in cognitive science provides such accounts, and to these theories we now turn.

Theories of Cognitive Subtasks Involved in Programming

It is the consensus of cognitive psychologists who have developed global theories of expert programming skill that computer programming is highly complex since "it involves subtasks that draw on different knowledge domains and a variety of cognitive processes" (Pennington, 1982, p. 11). Just as in the case of theories of problem solving in general, cognitive theories that have been developed of expert programming articulate a set of distinctive cognitive activities that take place in the development of a computer program. These activities are required for programming whether the programmer is novice or expert, since they constitute phases of the problem-solving process in any general theory of problem solving (e.g., Heller & Greeno, 1979; Newell & Simon, 1972; Polya, 1957). They may be summarized as: (1) understanding the programming problem; (2) designing or planning a programming solution; (3) writing a programming code that implements the plan; and (4) comprehension of the written program and program debugging. We will discuss each of these cognitive subtasks in turn, with an eye toward thinking about what kinds of cognitive demands each of them may make on the programmer.

1. Understanding the Problem

It is generally agreed that in attempting to solve a problem, the problem solver first sets up some form of "problem representation" in working memory which is used to model a problem in terms of what the problem solver knows about the problem domain, and how that knowledge is organized for him or her. In recent studies (e.g., Chi, et al., 1981), substantive qualitative as well as quantitative differences in the problem-solving processes of experts and novices for a given content domain, such as physics, have indicated that experts

set up radically different kinds of problem representations in their early attempts to understand the problem presented. In the case of applying physics to mechanical problems, experts engage in extensive qualitative problem analysis, or processes of problem understanding, before attempting to solve the problem, for example, through setting up a physical representation of the problem situation that was initially depicted in words (Larkin, 1977; McDermott & Larkin, 1978; Simon & Simon, 1978). Physics experts focus on deep structural features of the problems in problem categorization studies, sorting together problems which would be solved according to specific laws of physics, unlike novices, who focus more on the surface structural features of the problem structure, such as the objects involved, physics terms used, and the physical configurations described in the problem (Chi et al., 1981). What the expert appears to know is what kinds of features of the problem should constitute part of their problem representation; this knowledge is apparently facilitated by large-scale memory units in terms of problem types that are identified in terms of deep structure, and by the experts' facility in rapidly building qualitative physical symbolic representations of the verbal problem statements.

In a discussion of a computer-implemented model of physics problem solving, Larkin (1980) notes that the importance of a deep problem representation during the problem understanding process is that

> using these qualitative features, the [computer simulated] solver tentatively selects a method for solving the problem. It then applies key physics principles from that method to generate qualitative information about the problem--for example, information about the direction an object moves [our Design and Planning cognitive subtask]. When sufficient information has been generated to solve the abstracted qualitative problem, the model solver elaborates this qualitative solution by generating corresponding quantitative equations [our Program Coding phase] to actually solve the original problem. (p. 116)

What is illuminating for thinking about computer programming from problem-solving studies such as these, and in other domains such as geometry (Greeno, 1976) or writing (Flower & Hayes, 1979), is that the problem solver must have substantial domain-specific knowledge in order to set up a functional problem representation. With respect to understanding the problem, Larkin's physics solver "had to know what kind of features to abstract in constructing a useful simplified problem." For developing a problem-solving plan, it "had to know what kind of operations he could apply to solve abstracted problems,"

and for working out the details of the problem solution had to know "how these [operations] were to be elaborated when he returned to construct a full solution." Although solution debugging is not mentioned in this work, presumably he would also have to know how to check whether or not the solution was a correct one.

So domain-specific knowledge is very important for understanding the programming problem. If a child was asked to write a graphics program to draw a Colonial home, she would have to know about what Colonial houses looked like, their key identificational features, and so on. Similarly, to develop a game system, a child would need to know the many domain-specific facts about computer games, such as variable skill level, score feedback, and so on. Since domain-specific knowledge is such a fundamental aspect of understanding programming problems, serious thought needs to be given to what we know about conceptual development for any given content domain for which we are interested in posing programming problems for children. Certain domains, such as statistics, or simulations for complex domains such as ecosystems and economics, may well be out of reach for school-aged children, and constitute inappropriate programming project content. But the great variety of domains that children are learning about in school should provide ample opportunity for rich programming projects.

As for the specifics of the categories or types of problems that the expert programmer is able to identify in attempting to understand the programming problem, many different alternatives have been suggested, and little empirical evidence, even for adult experts, exists to distinguish them. We summarize those described by Pennington (1982) below:

a) Function-oriented. Problems would be seen as indicating different program goals or functions, in terms of what is to be accomplished, such as "update inventory accounts and produce reports" (e.g., Balzer, Goldman & Wile, 1977; Shneiderman & Mayer, 1979).

b) Data/process-oriented. Problems would be seen as specifying external object classes (e.g., updates, inventory accounts, status report), and operations (e.g., transform initial objects to final ones) applied to specific classes of objects (e.g., Brooks, 1982; Miller & Goldstein, 1977).

c) Sequence-oriented. Problems would be decomposed into their basic components or procedures, and problem representations would contain sequencing information (e.g., Atwood, Jeffries & Polson, 1980).

As noted above, the problem representation that serves as the outcome of the problem-understanding process is one of the most fundamental components of the problem-solving process, for programming as well as other content domains. For this reason, we expect the cognitive subtask of understanding the programming problem to be developmentally central. The cognitive demands of understanding programming problems, as we have seen, will depend at least upon the extent and organization of a child's domain-specific knowledge that is required for the problem at hand. But since the adult expert programmer literature is currently equivocal on what forms such problem representations may take, we cannot make precise the cognitive demands of program understanding. For at least some of the proposed alternatives, data/process-oriented and sequence-oriented, the child would need to be able to learn about the different classes of data objects and operations, in the first case, and about the concepts of "procedures" and "sequentiality," in the second case. Such basic requirements have direct implications for defining a "core" of minimal programming knowledge, to be discussed below.

2. Designing/Planning the Program

After achieving an initial problem representation, the programmer needs to map out a plan or design for the program to be written later in programming code. Atwood et al. (1980) provide an informative description of the requirements of this process:

> Software design is the process of translating a set of task requirements (functional specifications) into a structured description [design or plan] of a computer system that will perform the task. There are three major aspects to this description. The original specifications are decomposed into a collection of modules, or substructures, each of which satisfies part of the original problem description. This is often referred to as modular decomposition of the problem. In addition, these modules must communicate in some way. The designer must specify the interrelationships and interactions among the modules [also called procedures in smaller systems]. This includes the control structure, which indicates which modules are used by a given superordinate module to do its work and the conditions under which they are used. Lastly, a design may include a definition of the data structures required. (p. 3)

According to Brooks (1982), one-third of the entire time a program team spends on a software project (including coding and testing) should be spent planning the task. Atwood et al. (1980), in a de-

tailed analysis of the think-aloud protocols of two expert software designers as they solved a software design problem, found that they had available many general design strategies, such as problem decomposition, subgoal generation, retrieval of known solutions, generation and principled or "policy driven" selection of alternative solutions, and evaluative analysis and debugging of solution components. It is of some importance in this respect that a major move in programming instruction is to treat programming as an instance of a general problem of structured design (Floyd, 1979), rather than as machine and programming language-specific (Sheil, 1980).

At this point, someone is bound to object that, in the programming process, it is possible to bypass this step of program development altogether, that one may first make an initial reading of the problem, then sit down at the keyboard and begin composing code to achieve the task. And it has been said (Galanter, 1983) that one frequently finds much preplanning in PASCAL (a compiler language) programming, but often little or no planning prior to code writing for programming languages such as BASIC (an interpreted language). What are we to make of this observation in terms of defining design and planning as a distinct cognitive subtask in programming? Is it optional? The answer to this question certainly has consequences for the cognitive demands of programming, if one subtask ingredient to it involves whatever cognitive prerequisites there are for planning and design.

In response to this objection, we allow for the distinction commonly made, and applicable to the cognitive activity of programming, between planning-in-action versus preplanning (Rogoff & Gardner, 1983). In terms of this distinction, what the BASIC programmer is engaged in as he or she sits down and begins to generate programming code without a prior plan is planning-in-action, making decisions as he or she goes about the structure of the program, which evolves as the materials of the program are created. Schon and Bamberger (1982) have described the outcomes of such a planning-in-action creative process in art, music, and other related domains as a consequence of an iterative series of "conversations" the creator has with his or her partial creations. Bereiter (1979) has characterized a similar process in composing language text as "epistemic," in which one comes to see and understand new things as one channels one's ideas into a written product. But to return to programming, although planning-in-action is certainly possible, even sufficient, to produce a program, we expect such a planned-in-action program often to have great costs for the beginning programmer. The reason has to do with the anticipated difficulties of comprehension and debugging when one goes back to try to understand what one has done in

writing a program not built with foresight. Of course, for expert programmers the sheer automaticity of many programming projects, since they are able to recall successful plans for similar programs or software systems, will mean that little preplanning will be required for the program code generation. In other words, the adult programmer often can integrate the subtasks of planning and code writing. But the child as novice programmer is not at that level of understanding, and does not have a store of programming schemas available for ready reference during planning-in-action while creating a program. So we will continue in our discussion of the cognitive demands of programming to include the cognitive demands of the planning/design cognitive subtask of programming.

In terms of cognitive demands, details of the various proposals for how planning takes place in programming, whether the top-down orientations with successive plan refinement or more opportunistic approaches analogous to the work of Hayes-Roth and Hayes-Roth (1979), imply that preadolescents may have difficulties generating program designs, particularly ones that are complex and require hypothetical and counterfactual reasoning more characteristic of the older child. We shall provide a brief review of this literature in the section on conceptual development and programming. One of the principal cognitive problems comes down to what Stefik (1981a, 1981b) in his artificial intelligence work on planning called the recognition of "commitments" of plan components, involving seeing ahead or symbolically executing plans or plan parts in order to mentally simulate the consequences of particular design proposals, and finding problems with those commitments that indicate the need for a new design.

Pennington (1982) has indicated problems with the very general nature of proposals that program plans are hierarchical in nature, such hierarchy representing successive refinements of the program description until a solution that can be mapped out in programming code is reached. How do each of these successive versions of the plan represent and further elaborate the four basic types of program information, that is: (a) the purpose/function of a particular plan unit; (b) the structure of the data objects; (c) the sequencing of operations (control flow); and (d) the transformations of data objects (data flow)? As she notes:

> little empirical evidence exists on how programmers coordinate and transform the four types of information embodied in a final programming product, yet it seems that these coordinations underlie the complexity of programming and other planning tasks. (p. 19)

We would agree here, but note further that studies of the development of planning for any content domain are in their infancy (Friedman, Scholnick & Cocking, 1983; Pea, 1982). What evidence exists indicates that, at least for planning problems utilizing familiar classroom chores in a chore-scheduling task where the goal is to find the shortest possible plan for doing all the chores, children from 8 to 12 years of age are capable of substantial "debugging" of long plans through revisions.

We may now ask: What programming knowledge is necessary to design the program plan? As discussed earlier, expert programmers chunk familiar patterns of programs, as indicated by the quality of their program comprehension as indexed by program recall. It is currently unclear how this knowledge is organized or acquired (an account of cognitive skill acquisition comparable to J. R. Anderson [1982] could be offered as a model of the latter), although these are fundamental developmental questions.

Some proposals have been made on the character of the expert programmer's memory store, but little evidence is available. Some alternatives as reviewed by Pennington (1982) are set out below, and aim to answer the question of what programming knowledge schemas or chunks are available to the expert. The implication for our questions about children are that whatever kinds of programming knowledge are required by children of the age of interest would have to be learnable in order for program plans drawing on such knowledge to be achievable. So what are the schemas?

a) Anything from transactions (less than a programming statement) to chunks (unit that accomplishes a goal) to higher level chunks (familiar algorithms) (Mayer, 1981);

b) Hierarchy of patterns from operations (compare two numbers) to small algorithms (sum an array) to large algorithms (bubble sort) to program patterns (Shneiderman 1980; Shneiderman & Mayer 1979);

c) Known solutions/plans/plan elements (Atwood, Jeffries & Polson, 1980; Balzer et al., 1977; Miller & Goldstein, 1979);

d) High-level programming units such as loop and recursion structure (Rich, 1981; Rich & Shrobe, 1979; Soloway & Woolf, 1980; Soloway et al., 1982);

e) Building block units such as basic loop, augmentation, and filter (Waters, 1979);

f) Categories with internal structure, such as rules for data structures, enumerations (looping constructs), mappings, etc. (Barstow, 1977, 1979).

3. Coding a Program

This phase of program development consists of a translation from the most refined version of the program design into the programming code. Brooks' (1975) estimate of less than 200 coding templates necessary to define the syntactical arrangements of code in statements makes clear why it is said in the programming industry that coding is a much simpler process than program design, which appears to involve a much more vast and initially ill-defined problem space. According to Brooks (1982), only one-sixth of the time allocated to a software project should invlove the actual writing of the program code. It is unlikely that this phase can be completely independent of the program planning phase, since different programming languages provide different options for plan implementation, such as "the availability (or lack) of linked list data structures" (Pennington, 1982, p. 24).

Brooks' (1975, 1977) study of a programmer's coding performance found symbolic execution, or what we might describe as mental simulation, to be the major feature of the coding process. Brooks' account postulates that

> a plan element triggers a series of steps through which a piece of code is generated and then the programmer "symbolically executes" that piece of code in order to assign an effect to it. This effect is compared with the intended effect of the plan element; any discrepancy causes more code to be generated until the intended effect is achieved. Then the next plan element is retrieved and the process continues. Throughout this process a representation of the program is built up, storing information about the objects (variables, data structures, etc.), their meanings, and their properties. (See Pennington, 1982, p. 24)

Once again, as in the case of plan construction where symbolic execution plays a major role, we find that program coding requires substantial hypothetical thought. As for the cognitive demands of generating program code, we may note three general classes of apparent prerequisites: (1) ability to engage in hypothetical symbolic execution of code; (2) ability to learn the coding templates that define the syntactical knowledge necessary for code generation; and (3) ability to keep to the goal at hand, or program plan, unless deviations from it are required to generate the code; in such an

event the plan would need to be revised, with consequences for the code then to be generated to achieve it.

4. Comprehending and Debugging a Program

How do programmers comprehend programs? In order to debug or modify their programs, they need to learn from their own or others' programs. If they are to realize how much progress they have made in developing a program, comprehension must play a key role. One extremely useful paper in thinking about this problem is by Green, Sime and Fitter (1980), who emphasize at some length the importance but current neglect of developing means for "getting information out of programs as well as into them"--the program comprehension problem. They note that "some of the major problems [a programmer faces] come when the program is being debugged, or extended, or modified, or just when the past half-hour's work is being reviewed" (p. 894). Pennington (1982, p. 29) notes that "program comprehension also involves reversing the forward mappings from problem representation in the external domain to successive levels of plans to programming language code." Program comprehension would thus require an inferential retrieval of the program creation process.

Four very different views of the program comprehension process have been proposed, and they have not been compared in terms of their empirical validity: one is bottom-up, one is middle-out, one is top-down, and one is transformational (Pennington, 1982, pp. 26-27, whom we follow closely in this section). Shneiderman (1976) finds expert programmers to recall more gist, or high level logic, of the program than do novices. He later (1980) argued for a bottom-up construction of meaningful units of programming code, from operations to algorithms on up to the function of the program as a whole. Atwood and Ramsey (1978) have a multiple pass model analogous to Hayes-Roth and Hayes-Roth's (1979) opportunistic model of planning, in the sense that programmers utilize high-level and low-level information about the program structure advantageously in order to understand the program.

> On the first pass, some level of the [program] macrostructure is integrated, possibly as high as program function, possibly at some level of chunking. Successive passes lead to integration of lower level propositions (working down) and successive integrations of the macrostructure (working up). (Pennington, 1982, p. 27)

Brooks (1982) views program comprehension as hypothesis driven and as immediately seeking out the high-level schema for the program.

The program reader then is said to seek out evidence for predicted program components consistent with their high-level expectations. This process works iteratively until the program reader has assimilated all the code to understand its precise workings. The complex transformational account offered by Rich, Shrobe and Waters (Rich, 1981; Rich & Shrobe, 1978, 1979; Rich, Shrobe, Waters, Sussman & Hewitt, 1978; Rich & Waters, 1981; Shrobe, 1979; Waters, 1978, 1979) implies that program understanding is mediated by a hierarchical representation of three levels: (1) deep plans (purpose); (2) surface plans (in program structure); and (3) definitions of data objects, properties, and I/O specifications for program code segments.

What does all of this mean for the child who needs to be able to comprehend programs as one cognitive subtask of programming? Again, this is a complex question, since even at this level of subtask analysis this question is analogous to that of "What are the cognitive demands of (natural language) text comprehension?" which is far too general a question to be meaningful psychologically. The question asked should instead depend upon what kinds of text (in terms of text genre), logical complexity of the text, in terms of the inferences required to understand it (e.g., Clark, 1977), constituent statements, words and, in the case of the beginning reader, even letters, of which the text is composed (e.g., Gibson & Levin, 1975). At the most basic level, children would have to be able to read the lines of code and identify the meanings of the constituent elements of the program, or primitive commands. This much is basic. But a much more complex task is to understand the interrelationships between these lines of code, to recognize the units, modules, or procedures which make up the meaning of the program as a whole. Studies (e.g., Kurland & Pea, 1983) demonstrate that comprehending programs at multiple levels is a difficult task even for relatively experienced child programmers from 8 to 12 years of age. However, what is as yet unclear is whether children do not tend to comprehend programs at "deep" levels because they have difficulty decoding even the surface syntax, or whether for the type of programming activities children typically engage in there is little incentive to probe below the surface.

Summary

We have provided a brief account above of the four major constituent cognitive subtasks of programming insofar as they are currently understood in the literatures of cognitive science and software psychology. What we have observed is that even at this level of specificity, although we can articulate at a general level some kinds of knowledge and abilities that children should have in order to mentally

engage in these subtasks (which we will not review here), we found in each case that the four subtasks could be fruitfully decomposed still further. But surely such decomposition must at some point come to an end, or the resolution of our analysis will be so minute as to map one per one on each decision the programmer makes while programming. There are no doubt an infinite diversity of those, much like utterances in natural language, and a cognitive demand analysis that is an infinite list is not of much use. So how fine shall the grains of the cognitive demand analysis be?

Before falling into some existential abyss, let us not lose sight of the original goals of thinking about the cognitive demands of programming; that is, basically, to understand well enough what cognitive prerequisites different programming activities have so that children are able to gain entry to the world of programming, and so that overly complex programming subtasks are not required of them. Inevitably, there are those who will look for the curricular implications of these analyses, and a certain amount of fuzziness or imprecision will at first be likely. Insufficient data are available on the development of skills such as planning, symbolic execution, and problem understanding, and such data as do exist are derived from children's performances in task environments so unlike programming as not to be straightforwardly applicable to it. The consequence of this point is that specifying ages or prerequisite knowledge states at which certain programming activities can or cannot be undertaken is a risky business if done a priori, and cannot be warranted on the basis of available evidence. Instead, we need to design programming tasks in which children of different ages can attempt the programming cognitive subtasks we have outlined above, so that we can see on an empirical basis what children appear to be capable of.

We will now review the few observations that have been made to date on the development of programming skills. To anticipate, our concerns above will not be reflected in the available literature.

The Development of Programming Skills

What "Becoming a Programmer" Means: Children

The few available studies of children's programming have not been developmental in nature, articulating intermediate stages of competence en route to mastery and setting out constraints on the development of specific computer programming activities (such as those articulated above) in terms of prerequisite knowledge. Rather, these studies are observational and anecdotal studies of individuals' learning to program to show that "children can program," as well as to docu-

ment some of the motivational benefits of such computer programming experience (e.g., Papert et al., 1979). We find that little thought or research has been directed to the important problem of articulating intermediate levels of computer programming mastery. This is a serious knowledge gap since "understanding" is not an all-or-none mental state (e.g., Nickerson, 1982) and the processes undergirding developmental transformations of a person's understanding are of central concern for a developmental cognitive science. Because Dewey's concept of "readiness to learn" has proven to be of such wide instructional applicability, we believe that delineating these intermediate forms of understanding must be a goal of developmental research on programming. This dearth of knowledge is compounded with the problem that, from the popular literature on children and programming, it appears as if everything we have learned about cognitive development during the last quarter century has suddenly been rendered irrelevant by the advent of the microprocessor. For example, 5-year-olds who can get a graphics cursor to work in the Logo programming language are called "programmers," conveying the popular assumption that they have come to understand the logical operations ingredient to a program's flow of control, and are capable of all the cognitive subtasks of programming.

Child Novice Programmers

Only limited evidence, somewhat bleak in character, is currently available on levels of programming abilities achieved by individuals of different age groups in the precollege-age population. These statistics should not, of course, be taken to indicate what each child may be capable of if allowed his or own computer in school and the support of optimal instruction. In the National Assessment of Educational Progress survey of 2500 13-year-olds and 2500 17-year-olds during the 1977-78 school year (NAEP, 1980), even among the small percentage of students who claimed to be able to program a computer, "performance on flowchart reading exercises and simple BASIC programs revealed very poor understanding of algorithmic processes involving conditional branching" (cited by R.E. Anderson, 1982, p. 14).

It is worth noting in this context that in current instructional environments, children appear to have basic conceptual and representational difficulties in constructing dynamic mental models of what is happening when the computer is executing lines of their programs, which sets critical limits on the computer programming skill level that they attain. Furthermore, systematic but "naive" mental models or "naive epistemologies" (diSessa, 1982) of computer procedural functioning may initially guide and mislead children's understanding of programming, which, as we shall see, is the case for adult novice

programmers. Empirical studies of the program comprehension processes of children at various levels of computer programming experience will be essential for an understanding of this issue.

To take one example: In work at our laboratory with child Logo programmers (Kurland & Pea, 1983b), we have found that child novices frequently adopt a systematic but misguided conception of the manner in which control is passed[2] 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, control is passed back to the procedure which last called it. Children adopted 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 studies of Logo programming development (Pea, Hawkins & Sheingold, 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 they wrote reached but a moderate level of sophistication after a year's work and approximately 30 hours of on-line programming experience. We found that 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 and rote in character. To take one example: A child who had written a procedure using REPEAT which repeatedly printed her name on the screen was unable to recognize the efficiency of using the REPEAT command to draw a square. Instead, the child redundantly wrote the same line-drawing procedure four times in succession.

Programming Environment as Context for Development of Programming Skills

Much has been made recently in educational journals as well as in the popular press about which programming language is best for children to learn (Harvey, 1982; Tinker, 1982). Arguments for the relative benefits of one language over another have raged for years among computer scientists, though recent reviews indicate that very little empirical research of any quality has been conducted on this issue (Brooks, 1980; Moher & Schneider, 1982; Sheil, 1980). Operating systems, which are programs usually supplied with the computer by the manufacturer that define how a user interacts with the language

and how the language interacts with the hardware, are often ignored in arguments over the merits of different programming languages. Since the ease or difficulty of learning a language, especially for the novice, depends on the programming environment supported by the operating system as much as on the structure of the language itself, it is important to be clear on what exactly is meant by an operating system and a programming environment.

The operating system mediates all the activities of the computer. It permits the user to call up a language, and provides the link between programming commands and the hardware (disk drives, printers, tape units, etc.). The operating system may also support an editor through which the program can be entered into the computer, the debugging tools, the error trapping routines, the user subroutine libraries, the file management subsystems, and on-line help files. Depending on the sophistication of the operating system, a wide range of other programming aids and application packages, which constitute the programming environment, can also be made available. For example, UCSD-PASCAL, as it is implemented on most microcomputers, consists of a language--that is, a set of command statements which can be strung together according to specific syntactic and procedural rules--and an operating system (the UCSD P-system) which supports the editing, debugging, compiling, and filing of programs written in PASCAL (or FORTRAN, or assembler), as well as the linking of programs to extensive libraries of user routines. In a PASCAL programming environment one can also call upon numerous application packages and programming aids to help find program errors or to speed up the writing or editing of programs.

This distinction between a language per se and the programming environment in which one works with the language is critical. Many of the alleged advantages of Logo over BASIC as a language for children (Harvey, 1982) actually lie in the sophistication of Logo's programming environment. For example, compared to the standard BASIC that runs on an Apple computer, Apple Logo has a much more sophisticated editor, more detailed error messages, better debugging procedures, and provides more straightforward support for creating and manipulating files. However, a BASIC programming environment can be created on the Apple which provides the BASIC programmer with many of the advantages of the Logo environment. One can add to standard Apple BASIC a powerful program editor, debugging aids, libraries of useful routines which can be incorporated into any program, and even turtle graphic and recursive command structures of the type normally associated with Logo.

There is another important sense in which one language differs from another. Different languages are more or less well-suited for differ-

ent purposes. For example, SOLO, a language designed for students of cognitive psychology to study artificial intelligence, is much better adapted to this purpose than is BASIC (Eisenstadt, Laubsch & Kahney, 1981). However, writing a number-crunching program, a simple task in BASIC, would be quite difficult in SOLO. To say that SOLO is a better language than BASIC (or vice versa) without reference to the computer programming domain would be misleading and meaningless. Similarly, one characteristic of expert programmers (Kurland & Pea, in preparation) is they know many different languages. For any particular programming job, they tend to select the language best suited for that application taking into account the specific machine and operating environment in which they will be working.

A review of existing psychological studies of computer programming by Sheil (1981b) has highlighted some marginal effects for a variety of different measures (such as ease of debugging and comprehension measures) when comparing different programming languages that vary in features such as structures for expressing flow of control (e.g., the infamous GOTOs; Dikstra, 1968, 1976). The lesson to be learned is that the specific language chosen probably does not make a big difference, at least for adults. Far more important in carrying the variance for computer programming expertise are issues such as the resources that are available in the programming environment, how programming instruction occurs, and the amount of program writing, reading, and debugging students engage in.

Instructional Environment

Just as the programming environment is a critical factor in determining the facility with which programmers can work with a language (and thus what demands that language will place on the learner), the instructional environment plays a key role in determining how successfully students will be able to take advantage of the programming environment. Clearly, programming is not a black box into which students can simply be plunked. Many learners, children included, will need carefully sequenced instruction in how to use the operating system, how to combine programming lines into higher level units that accomplish some goal, how to select computer programming projects that are within their current capability, and how to think systematically about debugging and modifying programs.

The literature on teaching computer programming and designing computer programming tutors (e.g., Miller, 1974) supports the theory that deciding how best to introduce computer programming to students

and to assist them in writing programs is seriously hampered by a paucity of relevant pedagogical theory.

The question of how much, if any, instruction is best when introducing children to computer programming is hotly debated (e.g., Howe, 1981; Papert, 1980). At one extreme, it is possible to find schools in which computer programming (particularly in COBOL or BASIC) is taught like any other academic subject. Students have textbooks and workbooks, take tests on the definitions of various commands, and are expected to master a given set of computer programming constructs and to be able to write a given set of sample programs. At the other extreme, some teachers provide almost no direct instruction, encouraging children to explore possibilities, experiment, and create their own problems to solve. This "enlightenment" idea, popularized for programming by Papert (1980) in his book Mindstorms, holds that minimal overt instruction is necessary if the programming language is sufficiently engaging and simple to use, while at the same time being powerful enough for children to do exciting things. Though this view is not universally shared, even by devotees of Logo (Howe, 1981), it has had profound influence in the educational community.

As a result of a year of observation and research with children in two Logo classrooms (8- and 9-year-olds, 11- and-12 year-olds) in the Bank Street School for Children of the Bank Street College of Education, the importance of instructional context in promoting the development of computer programming skills has become apparent. Classroom teachers decided to adopt the instructional strategy advocated by Papert (1980), who suggests that children be allowed to explore the programming domain freely, with minimal intrusion or organized instruction. Thus, at the outset, the children were taught basic Logo commands and then encouraged to develop their own goals and projects. Each child was assigned two 45-minute work periods each week on the computers, but could add time when the computers were free or if other required classroom work had been completed. Teachers provided assistance with these individual or collaborative efforts, but offered minimal organized group instruction in new programming concepts and tools. Children learned about new concepts and programming possibilities as the need arose, and occasionally teachers organized small groups to introduce an interesting concept or possibility. There were no required computer programming assignments and no assessments of computer programming skill. During the year, the children who were interested learned enough computer programming to write simple procedures and perform routine housekeeping and editing functions. Only a few children, however, developed considerable facility with the programming language and routinely incorporated the difficult concepts of recursion, conditionals, and variables into their computer programming projects.

By year's end, both teachers were dissatisfied with the levels of computer programming skill in these classrooms and decided to use a more structured instructional context for teaching computer programming the next year. They later included computer programming as an integral part of their school curriculum, introduced a structured sequence of concepts and programming tools, and provided children with assignments, guided projects, and programming tasks. It is interesting to note in this context that Piaget himself criticized supposed "implementations" of his activity-based approach to cognitive development in educational settings because they lacked sufficient structure (Kamii, 1974).

There was considerable variability in the degree of interest individual children expressed in computer programming (Pea et al., 1983). Classroom observations, amount of time spent on the computer, and teachers' reports converged to indicate that about 25% of the 25 children in each class were highly interested in learning to program and spent correspondingly more time than other children on the computers. Approximately another 25% of each class expressed very little interest in computer programming and spent almost no time working at the computers after the initial introductory period. The remaining 50% were modestly interested, but often adjusted the time spent with computers to other interests that they had throughout the year. So, for example, if a child was interested in a research topic or writing assignment during a particular week, she might choose to focus on that work and sacrifice her time available for computer programming.

These findings of the need for instructional guidance for computer programming development receive support from extensive Logo instruction work by duBoulay and O'Shea (1976, 1978) in which typical conceptual problems that arise in teaching Logo programming are approached through a highly detailed teaching strategy consisting of 33 ordered worksheets for introducing computational ideas, problem-solving tactics, and debugging skill. Such carefully planned sequences of instruction may be important to ensure that computer programming schema knowledge is not "welded" (Shif, 1969) or "rigid" (Werner, 1957) with respect to its contexts of occurrence.

On a related point, 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 many technical details of that programming language. With the aid of the conceptual model, he argues, learners are able to assimilate the details of the programming language to the model rather than needing to induce the model from the details. Moran (1981) makes the impor-

tant point that Mayer's models may work because they are "synthetic constructions" composed of recognizable parts, such as "memory cells and procedural agents," in contrast to more global analogies ("the text editor is like a typewriter") which often don't work.

Accounts of Levels/Stages of Programming Skill Development

Observations of levels of computer programming skill development have been extremely general and more rationally than empirically derived as, for example, was our four-level model presented earlier in this report. Neither of the accounts to be described below delineates qualitatively distinct intermediate levels of performance in terms of the four kinds of cognitive subtasks involved in programming discussed above. For example, in a study of 12-year-old children learning to program in Logo, Howe (1980) describes three distinct overlapping "stages": (1) "product-oriented": aiming to produce effects without concern for how they are achieved; (2) "style-conscious": aiming to work in what is perceived as "correct Logo programming style," derived from worksheets that recommend specific programming, debugging, and editing methods; and (3) "creative problem solving": use of computer programming resources for analytic problem-solving activities; search for useful procedures from others to solve one's own problems; and possible production of plans, diagrams, or written problem specification.

No data are reported for the number of children successfully reaching the various stages, nor is it clear that one could at any point in time reliably identify children performing at any of these stages as defined. Further, the second stage appears to be an artifact of the way that Logo programming was taught.

Hoc (1977) has provided a related account of three general steps in the construction of a mental representation of a computational device language (COBOL) found for adult novice programmers. Such steps consisted of progressive "internalizations" of the constructs of the programming language.

What Does "Becoming a Programmer" Mean? Conceptual Difficulties of Adult Programmers: Some Initial Barriers

In contrast to highly skilled programmers, as many adults learn to program they reveal deep misunderstandings of computer programming concepts and 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). Misunderstandings such as

assuming all variables to be global (when some may be specific to one program), and expecting that observing one pass through a loop allows one to predict what will happen on all subsequent passes (when in fact the outputs of programming statements which test for certain conditions may change what will happen during any specific loop), were common for college adults after finishing a PASCAL programming course (Jeffries, 1982). Furthermore, Soloway, Bonar and Ehrlich (1983) have shown that error-ridden programs resulting from "buggy" concepts of looping constructs in PASCAL may be mitigated if the programmer's spontaneous cognitive strategy for solving looping programming problems is supported by an appropriate programming language construct. Subjects were better able to write correct looping programs when they used a READ/PROCESS strategy (on the i^{th} pass through a loop the i^{th} element is both read and processed) rather than a PROCESS/READ strategy (on the i^{th} pass the i^{th} element is processed and the next-i^{th} element is read). Research by Mayer (1976), Miller (1974), and Sime et al. (1977) also reveals that adult novice programmers have a difficult time with the flow of control concepts expressed by conditionals (for a review of these findings, see duBoulay, O'Shea & Monk, 1981). We expect that the roots of these misunderstandings may be based on inappropriate transfer from non-CP domains.

Since members of the educational community would probably assume that adult programmers are not beleaguered by conceptual problems in their programming efforts, we must drive this point home. Once we recognize that programming by "intellectually mature" adults is not characterized by error-free, routine performances (Anderson, 1983; Anderson, Farrell & Sauers, 1983), one might ask what should be expected of the child learning to program who devotes but a small percentage of his or her time in school to learning to program. In fact, the conceptual difficulties of adult programmers have been lamented by such computer programming polyglots and visionaries as Minsky (1970) and Floyd (1979) as due to what is all too frequently taught as computer programming. Too much focus, they urge, is placed on low-level form such as grammar, semantic rules, and some preestablished algorithms for solving classes of problems, while the pragmatics[1] of program design are left for students to discover for themselves.

Summary

Becoming a successful programmer is very hard. After many thousands of hours, programming does not become a routine task. One apparent reason for this is that program development, as constituted by its four major cognitive subtasks, involves the transformation of

ill-defined problems into well-defined ones (Greeno, 1976; Simon, 1973?), and then solving them. In conclusion, we note that the literature we have just reviewed in this section is not helpful in addressing the "becoming a programmer" question in ways that inform our demands analysis. Before turning to a consideration of what would constitute the "core" of programming knowledge, it is useful at this juncture to reflect briefly on some of the popular conceptions of the knowledge required for computer programming.

On the Cognitive Demands of Programming

Popular Conceptions of What Knowledge Computer Programming Requires

Is there any more to be be gained in our quest for the cognitive demands of programming subtasks from the popular literature on computer programming? We think not. Current knowledge about the prerequisite mental abilities, background knowledge, reasoning skills, and cognitive style that allow for the development of high-level computer programming skills in precollege-age students is entirely anecdotal, and highly vulnerable to the ideological biases that observers have brought to their observations. Often, the children who learn computer programming are preselected. The popular media, computers-in-schools magazines, and computer programming teachers have cited levels of mathematical problem solving ability, mechanical aptitude, and reasoning skills as key determinants of progress in learning computer programming. In earlier years, and to some extent even today, the prevalent attitude was that programming required highly developed mathematical abilities. But early programming aptitude test developers recognized that "mathematical knowledge was associated with the [specific type of] application and not with programming ability" (McNamara, 1967). This view is quite pervasive in newspapers and magazines today, and is a frequent source of fear for computer novices or those considering careers in computing, particularly women (Swaine, 1983). In biographical accounts of computer technicians and software developers at the frontier of their discipline (e.g., Kidder, 1981), the profiles which emerge do seem to support the popular impression of computer programming as linked to high mechanical and mathematical ability. And in the software engineering literature, there is a widespread belief that "we can be fairly sure that intelligence and educational differences will be reflected in performance levels in software experimentation" (Moher & Schneider, 1982, p. 74). There are also considerable anecdotal reports to the effect that children and adults who become serious programmers share, from an early age, a strong interest in figuring out how things work, solving puzzles, and searching through challenging problem spaces (Kidder, 1981).

Such observations as the above, however, are seriously compromised by the well-known propensity for humans to focus on the positive confirming cases in their search for correlations of real-world attributes, in this case a high level of some ability and a high level of computer programming mastery (Inhelder & Piaget, 1958; Nisbett & Ross, 1980; Shweder, 1977; Ward & Jenkins, 1965). Such systematic distortions and selective attention to aspects of the data from which correlations are derived calls for the scientific study of demand characteristics for level of computer programming development in nonpreselected school populations. Furthermore, the oversimplifications involved in these popular claims are notorious. They are largely uninformative, because, as we shall demonstrate in the next section, there are many different types of programming. And one may ask "what are the demands of that kind of programming" for each type, as well as for the cognitive subtasks within each type. Even more specifically, one may ask: "What are the demands of doing programming subtask x in programming type y?" As we shall see, asking what programming is raises many complex issues, much as asking what writing is, in which questions about demands are invariably linked to asking about writing genres, or different types of writing (Bereiter, 1979; Olson, Mack & Dufy, 1981). This skeptical stance must nonetheless be accompanied by a positive account of the potential cognitive demands of specific activities involved in computer programming. What is ultimately at issue is the empirical viability of such a priori considerations about "demands," at a level of analysis in terms of the cognitive subtasks of programming. We will return to this issue in the section on measuring program aptitude in adults.

Defining a "Core" of Programming Knowledge

What is the minimum or "core" of programming abilities that people should have, and what prerequisite knowledge and ability are required to attain that core? What are the goals of programming instruction that are appropriate? Many have recently been asking these questions, especially since much of society is mystified by computers, yet feel the need for developing some degree of computer literacy, however defined.

In a recent paper, Norman (1983) has distinguished four levels of computer literacy not unlike the four levels we have recently offered as fundamental (Pea & Kurland, 1983a). The first level involves knowing about general principles of computation; the second, how to use computers; the third, how to program computers; and the fourth, how to understand the science of computers. He characterized the first level as a necessary curriculum for everyone in society, and the list of concepts to be covered as:

- software versus hardware
- computer architecture: central processing unit, forms of memory
- terminal, peripherals, microprocessor
- algorithms: how they are developed and applied
- procedures and programming: what they are, how done
- machine intelligence
- communication networks
- data bases: how organized and used
- limitations of computers
- distributed offices
- multiprocessing
- computer security, as in time sharing

His second level was described as useful but not necessary for everyone and, like this report, he characterized level three, or programming, as very difficult and requiring many years of intensive study. Computer science is limited to the small minority who have sufficient interest and skill.

From our perspective, the best way to learn about algorithms, procedures, and programming, even for level one, is to have hands-on experience. We take the position that some core of programming ability is essential for this first level, not only "knowledge about" programming, although Norman's point about the great difficulties embedded in serious programming is well taken, as the discussion of cognitive subtasks involved in expert programming makes clear. Sheil (1981a), for example, argues quite convincingly that the increasingly widespread use of complex programmable devices, such as office information systems and calculators, "has made a basic appreciation of the nature of programming a modern survival skill" (Sheil, 1980, p. 1). What remains is to define what constitutes that basic appreciation. To answer these seminal questions about minimal programming literacy and its cognitive foundations, we will require a great deal of empirical research, since it is essentially a question of asking what is the core of natural language that an individual should learn in school in order to be judged as a competent member of the language community. Part of the answer will be provided by seeing how individuals with varying skill levels in programming are able to cope with the technological complexities of today's society. On the ideological side, Luehrmann (1981a, 1981b) has been an outspoken advo-

cate of the need for hands-on programming ability, as has Sheil; our sympathies lie with these manifestations of the well-established Deweyan pedagogy of "learning through doing" that has been rediscovered by modern cognitive science (e.g., J.R. Anderson, 1981; Anzai & Simon, 1979).

The list of core concepts that repeatedly turns up in discussions with computer programming professionals and computer programming instructors includes understanding the temporal logic of sequential instructions, control structures, data structures, variables, and procedurality. For example, Ralston and Shaw (1980) mention some important implications of the rapidly evolving nature of computer science for thinking about what students should learn, even though their prescriptions are meant to be applicable to the computer science major in college:

> Specific skills learned today will really become obsolete. The principles that underlie these skills, however, will continue to be relevant. Only by giving the student a firm grounding in these principles can he or she be protected from galloping obsolescence. Even a student who aspires only to be a programmer needs more than just programming skills. He or she needs to understand issues of design, of the capability and potential of software, hardware, and theory, and of algorithms and information organization in general. (p. 67)

Although many of these concerns are particular to individuals who are pursuing careers as computer scientists, the point of their argument is clear: general principles, not specifically tied to any single "best" programming language or programming machine (see Floyd, 1979), should be our educational goal for establishing a basic programming literacy.

More fundamental programming concepts, as we have seen in our earlier discussions of the cognitive subtasks of programming, are those necessary for: (1) problem understanding, such as data classes, types of data processing operations, procedurality, the temporal logic of sequential instructions (Galanter, 1983), and program functions; (2) designing and planning the program, such as "program schemas" of various types, "symbolic execution" of plans, and "debugging"; (3) program coding, such as syntactic rules for program code, and programming language "primitives"; and (4) program comprehension and debugging, such as program functions, procedures, and the concepts necessary for problem understanding.

Since the adult literature is of little assistance in determining what type and level of a learner's cognitive characteristics may influence the developmental course of computer programming learning, how might one identify the factors that influence whether or not a child will benefit from programming experience or develop proficiency in computer programming? Sheil (1980) suggests that many of the rules one needs to learn for computer programming are minor variants on already mastered skills from other domains, such as the quasi-procedural[3] knowledge of how to give/follow task instructions, directions and recipes, and the semantics of conditionals, temporal ordering, and causality. Sheil argues that one would expect such background knowledge to provide fundamental enabling skills which programming students apply by analogy in learning computer programming. Whether knowledge of this kind is necessary to learn to program is unlikely; it may however help facilitate learning the concepts of procedurality and sequentiality as children learn to program.

Defining the Cognitive Prerequisites for Learning Computer Programming: Where to Begin

What are some of the most plausible a priori candidates for cognitive demands of programming? In asking this question, we find ourselves again confronted with the question of what learning computer programming means. Thus far, we have been able to describe some of the component mental abilities of advanced programming skill in adults. This issue of expertise in computer programming is central to any discussion of demands, since we must always ask: "Demands in order to do what?"

While no research has been directly aimed at defining the cognitive prerequisites for learning computer programming (it has asked who will do it better, predictively, rather than who can do it at all), at least five factors have been mentioned frequently: (1) mathematical ability, (2) memory capacity, (3) analogical reasoning skills, (4) conditional reasoning skills, and (5) procedural thinking skills. These cognitive abilities are presumed to have an impact on or to mediate computer programming learning in a number of ways.

1) Mathematical ability. In addition to general intelligence, it has frequently been suggested that computer programming skill is linked to general mathematical ability. Historically, computers were developed to aid in the solution of difficult mathematical problems. Despite the fact that today many computer uses have very little to do with mathematics (e.g., database management, word processing, games, graphic design), the notion has persisted that to work with computers one should be mathematically sophisticated. Media accounts

of children using computers in schools have perpetuated the common belief that computer programming is the province of math whizzes.

To our knowledge, there is no evidence that any relationship exists between general math ability and computer programming skill, once general ability has been factored out. For example, in some of our own work we found that better Logo programmers were also high math achievers. However, these children also had generally high scores in English, social studies, and their other academic subjects as well. Thus, attributing their high performance in computer programming to their math ability ignores the relationship between math ability and general intelligence. Nonetheless, general math ability per se cannot yet be ruled out as a possible prerequisite to the successful mastery of computer programming skills.

2) Memory capacity. It is frequently observed that computer programming is a memory-intensive enterprise that requires great concentration and the ability to juggle values of a number of parameters at one time. Thus, individual differences in processing capacity are likely to influence who becomes a good programmer, on the hypothesis that small spans make programming too effortful. Traditional forward and backward span tasks, as well as the more recently developed transformational span measures (see Case & Kurland 1980; Case, Kurland & Goldberg, 1982), are frequently cited as indexing processes basic to learning. These demand measures assess how much information a student can coordinate at a given moment. For example, the Counting-Span Task (Case, Kurland & Goldberg, 1982) requires students to perform a sequence of cognitive operations (counting, in this case) while retaining the products of each individual operation for later recall. Tasks such as this have been shown to correlate reliably with general intelligence, Piagetian developmental level, and ability to learn and use problem-solving strategies (e.g., Hunt, 1978). They have not thus far been utilized in research on computer programming skill development.

3) Analogical reasoning skills. It is conceivable that an individual may have relevant background knowledge and capacities, yet neither connect them to the computer programming domain, nor make the transfer connections from computer programming to other domains during or after the learning of computer programming. This issue of the retrievability or "access" of knowledge is absolutely fundamental to learning and problem solving throughout the course of life (e.g., Brown, 1982), and the general role of analogy in problem solving and interpretative processes is widely recognized (Miller, 1979). These critical transfers by analogy of knowledge and strategies, both "into" and "out of" the learning of computer programming depend to some

extent on the availability of general analogical thinking skills. Tasks designed to measure ability for engaging in analogical thinking (e.g., Sternberg & Rifkin, 1979) or used in cognitive studies of analogical thinking (e.g., Gick & Holyoak, 1980, 1983; Holyoak, 1983) may serve as one key predictor of level of computer programming development and the quality and extent of transfer outcomes to be expected.

In one specific application of the uses of analogical thinking for learning computer programming, Mayer (1975, 1981) has argued that students commonly learn computer programming by comparing the flow of control[2] intrinsic to computational devices to that of physico-mechanical models they already possess. And duBoulay and O'Shea (1976, 1978) have successfully used extensive analogical modeling as a means of explaining computer functioning to beginning 12-year-old Logo programming students.

4) Conditional reasoning skills. A major component in computer programming, once past the elementary stages, is being able to handle conditional statements. Conditional commands underlie the operation of loops, tests, input checking, and a host of other programming functions. It is reasonable, therefore, to assume 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 control flow and data flow through conditional statements.

5) Procedural thinking skills. There are several kinds of quasi-procedural, everyday thought which may have a direct bearing on the facility with which a learner masters the "flow of control" procedural metaphor that is central to understanding computer programming.[1] Those which have been suggested include giving and following complex instructions (as in building a model), writing or following recipes, and concocting or carrying out directions for travel. Presumably, individuals who have a greater familiarity with these linear procedures that are analogous to the flow of control for commands in a computer program will more readily come to grips with the "procedural thinking" touted as a central component of computer programming expertise (Papert, 1980; Sheil, 1980).

Types of Programming

It is widely recognized within the computer science community that there are different types of programming. By the 1960s, McNamara, one of the developers of the widely used IBM Programmer Aptitude Test, had recognized the need to develop new tests which would

guide the selection of different types of programming jobs: "for instance, systems programming, applications programming, diagnostic programming, or program maintenance" (McNamara, 1967, p. 56). People are recruited for careers in industry, the military, for software houses, for educational companies, for videogame companies, not as programmers per se, but to do programming of a specific kind. We have indicated the importance of cognitive theories of what expert programmers do in conceptualizing the problems of the cognitive demands of programming. This point regarding the differentiation of types of programming activity in the adult programming community adds yet another layer of complexity to the problem, one which is not explicitly recognized even in the cognitive science literature on programming. Although it is acknowledged in that literature that "the programmer knowledge base" is an integral aspect of programming performance, studies so far have been concerned with problems that minimize the effect of domain-specific knowledge on programming (e.g., Jeffries, Turner, Polson & Atwood, 1981). Yet for the child learning to program, specific content problems will be worked on as she learns to program; thus, we know correspondingly little about the new developmental concerns raised.

Shneiderman (1980, p. 40) indicates some of these complexities in his discussion of types of programmers. He first distinguishes between three general types of programmers based on extent of effort expended in programming activity: (1) professional programmers; (2) occasional programmers; and (3) programmer hobbyists. Under the category of professional programmers, he then distinguishes "[1.1] systems programmers [who] work on operating systems, compilers or utilities that are used by [1.2] application programmers, who solve user problems." As examples of application programs, he lists banking, reservations, payroll, personnel management, accounts receivable, data collection, statistical analysis, inventory, and management reporting systems. Other types of applications for the professional programmer are videogame programs, educational programs and simulations, and military application programs such as battle simulations. For occasional programmers, specialties also emerge; those Shneiderman lists are: (2.1) scientific research, (2.2) engineering development, (2.3) marketing research, and (2.4) business applications (such as the types of applications programs listed above). For the hobbyist, types of programs may be of great diversity, such as small business application programs, home financial management programs, home inventory programs, and even music composition.

We may continue this differentiation process of asking about different types of programming by turning to hiring ads for programmers, in

which we expect tacit theories of programming specialization to be made manifest. In the "Help Wanted" section of The New York Times for March 28, 1983, for example, ads for programmers emphasized either: (1) type of programming by content area (systems software; applications development: actuarial, banking and investment, telecommunications, database, financial, graphics, military, payroll/accounting, robotics, scientific/engineering/R & D); (2) type of programming by programming language (e.g., Assembly, APL, Bal, BASIC Plus, Unix-C, CICS, COBOL, Dibol, DOS/ VS, FORTRAN, IMS, Macro, PASCAL, PL-1, RPG); (3) type of programming in terms of specific type(s) of computer programming environment(s) (e.g., Burroughs, DEC PDP-11, HP3000, IBM Systems 38, IBM 360/370, IBM 3033, microcomputers, Prime, VAX 780, Wang); (4) a combination of (2) and (3), e.g., PASCAL on microcomputers; or (5) a combination of (1), (2), and (3) e.g., FORTRAN/COBOL actuarial background with IBM experience desired. In one case, the "demands" question was even directly addressed: "[For] securities industry, we need strong math foundation including heavy calculus, probability theory and differential equations to provide financial research information to fixed income and other types of securities traders."

Different Evaluative Criteria for Different Kinds of Programming

There is no one evaluative scheme for assessing programming competency. Evaluation of such competency is necessarily different for public (i.e., business, scientific, or game programming), personal, and educational programming (in which clarity of exposition rather than utility or pragmatic value is important). In industry/ business, the emphasis is on rapid programming; in games, compact code; in some educational software, developmental structure of materials and problems. Personal programming may be workable, buggy, undocumented, and unreliable for some inputs which might be entered if other people were to use the program. For scientific programming, the main criteria are speed of code execution, provability, and accuracy. For videogame programming, the criteria are: "packing more power into smaller spaces. Increasing resolution. Controlling cost. Maximizing quality" (Parker Brothers advertisement for videogame designers). For public programming, the need is for brevity, and comprehensibility to a general audience (Shneiderman, 1980). With new types of programming and new markets for programs continually being created, the demands will evolve accordingly, as will the software quality metrics necessary to evaluate them.

Rich Versus Impoverished Programming Environments for Children

Educational programming environments for precollege-age individuals, are often impoverished, in the sense that the kinds of tool and utility

programs used by mature programmers to develop their programs are "not allowed" for the classroom. Often, teachers do not want children looking at programming language manuals where example programs are listed. Why is this? Although we have not seen this point of view made explicit in programming curricular materials or teacher training for programming instruction, we have heard teachers say that this emphasis is due to the teacher's belief that children must think through the chunks of programming operations that are "givens" in the tools and utilities. It is as if children will never be able to analyze these tools and fully understand them unless they themselves participate in their development. There is a corresponding fear of rote program use. This attitude might be viewed by some as analogous to a current controversy about the use of hand calculators by children: will using them for addition, multiplication, and so on make the children incapable of doing these arithmetical operations without a calculator? But the connection between this example and programming tools is weak. Without taking a position on the calculator example, we may note that how programming tools and utility programs are used may be determined by the teacher; the teacher may have children explain how the programs work, and learn whatever programming concepts are necessary for unpacking the function and structure of the constituent parts of the program. We believe that, just as children can learn important lessons by reading great literature, and even rewriting it with different literary purposes in mind, so may they gain programming knowledge and programming skill through having at their disposal the powerful programming tool kits and utility programs that aid the program development process.

We would apply the same point to the use of algorithms in programming by children. Adults are not expected to discover all the programming tricks and algorithms, but must learn them. The same should apply to programming instruction for children.

Individual Versus Team Program Development

It is naive to think that the way to assess expertise is to put people alone in a room and require them to generate in a given length of time a program that conforms to certain specifications. Not only is the relationship between rate of programming and program quality not well understood (Brooks, 1982), but requiring a person to program with no access to manuals or other people's help does not reflect the way good programmers do their work. In many work settings, programs are being written by design teams made up of program designers, managers, and coders working with formative researchers in a collaborative environment. It has been recognized that one person

may not be decentered enough to design, implement, and verify a program. The collaborative nature of today's programming bespeaks a similar orientation in programming instruction. Perhaps the best way to teach children to program is as apprentices, rather than as solitary students learning facts. One consequence of this view is that the usual question: "What are the demands for learning programming?" changes to: "What must one know to be a good member of a design team?" This may be the question to ask rather than: "What does one need to know to be able to design, implement, document, and verify a program all by oneself?" Learning to be explicit about how to design a program, and the details one needs to give to the implementers may be a much more important goal of teaching computer programming than arranging instruction so that each child can write an entire program by himself. In fact, with the recent advent of menu-driven automatic program-generating programs (such as The Last One for BASIC or Quickcode for generating dBase programs), the day may be approaching when "programming" will mean specifying the design of the program. This characterization of the program creation process, however, still leaves open the questions of the point at which a division of labor is necessary or desirable, and the point at which the emphasis can be switched from implementation to design.

How Do Data on Programmer Aptitudes and Abilities Help Us?

The standard multivariate approach to the cognitive demands of programming is to study aptitude/treatment interactions. The studies we will review almost always ask what cognitive profile it "takes" to become a programmer, or what previous specific or general mental abilities are required for subsequent success in learning to program.

It is of some importance to remark briefly on the historical foundations of research on "computer programming aptitudes," of which several hundred studies exist (see reviews in Ricardo, 1983; Schmidt et al., 1979). During the early 1950s, tests were developed in order that companies who required programmers would have better selection criteria for computer trainees than they would through interviews and work history alone. Not until years later were these tests and others further developed in order to select individuals for programming education at the college level (McNamara, 1967). The extension of the predictive validity of programming aptitude tests to actual programming job performance, rather than success in an industrial training course or a college programming course, has always been problematical (e.g., Bell, 1976).

One expecting answers from closely inspecting this multivariate literature about the "demands of learning to program," or from our goal-

indexed version of this question discussed earlier, will be disappointed. The tests were commonly developed by interviewing programming managers about the skills that good programmers seemed to have (McNamara, 1967; McNamara & Hughes, 1961), and then looking for test items that appeared to test for that ability (e.g., "logical reasoning" or "mathematical reasoning") in existing intelligence tests. Although most studies in which programming aptitude test scores correlated significantly with programming "success" (generally indicated by grades in industrial programming training courses or college programming courses) observed that "general intelligence" (when test scores were available) also correlated very highly with programming success, this does not seem to have moved the researchers to go further and ask whether the "programming aptitude" supposedly linked to programming skill constituted a specific aptitude factor above and beyond "general intelligence." We suspect that it may not. In fact, one survey (Mayer & Stalnaker, 1968) revealed that many companies use intelligence tests as their predictors of programmer success. In a general review of the computer personnel research presented to ACM Special Interest Group in Computer Personnel Research, Stalnaker says: "I think that if we have to have a very concise summary of our current knowledge, it is that the more intelligent person you can find, the better programmer you can probably get." If these observations are valid, the lesson from the many studies of predictive validity of programming aptitude tests is not so profound--those who do well on school-based intelligence tests are also likely to do well on programming aptitude tests. In any case, and more importantly, the findings of this multivariate literature are tangential to the questions we have posited as the central ones, that is, those pertaining to whether and how specific thinking skills identifiable in the practice of expert programmers are manifested in individuals during the process of learning to program.

It is interesting to note that close observations of the actual practice of programming, and the sequence of mental tasks required during program development was rarely undertaken. Even the most differentiated analysis of "what programmers do" (Berger, 1967) results in categories that have not really been unpacked in terms of their cognitive subtasks. The outcome of Berger's analysis was the following list of the seventeen major interpretable factors of programming and systems analysis:

PROGRAM PRODUCTION

1. General programming operations
2. Debugging
3. Programming real time systems

4. Lead programming responsibilities
5. Program production for special purpose computers
6. Program production planning and scheduling
7. Program production supervision
8. Utility program development (executives and compilers)
9. Utility program development (general purpose and library)
10. Program diagramming and testing

PROGRAM ANALYSIS AND DESIGN

11. Program system analysis (business and logistics)
12. Program system analysis and design

PROGRAMMING'S PERIPHERAL TASKS

13. Program system integration
14. Program system testing
15. Program installation or modification consulting
16. Program documentation
17. Training

On Measuring Programmer Aptitude in Adults

Two important recent studies that explode the myth of the nonbusiness student "who can't program" have been carried out by Ledbetter (1975) and Lemos (1981). In each case, when careful comparisons were made between large groups of business and nonbusiness college majors, no significant differences were found in programming language learning capability between the groups. This is a significant finding because in many universities, nonbusiness majors are assumed not to have the "necessary background skills" for learning computer programming and are taught about it rather than being given hands-on experience.

Probably the best known test instrument used to measure aptitude for programming is the PAT (Programmer's Aptitude Test), developed by IBM during the 1950s to select programmer trainees. We will briefly review this and other measures, and the dominant findings of modest correlations between scores on these aptitude batteries and various programming skill outcome measures, such as success in a first college programming course or a company's programming training program. But a central problem with such measures was noted by Weinberg (1971) some time ago in his pioneering book on the psychology of programming, in reference to the PAT: "Admittedly, it does predict scores in programming courses fairly well, but that is not what we are buying when we hire a programmer" (p. 173).

Many measures of programming aptitude have been developed for which independent validation reports are unavailable. Among the measures studies most intensively and used during the last three decades, the most prominent are:

Programmer Aptitude Test (PAT). IBM developed three different versions of the Programmer Aptitude Test (PAT) between 1955 and 1965, all of which have been out of print since 1973. The final version consisted of letter series, figure analogies, and arithmetical reasoning subparts.

Computer Programmer Aptitude Battery (CPAB). The CPAB, developed by Science Research Associates, consists of verbal meaning, reasoning, letter series, number ability, and diagramming subparts.

The Wolfe Programming Aptitude Tests (WPAT). (1) Aptitude Assessment Battery: Programming; (2) Programming Aptitude Test: School Edition. "The test measures...accuracy, deductive ability, reading comprehension of a complicated and extended explanation of a kind found in programming reference manuals, ability to grasp new and difficult concepts from a written explanation, and ability to reason with symbols" (Wolfe, 1969, p. 67).

For a thorough review of validation studies carried out on these different programming aptitude measures, see Ricardo (1983). A few examples will suffice for our purposes.

The most comprehensive and critical study examining computer programming aptitude tests and their predictive validity of which we are aware was carried out by Schmidt, Hunter, McKenzie and Muldrow (1979). Of the 161 studies they uncovered, 150 used the PAT and most of the others used the CPAB. Schmidt et al. outline two innovative Bayesian procedures for establishing validity generalization, and found that "the (multivariate) total PAT score validity is high for predicting performance of computer programmers and that this validity is essentially constant across situations." Their second method extends this conclusion for success in training programs as well.

In one extensive study, Ricardo (1983) used five factors (deductive reasoning, inductive reasoning, persistence, SAT verbal score, SAT math score) as a model for predicting success in a first programming course in college (in the programming language PL-1). She chose inductive reasoning as a target ability, since the PAT and CPAB validations generally found the figure analogies and (number or letter) series subtests most reliable in predicting programming success

(by the limited measures of "success" used). As McNamara (1967), one of the developers of the PAT, noted: "there appeared to be one basic requirement for a good programmer; that was analytic ability or logical reasoning ability" (p. 53). The reasoning for the use of this measure is typical of this genre of literature:

> The process of looking at items and discovering a rule is used in programming when a student must examine output received and predict future output. This is necessary when he has to determine where an error exists in an algorithm. The two methods of testing inductive reasoning which are used in this test [her Programming Readiness Test] are figure relationships and number series. In the figure relationships, the student is given two figures which are related in some way, and a third figure. He then has to select a fourth figure which preserves the relationship with the third. Some of the possible relationships between figures are the symmetries of translation, rotation, reflection, and glide reflection, variations in size, in number of sides and vertices, symmetry of part of a figure, shading of portions of a figure and decomposition of a figure. In number series, the student is given several numbers which are related by some rule, and he must select the next number. Some of the relationships which can be presented are arithmetic progression, geometric progression, addition of a variable whose value is itself a progression, alternating sets of terms of two different progressions, and power series.

Similarly, deductive reasoning is chosen since:

> The content area in programming which requires the application of deductive reasoning is decision making. This ability is needed in interpreting problem situations so that they may be expressed in algorithmic forms. Specifically, each condition of a problem must be reduced to a proposition, either simple or compound, which must be evaluated as true or false. Testing of algorithms requires the recognition of the processes of both positing and negating logical propositions. It is also essential that the student understand the applications of logical conjunction, disjunction, implication, and negation in interpreting the conditions of the problem. These applications are tested by the use of syllogisms. The meaning of existential and universal quantifiers, and the negation of statements using them, are also tested. (p. 66)

She found that all five of her variables were significant predictors of success in the semester-long programming course, in terms of either final exam grade or final grade (multiple R of .63 for final exam score and .56 using semester final grade). Like Mazlack (1980), she reported that year in school, sex, prior programming experience, and type of major correlated insignificantly or poorly with criteria of success such as test scores, final examination score, and final semester grade. Using all five factors a multiple regression analysis revealed a multiple R of .59 for final semester grade.

In many studies investigating the relationship between programming aptitude test scores and success in programming courses, college grade point average often correlated as highly with programming success measures (indicated by test, semester grades, or scores) as did the aptitude test scores (e.g., Bateman, 1973; Bauer, Mehrens & Vinsonhaler, 1968; Fowler & Glorfeld, 1981; Ledbetter, 1975; Mazlack, 1980; Newstead, 1975; Peterson & Howe, 1979; Stephens, Wileman & Konvalina, 1981). This raises some question about the generalizability of college measures of programming success to on-the-job programming success, since "college grade point averages of programmers have been shown to have no predictive value for programming performance" (Mayer & Stalnaker, 1968, p. 659).

Legal Aspects of Programming Aptitude Testing

It is of some interest that programming aptitude testing has come under substantial criticism developing out of the August 1966, Equal Employment Opportunity Commission (EEOC) Guidelines on Employment Testing Procedures, making all test users responsible for assuring that tests do not contain racial or cultural bias, and for ensuring current performance-related test score validity. The 1975 Supreme Court opinion in Albermarle Paper Co. vs. Moody now requires as law that preemployment tests be demonstrated as "predictive of or significantly correlated with important elements of work behavior which comprise or are relevant to the job or jobs for which candidates are being evaluated" (Ledvinka & Schoenfeldt, 1978). Karten (1982), in interviews carried out for an indepth article on Aptitude Tests for Computer World, found that the consequence of the law for consumers of the programming aptitude tests has been that "many of those closely involved with programmer trainee selection using aptitude tests are cautious about speaking for publication. Some fear involving their company in lawsuits and EEO actions" (p. 17).

"Interest Inventories" and Values of Computer Science Personnel

On a related tack, people have taken interest/value inventories of programmers and asked: "What are the interests and values of

programmers and students studying programming that may set them apart from nonprogrammers?" The logic of such a question is that if one has interests similar to those of persons successful in a specific occupation, then one will be more likely to enter that occupation and, under some construals, be more likely to succeed in it. In Weinberg's (1971) influential critique of the use of programmer aptitude batteries that are intelligence test-like, he offers his own view that "intelligence has less to do with the matter [of programming performance] than personality, work habits, and training" (p. 176). Others have asked if there is anything to the stereotype of the mechanically or mathematically oriented computer scientist who is introverted and has "lost value for humanity" (Zimbardo, 1980). Evidence on all these counts is weak. Perry and Cannon found that, although there are distinctive interests of programmers as indicated by scores for the Strong Vocational Interest Battery, the SVIB is not a useful predictor of performance in either programming training courses or in program production (reviewed by Mayer & Stalnaker, 1968, p. 665).

Adult Programmers: Problem-Solving Style

Testa (1973) used Witkin's measure of "perceptual style," the embedded figures task (EFT), to study programmer aptitude. The task involves a series of problems where the test taker must find a specific shape in the context of a larger set of other shapes, and the dependent measure is the time taken to locate the figure. Higher scores are viewed as an index of "field independent" cognitive style, lower scores as an index of "field dependent" cognitive style. Testa found a significant correlation between field independence and success in a college COBOL programming course as indicated by test grades. The explanation for this finding was that "programming requires an ability to perceive the whole and a concomitant ability to proceed from the general to the particular" (p. 50), and "clearly, the EFT must have tapped some of the characteristics of the programming task" (p. 52).

Cheney (1980) took a similar approach in hypothesizing a significant relationship between scores on a BASIC programming examination in a college course and "analytic" as opposed to "heuristic" cognitive style, as indexed by a questionnaire developed by Barkin (1974) and assumed to relate to the EFT's style categories of field independence/dependence. He found the scores on these two measures to be significantly correlated ($r = .82$). Cheney suggests that those scoring highly on the style measure, and presumed to be "analytic," may learn programming best by progressing at their own rate on programming projects, but that the "heuristic" thinkers may require more structured and formal teaching to understand how to program.

In each of these studies, the aim was to develop aptitude measures that would predict success in a programming course distinct from "mathematically oriented tests." Although they were successful in this limited goal, we do not yet know how cognitive style may be related to programming skill, and the studies do not bear on the more important question of how cognitive style may contribute to performance on specific programming subtasks. They do offer the suggestion, however, that cognitive style interacts with how one is taught programming, not with whether one can learn to program.

"Developmental Level" and the Learnability of Programming

Beyond asking what general cognitive characteristics may be prerequisite to or mediate a child's learning of computer programming, a more specific question has been raised by many: What "developmental level" may be required to benefit from computer programming experience? In the educational community, the question is more commonly phrased as: How old do children "have to be" before learning computer programming? The general concept of "developmental level" at the abstract theoretical levels of preoperational, concrete operational, and formal operational intellectual functioning has proved to be a useful one for developmental and instructional psychology in understanding children's ability to benefit from certain types of learning experiences (e.g., Inhelder, Sinclair & Bovet, 1974). However, the very generality of these stage descriptions is not readily applicable to the task of understanding the development of specific domains of knowledge such as computer programming skill.

In light of the lack of research on the development of computer programming knowledge and strategies, two reasons lead us to reject a formulation of the computer skill problem in terms of the concept of Piagetian "developmental level" as inappropriate (Favaro, 1983). First, there is considerable evidence that the development and display of Piagetian-defined logical abilities is importantly tied to content domain (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 what relationships might inhere between computer programming experience and performance on Piagetian tasks.

Our second concern with a Piagetian "developmental level" formulation applied to computer programming skill development is that the task of learning to program has not thus far been subjected to developmental analysis or characterized in terms of its component skills, except

insofar as we have reviewed earlier in the section on programming as a cognitive activity. We thus reject a general developmental level formulation as useful for articulating the cognitive demands of specific programming skill development, and embrace in its stead an approach that is more concept-based.

Conceptual Development and Programming

In the literature on child language development, one may find extensive discussions of the concept of "cognitive prerequisites." In that discipline, this concept has been developed as a means of exploring the relationships between language development and cognition; in our context, the relationships between programming development and cognition. Although many issues are quite distinct for these two disciplines, the sense of cognitive prerequisites used in language developmental studies is illuminating. The fundamental idea is that one may ask what mental resources are required by a particular kind of linguistic activity, such as particular spatial concepts and the use or understanding of such terms as "in," "on," and "under" or, more generally, language concerning location (e.g., Johnston, 1982). Slobin (1973) suggested two basic types of such mental resources: "the conceptual and factual knowledge which gives rise to communicative intentions, and the cognitive processing mechanisms which participate in rule formation" (Johnston, op. cit.). The question for programming is whether there is a similar set of cognitive prerequisites which sets limits on the conceptual and factual knowledge which a child can master for different types of programming or programming constructs.

Since we know of no studies that directly pertain to this question, we can only offer a few observations at this time. We know that precocious performance is possible in limited contexts on Piagetian formal operational and concrete operational measures. However, it is the flexibility of intellectual operations that is constitutive of development (Kaplan, 1983; Werner, 1957). What is needed are methods to identify the limits of knowledge use. Cognitive supports in programming environments that permit a child to seemingly go beyond his or her current level of logical development are as yet poorly understood (e.g., memory support in the form of catalogues, traces, listings, automatic indenting), but may allow programming performance that would be unexpected given current knowledge of children's cognitive abilities as measured in other task environments. Before the question of what level of conceptual development is necessary for a child to learn to program can be answered more fully, we need more data on such things as how children actually behave in programming environments, what they find difficult, what sorts of models of the computer

and of programming they utilize, and what compensatory strategies they can discover or use to circumvent some of the formal demands of programming.

Conclusions

Narrow Focus of Existing Research

As our review of the issues and literature concerning the demands of learning to program makes evident, most research addressed to the aptitude and ability question has taken place within the multivariate tradition of looking at aptitude or ability interactions with "treatments," such as programming training courses, computer science courses, or similar programming experiences. But since the majority of such studies treat programming as a homogeneous skill, and aptitude or ability as static features of persons, in terms of the issues we posited as central in the study of the psychology of programming, the conclusions reached by those studies are correspondingly moot. And even if these studies were more directly relevant, since they were carried out with adults rather than children, there would be serious questions about their applicability to those of precollege age of interest for our purposes.

Instead, we have shown the necessity of highlighting the different types of programming and programmers, the different cognitive subtasks involved in programming, and the social character of many programming efforts, which raises many new research questions to be answered.

"Demands" Questions Cannot Be Separated from Goals

It has also been argued here that asking what cognitive demands programming has for precollege age people is a question which must be asked in terms of the goals of the specific programming activities of concern. This is to say that without specifying which programming projects or programming concepts are being asked about, one cannot answer the question of demands, because "it all depends"--one must ask: "demands of what"? We would expect that a child of almost any age, even preliterate toddlers, would be capable of working with a programmable device for some purposes. Once the focus of the demand question has been narrowed to specific programming subtasks or specific programming activities, such as learning recursion in Logo or writing a bubble sort program in BASIC, we are led to difficult questions about a taxonomy of programming activities and goals. And we have noted that what is particularly difficult about such questions is that the very nature of programming is evolution-

ary, and new goals emerge in tandem with new purposes for which programming activities are recognized as relevant.

Central Necessity of Focus on Goals of Computer Education

With questions about a taxonomy of programming activities in mind, we must then go on to ask what are the educational goals of teaching children about programming. Is there a core of programming concepts and techniques, and computer science knowledge, that we would define as "basic" in some sense, to be achieved by all citizens of our nation? Such a core requires definition, and then one may ask what cognitive abilities are necessary in order to attain that knowledge, and how a curriculum is best designed for the different ages in the precollege population. The implication here is that much empirical research in school communities will be required to determine the best developmental ages for introducing specific programming concepts and activities. But the value dimension of the pedagogy of programming is one that must be addressed in terms distinct from the learnability questions; value questions are in a different domain of discourse--they are culture-concepts rather than nature-concepts (Cassirer, 1960; also see Kaplan & Werner, 1983).

Beyond this core, however it is defined (and we have made some preliminary recommendations in this report), one may outline different specializations in programming, and perhaps define a distribution of such specializations that a precollege age programming curriculum should contain, for we may wish children to have some level of first-hand familiarity with a range of different types of programming before they enter college, even if we do not expect them to be competent in all of them.

Individual Versus Social Aspects of Programming Skill

The dominant view in programming education for children today stresses individual achievement of programming skills. Yet we have indicated that, in the professions of programming, a much more common emphasis is on the development of programs by teams of individuals (Brooks, 1982). The different subtasks of developing a program--design, coding, evaluation--are worked out as a group, with some specializations or domains of expertise more highly represented in some individuals than others. The advantages of teamwork on program development are many: neither the "tyranny," "egocentrism, nor "lack of foresight" of the individual is as likely to be manifested in the finished program (Shneiderman, 1980; Weinberg, 1971). As in Kripke's (1972) discussion of the locus of "knowledge of natural language," the community is what is collectively viewed as

expert, rather than individuals. The members of the programming collective hold each other in checks and balances in terms of responsibility for meeting goals, developing adequate documentation or verification, and avoiding abuses of power (with knowledge of how a program works private to you as "power"). This team emphasis is also apparent in frontier efforts within artificial intelligence to develop complex expert systems.

Limits of Instructability of Programming Concepts/Actions

We have also indicated that there is no research to date which addresses the limits of instructability of programming at some level of expertise for some defined developmental level. Instead, studies have dealt with representative instructional settings, in which the status quo--for example, one teacher for 30 students--is the assumed instructional organizational context. What a child of any given age--whether defined chronologically or mentally--is not able to learn, even with massive instruction and practice, is currently unknown. This is an issue of some importance. For even if what we are now concerned with is what children of a specific age can learn about programming given existing educational contexts, ultimately, in the years ahead, what we need to know, as sophisticated and less expensive software and hardware become available, is whether there are limits to the instructability of specific programming concepts and activities when every child has at his or her disposal an "ideal" programming environment in which individualized and self-paced instruction is a reality. These questions are unanswerable today, but we should recognize the context-bound nature of our current understanding of children and the development of programming skills.

Footnotes

[1]One may 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). (p. 316)

Pragmatics in earlier times was referred to as "rhetoric." The field of pragmatics of natural language has 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."

Though there are clearly some important disanalogies to natural languages, a pragmatics of programming languages may be said to concern at least the study of a programming language or languages from the point of view of the user, especially of the (design) choices he makes in the organization of lines of programming code within programs (or software systems), the constraints he encounters (such as the requirements of a debuggable program which is well-documented for future comprehension and modification, by himself or other users) in using programming language in social contexts, and the effects his 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.

[2]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" (refs.) are used to allow nonlinear control. For example, one may "goto" other lines in the program than the next one in BASIC; in such a case the flow of

control passes to the line of programming code referred to in the GOTO statement. Because the "flow of control" for a program may be quite complex, programmers often utilize programming flowcharts, either to serve as a high-level plan for the program they will write, or to document the flow of control represented in the lines of their program.

[3]What is "quasi-procedural" rather than "procedural" about giving and following task instructions, directions, and recipes is that, unlike the case of procedural instructions in a computer program, there is generally some ambiguity in the everyday examples, such that the instructions, directions, and recipes do not always have unequivocal meanings (unlike the programming commands), and unconstrained by strict sequentiality, so that one may in many instances choose to bypass steps in a recipe or set of instructions, or alter the order of steps in their execution. Neither of these options is available in the strict procedurality of programmed instructions to the computer. Yet the similarities between the everyday examples and the case of programming instructions are compelling enough to make their designation as "quasi-procedural" understandable.

Bibliography

Abelson, H., & diSessa, A. Turtle geometry. Cambridge, MA: MIT Press, 1981.

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

Alan, J. C. C. The selection and training of computer personnel in the United Kingdom. Proceedings of the 6th Annual Conference of the Special Interest Group for Computer Personnel Research, 1968, 6, 64-73.

Alspaugh, C. A. A study of the relationships between student characteristics and proficiency in symbolic and algebraic computer programming (Doctoral dissertation, University of Missouri, 1970). Dissertation Abstracts International, 1970, 31, 4672B. (University Microfilms No. 71-3301)

Alspaugh, C. A. Identification of some components of computer programming aptitude. Journal for Research in Mathematics Education, 1972, 3, 89-98.

Alspaugh, C. A. The relationship of grade placement to programming aptitude and FORTRAN programming achievement. Journal for Research in Mathematics Education, 1971, 2, 44-48.

Anderson, D. M. Predicting success in an introductory computer course from the Aptitude Test for Programmer Personnel and other data. Proceedings of ACM Computer Science Conference '75, 1975, 13, 18-20.

Anderson, J. R. Learning to program. Unpublished manuscript, Carnegie-Mellon University, 1983.

Anderson, J. R., Farrell, R., & Sauers, R. Learning to program in Lisp. Unpublished manuscript, Carnegie-Mellon University, 1983.

Anderson, J. R. Acquisition of cognitive skill. Psychological Review, 1982, 89, 369-406.

Anderson, J. R. (Ed.). Cognitive skills and their acquisition. Hillsdale, NJ: Erlbaum, 1981.

Anderson, J. R., & Bower, G. H. Human associative memory. Washington, DC: Winston, 1973.

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

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

Anzai, Y., & Simon, H.A. The theory of learning by doing. Psychological Review, 1979, 86, 124-140.

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). Englewood, CO: Science Applications, Inc., 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). Alexandria, VA: U.S. Army Research Institute for the Behavioral and Social Sciences, 1978.

Balzer, R., Goldman, N., & Wile, D. On the use of programming knowledge to understand informal process descriptions. Proceedings of Pattern Directed Inference Workshop in SIGART Newsletter, 1977, 63.

Bamberger, J., & Schon, D. A. Learning as reflective conversation with materials: Notes from work in progress (Working Paper No. 17). Massachusetts Institute of Technology, Division for Study and Research in Education, December 1982.

Barkin, S. An investigation into some factors affecting informations systems utilization. Unpublished doctoral dissertation, University of Minnesota, Minneapolis, 1974.

Barnes, P. Programmer paranoia revisited. Proceedings of the 13th Annual Computer Personnel Research Group Conference, 1975, 114-131.

Barstow, D. R. A knowledge-based system for automatic program construction. Proceedings of the Fifth International Joint Conference on Artificial Intelligence, 1977, 382-388.

Barstow, D. R. Knowledge-based program construction. Amsterdam: North Holland, 1979.

Bateman, G.R. Predicting performance in a basic computer course. Proceedings of the 5th Annual Meeting of American Institute for Decision Sciences, 1973, 130-133.

Bauer, R., Mehrens, W. A., & Vinsonhaler, J. F. Predicting performance in a computer programming course. Educational and Psychological Measurement, 1968, 28, 1159-1164.

Bell, D. Programmer selection and programmer errors. The Computer Journal, 1976, 19, 202-206.

Bereiter, C. Development in writing. In L. W. Gregg, & E. R. Steinberg, (Eds.), Cognitive Processes in Writing. Hillsdale, NJ: Erlbaum, 1979.

Bereiter, C., & Scarmadalia, M. From conversation to composition: Instruction in a developmental process. In R. Glaser (Ed.), Advances in instructional psychology (Vol. 2). Hillsdale, NJ: Erlbaum, 1982.

Berger, R. M. Computer personnel selection and criteria development. Proceedings of the 2nd Annual Computer Personnel Research Group Conference, 1964.

Berger, R. M., & Berger, F. R. The Berger series of computer personnel tests. Santa Monica, CA: Psychometrics [no date].

Biamonte, A. J. Predicting success in programmer training. Proceedings of the 2nd Annual Computer Personnel Research Group Conference, 1964, 2, 9-12.

Biamonte, A. J. A study of the effect of attitudes on the learning of computer programming. Proceedings of the 3rd Annual Computer Personnel Research Group Conference, 1965, 3, 68-74.

Biermann, A. W. Approaches to automatic programming. In M. Rubinoff & M. C. Yovits (Eds.), Advances in computers (Vol. 15). New York: Academic Press, 1976.

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.

Bloom, A. M. Advances in the use of programmer aptitude tests. In T. A. Rullo (Ed.), Advances in computer programming. Philadelphia: Heyden & Son, 1980.

Bloom, A. M. Test the test for programming applicants. Data Management, 1978, 16, 37-39.

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

Bonar, J., & Soloway, E. Uncovering principles of novice programming (Research Report No. 240). New Haven: Yale University Department of Computer Science, November 1982. (To appear in the Tenth SIGPLAN-SIGACT Symposium on the Principles of Programming Languages, Austin, Texas, January 1983).

Borst, M. A. Programmers vs. non-programmers in a Fortran comprehension test (Tech. Rep. No. TR-78-388100-4). Arlington, VA: Information Systems Programs, General Electric Company, 1978.

Boysen, J. P., & Keller, R. F. Measuring computer program comprehension. Proceedings of the 11th ACM Symposium on Computer Science Education, 1980.

Brooke, J. Tools for the job: The human factor (programming). Computer Age, 1980, 11, 31-32.

Brooke, J. B., & Duncan, K. D. An experimental study of flowcharts as an aid to identification of procedural faults. Ergonomics, 1980(a), 23, 387-399.

Brooke, J. B., & Duncan, K. D. Experimental studies of flowchart use at different stages of program debugging. Ergonomics, 1980(b), 23, 1057-1091.

Brooks, F. P. The mythical man-month. Reading, MA: Addison-Wesley, 1982.

Brooks, R. E. A model of cognitive behavior in writing code for computer programs. Unpublished doctoral dissertation, Carnegie-Mellon University, 1975.

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

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

Brooks, R. E. Using a behavioral theory of program comprehension in software engineering. Proceedings of the 3rd International Conference on Software Engineering, 1978, 196-201.

Brooks, R. E. A theoretical analysis of the role of documentation in the comprehension of computer programs. Proceedings of the Conference on Human Factors in Computer Systems, Gaithersburg, MD, 1982.

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

Brown, A. L. Metacognition, executive control, self-regulation and other even more mysterious mechanisms. In R. H. Kluwe & F. E. Weinert (Eds.), Metacognition, motivation and learning. West Germany: Luhlhammer, 1983(a). In press.

Brown, A. L. Learning to learn how to read. In J. Langer & T. Smith-Burke (Eds.), Reader meets author, bridging the gap: A psycholinguistic and social linguistic perspective. Newark, NJ: Dell, 1983(b).

Brown, A. L., Bransford, J. D., Ferrara, R. A., & Campione, J. C. Learning, remembering, and understanding. To appear in J. H. Flavell & E. M. Markman (Eds.), Carmichael's manual of child psychology (Vol. 1). New York: Wiley, 1983.

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

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

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

Buff, R. J. The prediction of academic achievement in FORTRAN language programming courses. Unpublished doctoral dissertation, New York University, 1972.

Burns, W. J. A study of interaction between aptitude and treatment in the learning of a computer programming language (Doctoral dissertation, University of Maryland, 1974). Dissertation Abstracts International, 1974, 35, 904A. (University Microfilms No. 74-17).

Burton, R. B. DEBUGGY: Diagnosis of errors in basic mathematics skills. In D. H. Sleeman & J. S. Brown (Eds.), Intelligent tutoring systems. New York: Academic Press, 1981.

Byrne, R. Planning meals: Problem-solving on a real data-base. Cognition, 1977, 5, 287-332.

Byrant, A., & Ameen, P. A. Use of personal history, activities, ability and attitude questionnaire to predict success as a systems analyst. Proceedings of the 16th Annual Conference of the Computer Personnel Research Group Conference, 1980, 133-143.

Cannon, W. M. Toward a new vocational interest scale for computer programmers--a procedural report. Proceedings of the 3rd Annual Computer Personnel Research Group Conference, 1965, 3, 68-74.

Cannon, W. M., & Perry, D. K. A vocational interest scale for computer programmers. Proceedings of the 4th Annual Conference of the Special Interest Group for Computer Personnel Research, 1966, 4, 61-80.

Capstick, C. K., Gordon, J. D., & Salvadori, A. Predicting performance by university students in introductory computing courses. Special Interest Group in Computer Science Education Bulletin, 1975, 7, 21-29.

Card, S. K. User perceptual mechanisms in the search of computer command menus. Proceedings of the Conference on Human Factors in Computer Systems, Gaithersburg, MD, 1982.

Card, S. K., Moran, T., & Newell, A. Applied information-processing psychology: The human-computer interface. Hillsdale, NJ: Erlbaum, 1982.

Card, S. K., Moran, T. P., & Newell, A. Computer text editing: An information processing analysis of a routine cognitive skill. Cognitive Psychology, 1980, 12, 32-74.

Card, S. K., & Newell, A. Are we ready for a cognitive engineering? (AIP Paper No. 14). Xerox PARC, 1981.

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

Case, R., & Kurland, D. M. A new measure for determining children's subjective organization of speech. Journal of Experimental Child Psychology, 1980, 30, 206-222.

Case, R., Kurland, D. M., & Goldberg, J. Operational efficiency and the growth of short-term memory span. Journal of Experimental Child Psychology, 1982, 33, 386-404.

Cassirer, E. The logic of the humanities. Translated by C. S. Howe. New Haven: Yale University Press, 1960.

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

Cheney, P. Cognitive style and student programming ability: An investigation. Association for Educational Data Systems Journal, 1980, 13, 285-291.

Chi, M.T.H., Feltovich, P.J., & Glaser, R. Categorization and representation of physics problems by experts and novices. Cognitive Science, 1981, 5, 121-152.

Clark, H. H. Bridging. In P. N. Johnson-Laird & P. C. Wason (Eds.), Thinking: Readings in cognitive science. Cambridge, MA: Cambridge University Press, 1977.

Chipman, S., Siegel, J., & Glaser, R. (Eds.), Thinking and learning skills: Current research and open questions. Hillsdale, NJ: Erlbaum, 1983. In press.

Clement, J., Lochhead, J., & Monk, G. Translation difficulties in learning mathematics (Tech. Rep.). Amherst, MA: Cognitive Development Project, Department of Physics and Astronomy, University of Massachusetts, 1979.

Collins, A., & Gentner, D. Constructing runnable mental models. Proceedings of the Fourth Annual Conference of the Cognitive Science Society. Ann Arbor, MI, August 1982.

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

Correnti, R. J. Predictors of success in the study of computer programming at two-year institutions of higher education (Doctoral dissertation, Ohio University, 1969). Dissertation Abstracts International, 1969, 30, 3718A. (University Microfilms No. 70-4732)

Cromer, R. F. The development of language and cognition: The cognition hypothesis. In B. Foss (Ed.), New perspectives in child development. London, England: Penguin, 1974, pp. 184-252.

Crystal, D. A first dictionary of linguistics and phonetics. Cambridge, MA: Cambridge University Press, 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, 1979, SE-5, 96-104.

DeKleer, J., & Brown, J. S. Mental models of physical mechanisms and their acquisition. In J. R. Anderson (Ed.), Cognitive skills and their acquisition. Hillsdale, NJ: Erlbaum, 1981.

Dewey, J. Human nature and conduct. New York: Henry Holt, 1922.

Dewey, J. The school and society. Chicago: University of Chicago Press, 1900.

Dickmann, R. A., & Lockwood, J. 1966 survey of test use in computer personnel selection. Proceedings of the 4th Annual Computer Personnel Research Group Conference, 1966, 4, 15-27.

Dijkstra, E. W. GO-TO statement considered harmful. Communications of the ACM, 1968, 11, 147-148, 538, 541.

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

DiPersio, T., Isbister, D., & Shneiderman, B. An experiment using memorization/reconstruction as a measure of programmer ability. International Journal of Man-Machine Studies, 1980, 13, 339-354.

diSessa, A. A. Unlearning Aristotelian physics: A study of knowledge-based learning. Cognitive Science, 1982, 6, 37-75.

duBoulay, J. B. H. Teaching teachers mathematics through programming. International Journal of Mathematical Education, Science and Technology, 1980, 11, 347-360.

duBoulay, J. B. H., & O'Shea, T. How to work the LOGO machine: A primer for ELOGO (DAI Occasional Paper No. 4). Edinburgh, Scotland: Department of Artificial Intelligence, University of Edinburgh, 1976.

duBoulay, J. B. H., & O'Shea, T. Seeing the works: A strategy for teaching interactive programming (DAI Working Paper No. 28). Edinburgh, Scotland: Department of Artificial Intelligence, University of Edinburgh, 1978.

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, 1981, 14, 237-249.

Durward, M. L. The Computer Programmer Aptitude Battery: A field trial. Vancouver, BC: Vancouver Board of School Trustees, 1973. (ERIC Document Reproduction Service No. ED 088 913)

Dwyer, T. A. Soloworks: Computer based laboratories for high school mathematics. Science and Mathematics, 1975, 93-99.

Egly, D. G., & Wescourt, K. T. Cognitive style, categorization and vocational effects on performance of REL database users. SIGSOC Bulletin, 1982, 13, 91-97.

Ehrlich, K., & Soloway, E. An empirical investigation of the tacit plan knowledge in programming. In J. Thomas & M. Schneider (Eds.), Human factors in computer systems. Norwood, NJ: Ablex, 1983.

Eisenstadt, M., & Laubsch, J. H. Towards an automated debugging assistant for novice programmers. Proceedings of the AISB-80 Conference on Artificial Intelligence, Amsterdam, The Netherlands, 1980.

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.

Ericcson, K. A., & Simon, H. Verbal reports as data. Psychological Review, 1980, 87, 215-251.

Falmagne, R. J. (Ed.). Reasoning: Representation and process in children and adults. Hillsdale, NJ: Erlbaum, 1975.

Favaro, P. My five year old knows BASIC. Creative Computing, 1983, 9, 158-166.

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. Cambridge, MA: Bolt Beranek and Newman, October 1981.

Feurzeig, W., Papert, S., Bloom, M., Grant, R., & Solomon, C. Programming languages as a conceptual framework for teaching mathematics (Report No. 1899). Cambridge, MA: Bolt Beranek and Newman, 1969.

Flavell, J., & Draguns, J. A microgenetic approach to perception and thought. Psychological Bulletin, 1957, 54, 197-217.

Flower, L., & Hayes, J. R. Plans that guide the composing process. In C. Fredericksen, M. Whiteman, & J. Dominic (Eds.), Writing: The nature, development and teaching of written communication. Hillsdale, NJ: Erlbaum, 1979.

Floyd, R. W. The paradigms of programming. Communications of the ACM, 1979, 22, 455-460.

Fowler, G. C., & Glorfeld, L. W. Predicting aptitude in introductory computing: A classification model. Association for Educational Data Systems Journal, 1981, 14, 96-109.

Fowler, G. C., & Glorfeld, L. W. Validation of a mode for predicting aptitude for introductory computing. Special Interest Group for Computer Science Education Bulletin, 1982, 14, 140-143.

Friedman, S. L., Scholnick, E. K., & Cocking, R. R. (Eds.), Blueprints for thinking: The development of social and cognitive planning skills. Cambridge, MA: Cambridge University Press, 1983. In press.

Galanter, E. Kids and computers: The parents' microcomputer handbook. New York: Putnam, 1983.

Gannon, C. Error detection using path testing and static analysis. Computer, 1979, 26-31.

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

Gannon, J. D. Data types and programming reliability: Some preliminary evidence. Proceedings of the Symposium on Computer Software Engineering, 1976(b).

Gannon, J. D. An experimental evaluation of data type conventions. Communications of the ACM, 1977, 20, 584-595.

Gannon, J. D., & Horning, J. J. Language design for programming reliability. IEEE Transactions on Software Engineering, 1975, SE-1, 179-191.

Gibson, E. J., & Levin, H. The psychology of reading. Cambridge, MA: MIT Press, 1975.

Gick, M. L., & Holyoak, K. J. Analogical problem solving. Cognitive Psychology, 1980, 12, 306-355.

Gick, M. L., & Holyoak, K. J. Schema induction and analogical transfer. Cognitive Psychology, 1982, 15, 1-39.

Goldberg, P. C. Structured programming for non-programmers. In Structured programming: An infotech state of the art report. Maidenhead: Infotech International, 1976.

Goldin, S. E., & Hayes-Roth, B. Individual differences in planning processes. A Rand Note (N-1488-ONR) prepared for the Office of Naval Research, June 1980.

Goldman, N., Balzer, R., & Wile, D. The inference of domain structure from informal process descriptions. Proceedings of Pattern Directed Inference Workshop in SIGART Newsletter #63, 1977(a).

Goldman, N., Balzer, R., & Wile, D. The use of a domain model in understanding informal process descriptions. Proceedings of the Fifth Joint Conference on Artificial Intelligence, 1977(b).

Goldstein, I. Understanding simple picture programs (AI Tech. Rep. No. 294). Cambridge, MA: MIT Artificial Intelligence Laboratory, 1974.

Goldstein, I. P. The genetic graph: A representation for the evolution of procedural knowledge. International Journal of Man-Machine Studies, 1979, 11, 51-77.

Goldstein, I., & Miller, M. AI based personal learning environments (AI Memo No. 384). Cambridge, MA: Massachusetts Institute of Technology, 1976(a).

Goldstein, I., & Miller, M. Structured planning and debugging: A linguistic theory of design (AI Memo No. 387). Cambridge, MA: MIT Artificial Intelligence Laboratory, 1976(b).

Goldstein, I., & Papert, S. Artificial intelligence, language, and the study of knowledge. Cognitive Science, 1977, 1, 84-123.

Gotterer, M. H., & Stalnaker, A. W. Predicting programming performance among non-preselected trainee groups. Proceedings of the 2nd Annual Computer Personnel Research Group Conference, 1964, 2, 61-82.

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

Gould, J. D., & Drongowski, P. An exploratory investigation of computer program debugging. Human Factors, 1974, 16, 258-277.

Gray, J. D. Predictibility of success and achievement level of data processing technology students at the two-year post-secondary level. Unpublished doctoral dissertation, Georgia State University, 1974.

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

Green, T. R. G. Programming as cognitive activity. In H. T. Smith & T. R. G. Green (Eds.), Human interaction with computers. New York: Academic Press, 1980(a).

Green, T. R. G., Sime, M. E., & Fitter, M. J. The problem the programmer faces. Ergonomics, 1980(b), 23, 893-907.

Green, T. R. G., Sime, M. E., & Guest, D. J. The effect of syntax on reading and generating programming languages. Bulletin of the British Psychological Society, 1974.

Greeno, J. Indefinite goals in well-structured problems. Psychological Review, 1976, 83, 479-491.

Halasz, F., & Moran, T. P. Analogy considered harmful. Proceedings of the Conference on Human Factors in Computer Systems, Gaithersburg, MD, 1982.

Hall, R. S. The construction of a selection battery for programmers adapted to South African conditions. Proceedings of the 8th Annual Computer Personnel Research Group Conference, 1970, 108-143.

Halstead, M. H. Elements of software science. New York: Elsevier, 1977.

Halstead-Nussloch, R. Programmer set and degree of language structure in programming performance. Proceedings of the 25th Annual Human Factors Society Meetings, 1981, 12-16.

Harmon, M. Computer literacy: A teacher challenge. Section 12: "Employment outlook in high technology." The New York Times, March 27, 1983, pp. 27-28.

Harvey, B. Why LOGO? Byte, 1982, 7, 163-193.

Hawkins, J., & Fiess, K. The effects of programming experience on children's conceptions of computer functioning (Report No. 13). New York: Center for Children and Technology, Bank Street College of Education, February 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, 1983, 2. In press.

Hayes, J. R. Problem topology and the solution process. Journal of Verbal Learning and Verbal Behavior, 1965, 4, 371-379.

Hayes, J. R., & Simon, H. A. Psychological differences among problem isomorphs. In N. J. Castellan, Jr., D. B. Pisoni, & G. R. Potts (Eds.), Cognitive theory (Vol. 2). Hillsdale, NJ: Erlbaum, 1977.

Hayes-Roth, B. Estimation of time requirements during planning: The interactions between motivation and cognition. A Rand Note (N-1581-)NR) prepared for the Office of Naval Research, November 1980.

Hayes-Roth, B., & Hayes-Roth, F. A cognitive model of planning. Cognitive Science, 1979, 3, 275-310.

Hayes-Roth, B., Hayes-Roth, F., Shapiro, N., & Wescourt, K. Planners' workbench: A computer aid to re-planning. A Rand Paper (P-6688), October 1981.

Heller, J. I., & Greeno, J. G. Information processing analyses of mathematical problem solving. In R. W. Tyler & S. H. White (Eds.), Testing, teaching and learning. Washington, DC: U.S. Department of Health, Education, and Welfare, 1979.

Helms, S. Finding a round peg for a round hole (staff recruitment). Data Processing, 1979, 21, 10-11.

Hewitt, C., & Smith, B. Towards a programming apprentice. IEEE Transactions on Software Engineering, 1975, SE-1.

Hierdorn, C. E. Automatic programming through natural language dialogue: A survey. IBM Journal of Research and Development, 1976, 20, 302-313.

Hoare, C. A. R. Communicating sequential processes. Communications of the ACM, 1978, 21, 666-677.

Hoc, J. M. Role of mental representation in learning a programming language. International Journal of Man-Machine Studies, 1977, 9, 87-105.

Hollenback, G. P., & McNamara, W. J. CUCPAT and programming aptitude. Personnel Psychology, 1965, 18, 101-106.

Holyoak, K. J. Analogical thinking and human intelligence. In R. J. Sternberg (Ed.), Advances in the psychology of human intelligence (Vol. 2). Hillsdale, NJ: Erlbaum, 1983.

Howe, J. A. M. Developmental stages in learning to program. In F. Klix & J. Hoffman (Eds.), Cognition and memory: Interdisciplinary research of human memory activities. Amsterdam: North Holland, 1980.

Howe, J. A. M. Learning mathematics through LOGO programming (Research Paper No. 153). Edinburgh, Scotland: Department of Artificial Intelligence, University of Edinburgh, 1981.

Howe, J. A. M., O'Shea, T., & Plane, F. Teaching mathematics through LOGO programming: An evaluation study. In R. Lewis & E. D. Tagg (Eds.), Computer-assisted learning--scope, progress and limits. Amsterdam: North Holland, 1979.

Howell, M. A., Vincent, J. W., & Gay, R. A. Testing aptitude for computer programming. Psychological Reports, 1967, 20, 1251-1256.

Hunt, D., & Randhawa, B. S. Relationship between and among cognitive variables and achievement in computational science. Educational and Psychological Measurement, 1973, 33, 921-928.

Hunt, E. Mechanics of verbal ability. Psychological Review, 1978, 85, 109-130.

Inhelder, B., & Piaget, J. The growth of logical thinking from childhood to adolescence (A. Parsons & S. Milgram, Trans.). New York: Basic Books, 1958.

Inhelder, B., Sinclair, H., & Bovet, M. Learning and the development of cognition. Cambridge, MA: Harvard University Press, 1974.

International Business Machines. Manual for administration and scoring the Aptitude Test for Programmer Personnel. White Plains, NY: IBM Technical Publications Department, 1964.

Irons, D. M. Predicting programming performance in novice programmers by measures of cognitive abilities (Doctoral dissertation, Texas Christian University, 1982). Dissertation Abstracts International, 1982, 43, 1283B.

Jacobs, S. J. Cognitive predictors of success in computer programmer training. Proceedings of the 11th Annual Conference of the Special Interest Group for Computer Personnel Research, 1973, 11, 98-113.

Jacknow, L. Learning, evaluation, and systems models in engineering, science, and technology curricula. Journal of Educational Technology Systems, 1979-1980, 8, 51-66.

Jackson, M. A. Principles of program design. New York: Academic Press, 1975.

Jackson, M. A. The design and use of conventional programming languages. In H. T. Smith & T. R. G. Green (Eds.), Human interaction with computers. London: Academic Press, 1980.

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.

Jeffries, R., Turner, A. A., Polson, P. G., & Atwood, M. E. The processes involved in designing software. In J. R. Anderson (Ed.), Cognitive skills and their acquisition. Hillsdale, NJ: Erlbaum, 1981.

Johnson, R. T. Review of Computer Programmer Aptitude Battery. In O. K. Buros (Ed.), The Seventh Mental Measurements Yearbook. Highland Park, NJ: Gryphon Press, 1972.

Johnson, R. T. Review of IBM Aptitude Test for Programmer Personnel. In O. K. Buros (Ed.), The Seventh Mental Measurements Yearbook. Highland Park, NJ: Gryphon Press, 1972.

Johnson, W. L., Draper, S., & Soloway, E. An effective bug classification scheme must take the programmer into account. Proceedings of the Workshop on High-Level Debugging, Palo Alto, CA, 1983.

Johnston, J. R. Cognitive prerequisites: The evidence from children learning English. In D. I. Slobin (Ed.), Universals of language acquisition. Hillsdale, NJ: Erlbaum, 1982.

Kahn, G., & MacQueen, D. B. Coroutines and networks of parallel processes. Proceedings of the IFIPS Congress. Amsterdam: North Holland, 1977.

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, August 4-6, 1982.

Kamii, C. Pedagogical principles derived from Piaget's theory: Relevance for educational practice. In M. Schwebel & J. Raph (Eds.), Piaget in the classroom. London: Routledge & Kegan Paul, 1974.

Kaplan, B. Genetic-dramatism. In S. Wapner & B. Kaplan (Eds.), Toward a holistic developmental psychology. Hillsdale, NJ: Erlbaum, 1983.

Kaplan, B., & Wapner, S. (Eds.), Value presuppositions in theories of human development: Proceedings of the Second Biennial Conference of the Heinz Werner Institute (1983). Hillsdale, NJ: Erlbaum, to appear in 1984.

Karten, H. Issues in hiring programmers: 2. Testing: Pros and cons of standardized aptitude tests for programmers. Computerworld, 1982, 16, 15-17.

Kidder, T. The soul of a new machine. Boston: Little, Brown, 1981.

Klahr, D., & Robinson, M. Formal assessment of problem-solving and planning processes in preschool children. Cognitive Psychology, 1981, 13, 113-147.

Kripke, S. A. Naming and necessity. In D. Davidson & G. Harman (Eds.), Semantics of natural language. Dordrecht, Holland: Reidel, 1972.

Kurland, D. M., & Pea, R. D. Children's mental models of recursive Logo programs (Tech. Rep. No. 10). New York: Center for Children and Technology, Bank Street College of Education, February 1983(a).

Kurland, D. M., & Pea, R. D. Expert programmers: Diversity in processes of program development (Working Paper). New York: Center for Children and Technology, Bank Street College of Education, April 1983(b).

Laboratory of Comparative Human Cognition. Culture and cognitive development. In W. Kessen (Ed.), Carmichael's manual of child psychology: History, theories and methods. New York: Wiley, 1983. In press.

Laboratory of Comparative Human Cognition. Microcomputer communication networks for education. The Quarterly Newsletter of the Laboratory of Comparative Human Cognition, April 1982, 4 (2).

Larkin, J. H. Skilled problem solving in physics (Tech. Rep.). Berkeley, CA: Group in Science and Mathematics Education, University of California at Berkeley, 1977.

Larkin, J. H. Teaching problem solving in physics: The psychological laboratory and the practical classroom. In D. T. Tuma & F Reif (Eds.), Problem solving and education: Issues in teaching and research. Hillsdale, NJ: Erlbaum, 1980.

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

Larsen, S. G. Kids and computers: The future is today. Creative Computing, 1979, 5, 58-60.

Lawler, R. W. Extending a powerful idea (Logo Memo No. 58). Cambridge, MA: MIT Artificial Intelligence Laboratory, July 1980.

Ledbetter, W. N. Programming aptitude: How significant is it? Personnel Journal, 1975, 54, 165-166, 175.

Ledvinka, J., & Schoenfeldt, L. F. Legal developments in employment testing: "Albermarle" and beyond. Personnel Psychology, 1978, 31.

Leeper, R. R., & Silver, J. L. Predicting success in a first programming course. Special Interest Group in Computer Science Education Bulletin, 1982, 14, 147-150.

Leitner, H. H., & Lewis, H. R. Why Johnny can't program: A progress report. Special Interest Group in Computer Science Education Bulletin, 1978, 10, 266-276.

Lemos, R. S. A comparison of non-business and business student test scores in BASIC. Association for Educational Data Systems Journal, 1981, 14, 151-158.

Lemos, R. FORTRAN programming: An analysis of pedagogical alternatives. Journal of Educational Data Processing, 1975, 12, 21-29.

Lemos, R. S. Measuring programming language proficiency. Association for Educational Data Systems Journal, 1980, 13, 261-273.

Lemos, R. Methods, styles, and attitudes in the programming language classroom. Computer, 1980, 13, 58-65.

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

Lewis, C. Skill in algebra. In J. R. Anderson (Ed.), Cognitive skills and their acquisition. Hillsdale, NJ: Erlbaum, 1981.

Lochhead, J. An anarchistic approach to teaching problem solving methods. Paper presented at the Annual Meeting of the American Educational Research Association, San Francisco, April 1979.

Lotheridge, C. D. Discussion on papers of Mussio, Wahlstrom and Seiner. Proceedings of the 9th Annual Computer Personnel Research Group Conference, 1971, 9, 47-53.

Love, L. T. Relating individual differences in computer programming performance to human information processing abilities (Doctoral dissertation, University of Washington, 1977). Dissertation Abstracts International, 1977, 38, 1443B. (University Microfilms No. 77-18379)

Luchins, A. S., & Luchins, E. H. Rigidity of behavior: A variational approach to the effect of Einstellung. Eugene, OR: University of Oregon Press, 1959.

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

Luehrmann, A. Should the computer teach the student or vice versa? In R.P. Taylor (Ed.), The computer in the school: Tutor, tool, tutee. New York: Teachers College Press, 1981(b).

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

Martin, M. A. A study of the concurrent validity of the Computer Programmer Aptitude Battery. Studies in Personnel Psychology, 1971, 3, 69-76.

Matz, M. Towards a process model of high school algebra errors. In D. H. Sleeman & J. S. Brown (Eds.), Intelligent tutoring systems. New York: Academic Press, 1981.

Mayer, D. B., & Stalnaker, A. W. Computer personnel research--issues and progress in the 60's. Proceedings of the 5th Annual Conference of the Special Interest Group for Computer Personnel Research, 1967, 5, 6-41.

Mayer, D. B., & Stalnaker, A. W. Selection and evaluation of computer personnel--the research history of SIG/CPR. Proceedings of the 23rd National Conference of the Association for Computing Machinery, 1968, 657-670.

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

Mayer, R. E. A psychology of learning BASIC. Communications of the ACM, 1979, 22, 589-593.

Mayer, R. E. The psychology of learning computer programming by novices. Computing Surveys, 1981, 13, 121-141.

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

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

Mazlack, L. J. Does a computer have sexual preferences? Special Interest Group for Computer Science Education Bulletin, 1972, 8, 74-78.

Mazlack, L. J. Predicting student success in an introductory programming course. Comput. J., 1978, 21, 380-381.

Mazlack, L. J. Identifying potential to acquire programming skill. Communications of the Association for Computing Machinery, 1980, 23, 14-17.

McDermott, J., & Larkin, J. H. Representing textbook physics problems. Proceedings of the 2nd National Conference of the Canadian Society for Computational Studies of Intelligence, University of Toronto, 1978.

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

McNamara, W. J. The selection of computer personnel--past, present, future. Proceedings of the 5th Annual Conference of the Special Interest Group for Computer Personnel Research, 1967, 5, 52-56.

McNamara, W. J., & Hughes, J. L. Review of research on the selection of computer programmers. Personnel Psychology, 1961, 14, 39-51.

McNamara, W. J., & Hughes, J. L. Manual for the revised Programmer Aptitude Test. White Plains, NY: IBM, 1969.

McNicholl, D. G., & Magell, K. The subjective nature of programming complexity. Proceedings of the Conference on Human Factors in Computer Systems, Gaithersburg, MD, 1982.

Meltzer, B. Brains and Programs. Proceedings of the International Computing Symposium, Liege, Belgium. Amsterdam: North Holland, 1977, pp. 81-84.

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

Miller, L. A. Natural language programming: Styles, strategies, contrasts (Tech. Rep. No. RC-8687). New York: IBM Research Center, 1980.

Miller, L. A., & Becker, C. A. Programming in natural English (Tech. Rep. RC-5134). Yorktown Heights, NY: IBM Watson Research Center, 1974.

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

Miller, M. A structured planning and debugging environment for elementary programming. International Journal of Man-Machine Studies, 1978, 11, 79-95.

Miller, M., & Goldstein, I. Parsing protocols using problem solving grammars (AI Memo No. 385). Cambridge, MA: MIT Artificial Intelligence Laboratory, 1976(a).

Miller, M., & Goldstein, I. SPADE: A grammar based editor for planning and degugging programs (AI Memo No. 386). Cambridge, MA: MIT Artificial Intelligence Laboratory, 1976(b).

Miller, M. L., & Goldstein, I. PAZATN: A linguistic approach to automatic analysis of elementary programming protocols (AI Memo No. 388). Cambridge, MA: MIT Artificial Intelligence Laboratory, 1976(c).

Miller, M. L., & Goldstein, I. Problem solving grammars as formal tools for intelligent CAI. Proceedings of ACM77, 1977.

Miller, M. L., & Goldstein, I. Structured planning and debugging. Proceedings of the Fifth International Joint Conference on Artificial Intelligence, Cambridge, MA, 1977.

Miller, M. L., & Goldstein, I. P. Planning and debugging in elementary programming. In P.H. Winston & R.H. Brown (Eds.), AI: An MIT Perspective (Vol. 1). Cambridge, MA: MIT Press, 1979.

Milner, S. D. The effects of teaching computer programming on performance in mathematics (Doctoral dissertation, University of Pittsburgh, 1973). Dissertation Abstracts International, 1973, 33, 4183A-4184A. (University Microfilms No. 72-4112)

Milner, S. The effects of computer programming on performance in mathematics. ERIC Report No. ED076391, 1973.

Minsky, M. Form and content in computer science. Communications of the ACM, 1970, 17, 197-215.

Moher, T., & Schneider, G. M. Methods for improving controlled experimentation in software engineering. Proceedings of the Fifth International Conference on Software Engineering. Silver Spring, MD: IEEE Computer Society, 1981, pp. 224-233.

Moher, T., & Schneider, G. M. Methodology and experimental research in software engineering. International Journal of Man-Machine Studies, 1982, 16, 65-87.

Monk, G. S. Constructive calculus. Seattle: University of Washington, 1978.

Moran, T. P. The command language grammar: A representation for the user interface of interactive computer systems. International Journal of Man-Machine Studies, 1981, 15, 3-50.

Mussio, J. J., & Wahlstrom, M. W. Predicting performance of programmer trainees in a post-high school setting. Proceedings of the 9th Annual Conference of the Special Interest Group for Computer Personnel Research, 1971, 9, 26-46.

National Assessment of Educational Progress. Procedural handbook: 1977-78 mathematics assessment. Denver, CO: Education Commission of the States, 1980.

National Institute of Education. Demands and cognitive consequences of computer learning. Request for Proposal NIE-R-82-0011, July 1982.

Nelson, B. W. Issues in hiring Programmers: 1. Selection: How to pick candidates for entry-level programmer training. Computerworld, 1982, 16, 1-6.

Nelson, B., & Lowrey, J. Issues in hiring programmers: 3. Hiring versus training: Experienced programmers or trainees? A productive time/cost model. Computerworld, 1982, 16, 21-26.

Nelson, K. The syntagmatic-paradigmatic shift revisited: A review of research and theory. Psychological Bulletin, 1977, 84, 93-116.

Newell, A. One final word. In D. T. Tuma & F. Reif (Eds.), Problem solving and education. Hillsdale, NJ: Erlbaum, 1980.

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

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

Newstead, P. R. Grade and ability predictions in an introductory programming course. Special Interest Group in Computer Science Education Bulletin, 1975, 7, 87-91.

Nickerson, R. S. Thoughts on teaching thinking. Educational Leadership, October 1981(a).

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

Nickerson, R. S. Understanding understanding (Tech. Rep.). Cambridge, MD: Bolt Beranek and Newman, 1982.

Nisbett, R. E., & Ross, L. Human inference: Strategies and shortcomings of social judgment. Englewood Cliffs, NJ: Prentice-Hall, 1980.

Nisbett, R. E., & Wilson, T. D. Telling more than we can know: Verbal reports on mental processes. Psychological Review, 1978, 84, 231-259.

Norcio, A. F., & Kerst, S. M. Human memory organization for computer programs. Human Factors. In press.

Norman, D.A. Worsening the knowledge gap: The mystique of computation builds unnecessary barriers. Paper presented at New York Academy of Sciences Conference on "Computer culture: The Scientific, Intellectual, and Social Impact of the Computer," New York City, April 5-7, 1983.

Olson, G., Mack, R. L., & Dufy, S. A. Cognitive aspects of genre (Tech. Rep. 11). Ann Arbor: University of Michigan Center for Cognitive Science, 1981.

Owens, B. B. An interaction study of reasoning aptitudes, model presence and methods of approach in the learning of a computer programming language (Doctoral dissertation, New York University, 1978). Dissertation Abstracts International, 1978, 38, 7122A. (University Microfilms No. 7808480)

Palormo, J. M. The Computer Programmer Aptitude Battery--a description and discussion. Proceedings of the 5th Annual Conference of the Special Interest Group for Computer Personnel Research, 1967, 5, 57-63.

Palormo, J. M. Computer Programmer Aptitude Battery, manual. Chicago: Science Research Associates, 1967.

Palormo, J. M., Campbell, B. A., & Schofeld, M. Computer Programmer Aptitude Battery, examiner's manual (2nd ed.). Chicago: Science Research Associates, 1974.

Papert, S. Mindstorms. New York: Basic Books, 1980.

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

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

Papert, S., Watt, D., diSessa, A., & Weir, S. Final report of the Brookline LOGO Project: An assessment and documentation of a children's computer laboratory. Cambridge, MA: MIT Division for Study and Research in Education, 1979.

Pea, R. D. Programming and problem solving: Children's experience with LOGO. Paper presented at symposium, "Chameleon in the classroom: Developing roles for computers", Annual Meetings of the American Educational Research Association, Montreal, Canada, April 1983. (Also Technical Report No. 12, Bank Street College of Education, Center for Children and Technology)

Pea, R. D. What is planning development the development of? In D. Forbes & M. Greenberg (Eds.), New directions in child development: Children's planning strategies (Vol. 18). San Francisco: Jossey-Bass, December 1982.

Pea, R. D., & Hawkins, J. A microgenetic study of planning processes in a chore-scheduling task. In S. L. Friedman, E. K. Scholnick, & R. R. Cocking (Eds.), Blueprints for thinking: The development of social and cognitive planning skills. Cambridge, MA: Cambridge University Press, 1983. In press.

Pea, R. D., Hawkins, J., & Sheingold, K. Developmental studies on learning LOGO computer programming. Paper presented at the Annual Meetings of the Society for Research in Child Development, Detroit, April 1983.

Pea, R. D., & Kurland, D. M. On the cognitive effects of learning computer programming (Tech. Rep. No. 9). New York: Center for Children & Technology, Bank Street College of Education, 1983(a).

Pea, R. D., & Kurland, D. M. Learning LOGO programming and the development of planning skills (Tech. Rep. No. 16). New York: Center for Children and Technology, 1983(b).

Penney, G. Aptitude testing for employment in computer jobs. In O. Lecarme & R. Lewis (Eds.), Computers in education. Amsterdam: North Holland/American Elsevier, 1975.

Pennington, N. Cognitive components of expertise in computer programming: A review of the literature (Tech. Rep. No. 46). Ann Arbor: University of Michigan Center for Cognitive Science, July 1982.

Perry, D. K. Vocational interests and success of computer programmers. Personnel Psychology, 1966, 19, 517-524.

Petersen, C. G., & Howe, T. G. Predicting academic success in introduction to computers. Association for Educational Data Systems Journal, 1979, 12, 182-191.

Pitariu, H. Data concerning psychological selection of analyst programmers. Studia Psychologica, 1974, 16 (2).

Pitariu, H. Occupational selection of analyst-programmers and assistant-programmers. Revista de Psihologie, 1978, 21, 187-208.

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

Polya, G. How to solve it. New York: Doubleday-Anchor, 1957.

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

Ralston, A., & Shaw, M. Curriculum '78--is computer science really that unmathematical? Communications of the ACM, 1980, 23, 67-70.

Ramsey, H. R., & Atwood, M. E. Human factors in computer systems: A review of the literature (NTIS AD-AO75-679). Englewood, CO: Science Applications, 1979.

Ramsey, H. R., Atwood, M. E., & Van Doren, J. R. Flowcharts vs. program design languages: An experimental comparison. Proceedings of the 22nd Annual Meeting of the Human Factors Society, Santa Monica, CA. 1978.

Reed, S. Technology still a novice in classrooms. Section 12: "Employment outlook in high technology." The New York Times, March 27, 1983, p. 61.

Reitman, J. S., & Reuter, H. H. Organization revealed by recall orders and confirmed by pauses. Cognitive Psychology, 1980, 12, 554-581.

Resnick, L. B. Task analysis in instruction design: Some cases from mathematics. In D. Klahr (Ed.), Cognition and instruction. Hillsdale, NJ: Erlbaum, 1976, pp. 51-80.

Ricardo, C. M. Identifying student entering characteristics desirable for a first course in computer programming. Unpublished doctoral dissertation, Columbia University, Graduate School of Arts and Sciences, 1983.

Rich, C. A. A library of plans with applications to automated analysis (Tech. Rep. No. 294). MIT Artificial Intelligence Laboratory, 1980.

Rich, C. A. Formal representation for plans in the programmer's apprentice. Proceedings of International Joint Conference on Artificial Intelligence, Vancouver, BC, 1981.

Rich, C. A., & Shrobe, H. E. Initial report on a LISP programmer's apprentice. IEEE Transactions on Software Engineering, 1978, SE-4, 456-467.

Rich, C., & Shrobe, H. E. Design of a programmer's apprentice. AI: An MIT perspective. Cambridge, MA: MIT Press, 1979.

Rich, C., Shrobe, H., Waters, R., Sussman, G., & Hewitt, C. Programming viewed as an engineering activity (AI Memo No. 49). Cambridge, MA: MIT Artificial Intelligence Laboratory, 1978.

Rich, C., & Waters, R. C. Abstraction, inspection, and debugging in programming (AI Memo No. 634). Cambridge, MA: MIT Artificial Intelligence Laboratory, 1981.

Richards, V. G., Green, T. R. G., & Manton, J. What does problem representation affect: Chunk size, memory load, or mental process (Memo No. 319). England: MRC Social and Applied Psychology Unit, Sheffield University, 1979.

Ritch, P. A. A study of the Aptitude Test for Programmer Personnel as a predictor of success for students majoring in computer science and data-processing at the Chattanooga State Technical Institute (Doctoral dissertation, University of Tennesee, 1973). Dissertation Abstracts International, 1974, 35, 4647A. (University Microfilms No. 74-3862)

Robb, J. A. A study in the selection of predictors for success in electronic data processing courses. Proceedings of the Association for Educational Data Systems International Convention, 1976, 210-221.

Rogoff, B., & Gardner, W.P. Guidance in cognitive development: An examination of mother-infant instruction. In B. Rogoff & J. Lave (Eds.), Everyday cognition: Its development in social context. Cambridge, MA: Harvard University Press, 1983. In press.

Ross, P., & Howe, J. Teaching mathematics through programming: Ten year on. In R. Lewis & D. Tagg (Eds.), Computers in education. Amsterdam: North Holland, 1981.

Rouse, W. B. Systems engineering models of human-machine interaction. Amsterdam: North Holland, 1980.

Rumelhart, D. E. Schemata: The building blocks of cognition. In R. J. Spiro, B. C. Bruce, & W. F. Brewer (Eds.), Theoretical issues in reading comprehension: Perspectives from cognitive psychology, linguistics, artificial intelligence, and education. Hillsdale, NJ: Erlbaum, 1980.

Sackman, H. Man-computer problem solving. Princeton, NJ: Auerbach, 1970.

Schank, R. Dynamic memory. Cambridge: Cambridge University Press, 1982.

Schank, R., & Abelson, R. P. Scripts, plans, goals and understanding. Hillsdale, NJ: Erlbaum, 1977.

Schmidt, F. L., Gast-Rosenberg, I., & Hunter, J. R. Validity generalization results for computer programmers. Journal of Applied Psychology, 1980, 65, 643-661.

Schmidt, F. L., Hunter, J. E., McKenzie, R. C., & Waldrow, T. W. Impact of valid selection procedures on work-force productivity. Journal of Applied Psychology, 1979, 64, 609-626.

Schon, D.A. The reflective practitioner. Cambridge, MA: MIT Press, 1982.

Seidel, R. J., Anderson, R. E., & Hunter B. Computer literacy: Issues and directions for 1985. New York: Academic Press, 1982.

Seidman, R. H. The effects of learning a computer programming language on the logical reasoning of school children. ERIC Document.

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

Sheil, B. A. The psychological study of programming. Computing Surveys, March 1981(b), 13 (1).

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

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.

Shen, V. Y. The relationship between student grades and software science parameters. Proceedings of the 3rd International Computer Software and Applications Conference, 1979, 783-787.

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

Sheppard, S. B., & Love, L. T. A preliminary experiment to test influences on human understanding of software. Proceedings of the 21st Meeting of the Human Factors Society, Santa Monica, CA, 1977.

Shif, Z. I. Development of children in schools for mentally retarded. In M. Cole & I. Maltzman (Eds.), A handbook of contemporary Soviet psychology. New York: Basic Books, 1969.

Schroeder, M. H. Piagetian, mathematical, and spatial reasoning as predictors of success in computer programming (Doctoral dissertation, University of Northern Colorado, 1979). Dissertation Abstracts International, 1979, 39, 4850A. (University Microfilms No. 79-02855)

Seiler, J. Survey of validation studies on computer personnel selection instruments. Proceedings of the 5th Annual Conference of the Special Interest Group for Computer Personnel Research, 1967, 5, 43-51.

Seiner, P. Programmer Aptitude and Competence Test System (PACTS). Proceedings of the 9th Annual Conference of the Special Interest Group for Computer Personnel Research, 1971, 9, 3-25.

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

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

Shneiderman, B. Perceptual and cognitive issues in the syntactic/semantic model of programmer behavior. In W. Camm & R. E. Granda (Eds.), Symposium proceedings, human factors and computer science, 1978.

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

Shneiderman, B. Teaching programming: A spiral approach to syntax and semantics. Computers and Education, 1977(b), 1, 193-197.

Shneiderman, B., & Mayer, R. E. Syntactic/semantic interactions in programmer behavior: A model and experimental results. International Journal of Computer and Information Sciences, 1979, 7, 219-239.

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

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

Shneiderman, B., & McKay, D. Experimental investigations of computer program debugging and modification. Proceedings of the 6th International Ergonomics Association, 1976.

Shrobe, H. E. Dependency directed reasoning for complex program understanding (AI Tech. Rep. No. 503). Cambridge, MA: MIT Artificial Intelligence Laboratory, 1979.

Shrobe, H. E., Waters, R., & Sussman, G. A hypothetical monologue illustrating the knowledge of underlying program analysis (AI Memo No. 507). Cambridge, MA: MIT Artificial Intelligence Laboratory, 1979.

Shweder, R. A. Likeness and likelihood in everyday thought: Magical thinking and everyday judgments about personality. Current Anthropology, 1977, 18, 637-658.

Sime, M. E., Arblaster, A. T., & Green, T. R. G. Reducing programming errors in nested conditions by prescribing a writing procedure. International Journal of Man-Machine Studies, 1977, 9, 119-126.

Sime, M. E., Arblaster, A. T., & Green, T. R. G. Structuring the programmer's task. Journal of Occupational Psychology, 1977, 50, 205-216.

Sime, M. E., Green, T. R. G., & Guest, D. J. Psychological evaluation of two conditional constructions used in computer languages. International Journal of Man-Machine Studies, 1973, 5, 105-113.

Sime, M. E., Green, T. R. G., & Guest, D. J. Scope marking in computer conditions--a psychological evaluation. International Journal of Man-Machine Studies, 1977, 9, 107-118.

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, 1977, 9, 119-126.

Simon, H. A. The structure of ill structured problems. Artificial Intelligence, 1973, 4, 181-201.

Simon, H. A. Problem solving and education. In D. T. Tuma & F. Reif (Eds.), Problem solving and education: Issues in teaching and research. New York: Halsted Press, 1980.

Simon, H. A., & Hayes, J. R. The understanding process: Problem isomorphs. Cognitive Psychology, 1976, 8, 165-190.

Simon, D. P. & Simon, H. A. Individual differences in solving physics problems. In R. Siegler (Ed.), Children's thinking: What develops? Hillsdale, NJ: Erlbaum, 1978.

Simpson, D. The aptitudes of computer programmers. The Computer Bulletin, 1970, 14, 37-40.

Simpson, D. Psychological testing in computing staff selection--a bibliography. The Computer Bulletin, 1972, 16, 401-404.

Simpson, D. Aptitude testing of programmers. Computer Weekly, 1972, 13, 305.

Sinclair, H. Developmental psycholinguistics. In D. Elkind & J. H. Flavell (Eds.), Studies in cognitive development: Essays in honor of Jean Piaget. New York: Oxford University Press, 1969.

Skelton, J. L. Time-sharing vs. batch processing and teaching beginning computer programmer: An experiment. Association for Educational Data Systems Journal, 1972, 5, 91-97.

Slobin, D.I. Cognitive prerequisites for the development of grammar. In C. Ferguson & D. Slobin (Eds.), Studies of child language development. New York: Holt, Rinehart & Winston, 1973.

Slobin, D. I. (Ed.). Universals of language acquisition. Hillsdale, NJ: Erlbaum, 1982.

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. Cambridge, MA: Bolt Beranek and Newman, 1981.

Smith, M. Patsy's gift for spotting programming skill. Practical Computing, 1982, 5, 108-114.

Smith, N. L. Review of the Computer Programmer Aptitude Battery. In O. K. Buros (Ed.), The Eighth Mental Measurements Yearbook. Highland Park, NJ: Gryphon Press, 1978.

Soloway, E., Bonar, J., & Ehrlich, K. Cognitive strategies and looping constructs: An empirical study. Communications of the ACM, 1983. In press.

Soloway, E., & Ehrlich, K. Tacit programming knowledge. Proceedings of the Fourth Annual Conference of the Cognitive Science Society, Ann Arbor, MI, August 4-6, 1982.

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

Soloway, E., Lochhead, J., & Clement, J. Does computer programming enhance problem solving ability? Some positive evidence on algebra word problems. In R. Seidel, R. Anderson, & B. Hunter (Eds.), Computer literacy: Issues and directions for 1985. New York: Academic Press, 1982.

Soloway, E., Rubin, E., Woolf, B., Bonar, J., & Johnson, W. L. MENO-II: An AI-based programming tutor (Research Report No. 258). New Haven: Yale University, Department of Computer Science, December 1982.

Soloway, E., & Woolf, B. Problems, plans and programs. Proceedings of the 11th ACM Technical Symposium on Computer Science Education, 1980.

Spiro, R. J., Bruce, B. C., & Brewer, W. F. (Eds.). Theoretical issues in reading comprehension. Hillsdale, NJ: Erlbaum, 1980.

Stalnaker, A. W. The Watson-Glaser Critical Thinking Appraisal as a predictor of programming performance. Proceedings of the 3rd Annual Computer Personnel Research Group Conference, 1965, 3, 75-78.

Statz, J. Problem solving and LOGO. Final report of Syracuse University LOGO Project, Syracuse University, New York, 1973.

Stefik, M. Planning with constraints (MOLGEN: Part 1). Artificial Intelligence, 1981(a), 16, 111-140.

Stefik, M. Planning and metaplanning (MOLGEN: Part 2). Artificial Intelligence, 1981(b), 16, 141-170.

Stephens, L. J., Wileman, S., & Konvalina, J. Group differences in computer science aptitude. Association for Educational Data Systems Journal, 1981, 14, 84-95.

Sternberg, R. J., & Rifkin, B. The development of analogical reasoning processes. Journal of Experimental Child Psychology, 1979, 27, 195-232.

Steyn, D. W., & Hall, R. S. A description of the test battery for computer programmers. South African Computer Bulletin, 1971, 12, 14-17.

Strizenec, M. The psychological requirements of a computer programmer. Mech. Aut. Adm., 1975, 15, 492-493.

Swaine, M. Editorial: Women and computers. Infoworld, 1983, 5, 28.

Taylor, R. P., & Fisher, J. Information sources practicing programmers use to acquire new concepts and the relation between prior programmer education and concept impact. Computer Personnel, 1979, 8, 2-4.

Testa, C. J. A new approach to programming aptitude testing. Proceedings of the 11th Annual Conference of the Special Interest Group for Computer Personnel Research, 1973, 11, 49-61.

Thayer, R. H., Pyster, A. B., & Wood, R. C. Major issues in software engineering project management. IEEE Transactions on Software Engineering, 1981, SE-7, 333-342.

Tillman, M. An examination of the predictive validity of several potential predictors of the work proficiency of computer programmers. Computer Personnel, 1974, 5, 3-10.

Tinker, B. Logo's limits: Or which language should we teach? Hands On!, 1982, 6, 3-6.

Tobias, S. Overcoming math anxiety. New York: Norton, 1978.

Tomlinson-Keasey, C., & Keasey, C. The mediating role of cognitive development in moral judgment. Child Development, 1974, 45, 291-298.

Tuma, D. T., & Reif, F. (Eds.). Problem Solving and Education: Issues in Teaching and Research. Hillsdale, NJ: Erlbaum, 1980.

Tversky, D., & Kahneman, D. Science, 1980, 211.

U.S. Department of Labor. Cross-validation of the General Aptitude Test Battery and development of a weighted application blank for computer technology trainees. ERIC Document Reproduction Service, 1969, No. ED 068 546.

Van Der Burg, P. B., & Van Der Herik, H. J. The testing of programming skill. Informatie, 1980, 22, 790-794.

VanLehn, K. Bugs are not enough: Empirical studies of bugs, impasses and repairs in procedural skills. Xerox Cognitive and Instructional Sciences Series, March 1981, CIS-11.

Veldman, D. J. Review of the Computer Programmer Aptitude Battery. In O. K. Buros (Ed.), The Seventh Mental Measurements Yearbook. Highland Park, NJ: Gryphon Press, 1972.

Wallace, J. C. The selection and training of men and women programmers in a bank. Computers and Automation, April 1965, 23-25.

Walker, E., & Markham, S. J. Computer programming aptitude tests. Australian Psychologist, 1970, 5, 699-703.

Ward, W. C., & Jenkins, H. M. The display of information and the judgment of contingency. Canadian Journal of Psychology, 1965, 19, 231-241.

Waterman, D., & Newell, A. PAS-II: An interactive task-free version of an automatic protocol analysis system. Proceedings of the 3rd International Joint Conference on Artificial Intelligence, 1973.

Waters, R. C. A system for understanding mathematical FORTRAN programs (AI Memo No. 368). Cambridge, MA: MIT Artificial Intelligence Laboratory, 1976.

Waters, R. C. Automatic analysis of the logical structure of programs (AI Tech. Rep. No. 492). Cambridge, MA: MIT Artificial Intelligence Laboratory, 1978.

Waters, R. C. A method for analyzing loop programs. IEEE Transactions on Software Engineering, 1979, SE-5, 237-247.

Waters, R. C. The programmer's apprentice: Knowledge based program editing. IEEE Transactions on Software Engineering, 1982, SE-8 (1).

Watt, D. H. A comparison of the problem solving styles of two students learning LOGO: A computer language for children. Proceedings of the National Educational Computing Conference, 1979, 255-260.

Watt, D. H. Logo in the schools. Byte, 1982, 7, 116-134.

Weinberg, G. The psychology of computer programming. New York: Van Nostrand, 1971.

Weir, S. LOGO as an information prosthetic for the handicapped (Working Paper No. WP-9). Cambridge, MA: MIT, Division for Studies and Research in Education, May 1981.

Weir, S., & Watt, D. LOGO: A computer environment for learning-disabled students. The Computer Teacher, 1981, 8, 11-17.

Weizenbaum, J. Computer power and human reason: From judgment to calculation. San Francisco: Freeman, 1976.

Wells, G. W. Relationship between the processes involved in problem solving and the process involved in computer programming. Dissertation Abstracts International, 1981, 42, 168 pp.

Werner, H. The concept of development from a comparative and organismic point of view. In D. R. Harris (Ed.), The concept of development. Minneapolis: University of Minnesota Press, 1957.

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

Wileman, S. et al. Influencing success in beginning computer science courses. Journal of Educational Research, 1981, 74, 223-226.

Wilkinson, A. An analysis of the effect of instruction in electronic computer programming logic on mathematical reasoning ability (Doctoral dissertation, Lehigh University, 1973). Dissertation Abstracts International, 1973, 33, 4204A. (University Microfilms No. 73-4290)

Willoughby, T. C. Are programmers paranoid? Proceedings of the 10th Annual Computer Personnel Research Group Conference, 1972, 47-54.

Willoughby, T. C. Current perspectives in selection testing. Proceedings of the 9th Annual Conference of the Special Interest Group for Computer Personnel Research, 1971, 9, 54-74.

Winkler, C. The Computer Careers Handbook. New York: ARCO. To appear in 1983.

Wirth, N. On the composition of well-structured programs. Computing Surveys, 1974, 6, 247-259.

Wolfe, J. M. An interim validation report on the Wolfe Programming Aptitude Test (Experimental form S). Computer Personnel, 1977, 6, 9-12.

Wolfe, J. M. Perspectives on testing for programming aptitude. Proceedings of the Annual Conference of the Association for Computing Machinery, 1971, 268-277.

Wolfe, J. M. Testing for programming aptitude. Datamation, April 1969, 67-72.

Wolfe, J. M. Validation report. Computer Personnel, 1974, 5, 15-16.

Wolfe, J. M. The Wolfe Programming Aptitude Test (School Edition). Proceedings of the 9th Annual Conference of the Special Interest Group for Computer Personnel Research, 1971, 7, 180-187.

Wright, P., & Reid, F. Written information: Some alternatives to prose for expressing the outcomes of complex contingencies. Journal of Applied Psychology, 1973, 57, 160-166.

Young, R. M. The machine inside the machine: Users' models of pocket calculators. International Journal of Man-Machine Studies, 1981, 15, 51-85.

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

Zabarenko, L., Badger, G. F., & Williams, E. B. TABRA: A projective test for computer personnel research, preliminary report. Proceedings of the 8th Annual Conference of the Special Interest Group for Computer Personnel Research, 1970, 8, 92-107.