ED 295 618 IR 013 324

AUTHOR Perkins, David; Martin, Fay
TITLE Fragile Knowledge and Neglected Strategies in Novice Programmers. IR85-22.
INSTITUTION Educational Technology Center, Cambridge, MA.
SPONS AGENCY National Inst. of Education (DHEW), Washington, D.C.
PUB DATE Oct 85
CONTRACT 400-83-0041
NOTE 35p.; For a related report, see IR 013 327.
PUB TYPE Reports - Research/Technical (143)

EDRS PRICE MF01/PC02 Plus Postage.
DESCRIPTORS Interviews; *Knowledge Level; *Problem Solving; *Programing; *Questioning Techniques; Secondary Education; *Teaching Methods
IDENTIFIERS Fragile Knowledge

ABSTRACT

As part of an ongoing program of research to identify the difficulties encountered by novice programmers and to develop teaching strategies to help them overcome these obstacles, researchers employed a scaffolded interview procedure with 20 high school students enrolled in the second semester of a year-long BASIC course. Investigators presented each student with a sequence of eight programming problems, ranging from easy to difficult. They asked questions to track student thinking and intervened in student difficulties with graduated levels of assistance: first, general prompts to provoke strategic thinking; second, hints, leading questions, and bits of information; and third, exact solutions to the immediate dilemma. Results showed that student difficulties stem from knowledge that is fragile in several ways, i.e., partial knowledge, inert knowledge, lack of a critical filter, misplaced knowledge, and conglomerated knowledge. Findings indicate that novice programming students might benefit from explicit teaching of strategies for controlled exploration as part of their instruction in beginning programming. Explicit teaching of strategic skills is a promising way to help students gain control of the programming process and appreciate the need for precision in understanding and using programming commands. (27 references) (Author/MES)

85-22

ED295618

FRAGILE KNOWLEDGE AND NEGLECTED STRATEGIES IN NOVICE PROGRAMMERS

Technical Report

October 1985

Educational Technology Center
Harvard Graduate School of Education
337 Gutman Library Appian Way Cambridge MA02138

IR013334

FRAGILE KNOWLEDGE AND NEGLECTED STRATEGIES IN NOVICE PROGRAMMERS

Technical Report

October 30, 1985

Written by:

David Perkins

Fay Martin

Programming Group:

Betty Bjork	Marie Salah
Chris Hancock	Nancy Samaria
Renee Hobbs	Paul Shapiro
Jack Macleod	Rebecca Simmons
Fay Martin	Tara Tuck
David Perkins	Martha Stone Wiske

We gratefully acknowledge the collaboration with teachers and school systems that made possible the research reported here. Preparation of this research report was supported in part by the National Institute of Education (contract # NIE 400-83-0041). Opinions expressed herein are not necessarily shared by NIE and do not represent Institute policy.

Abstract

Many students have great difficulty mastering the basics of programming. Inadequate knowledge, neglect of general problem-solving strategies, or both might explain their troubles. We report a series of clinical interviews of students taking first year BASIC in which an experimenter interacted with students as they worked, systematically providing help as needed in a progression from general strategic prompts to particular advice. The results indicate a substantial problem of "fragile knowledge" in novices -- knowledge that is partial, hard to access, and often misused. The results also show that general strategic prompts often resolve these difficulties. Recommendations for teaching more robust knowledge and general strategics are made. Implications for the impact of programming on general cognitive skills are considered.

Fragile Knowledge and Neglected Strategies in Novice Programmers

Plentiful evidence speaks to the difficulties encountered by beginning students of programming in primary and elementary school. Linn (1985) reported only modest achievement in BASIC in most schools in a study ranging over a number of school systems. Research conducted by Pea and Kurland documented the minimal competency achieved by students learning Logo under relatively nondirective conditions (Pea & Kurland, 1984a, Pea & Kurland, 1984b). Perkins, Hancock, Hobbs, Martin, and Simmons (in press) have discussed patterns of behavior displayed by many novice programmers as part of their partial mastery, patterns such as haphazard tinkering where a student attempts to repair a buggy program by a series of almost arbitrary changes. While reasonably successful efforts to teach youngsters programming have been reported from time to time (Clements, 1984; Clements & Gullo, 1984; Linn, 1985; Nachmias, Mioduser, & Chen, 1985), typical results at the primary and secondary level seem disappointing and call for efforts to understand the nature of the difficulties and remedy them.

One natural approach to defining the difficulties asks what students typically know and what they do not. Do they understand the basic operations of the computer language in question? Do they possess problem solving strategies for managing an attack on a problem, as has proved important in mathematical problem solving for example (Polya, 1954, 1957; Schoenfeld, 1982; Schoenfeld & Herrmann, 1982)? Do they have in their repertoires prototypical programming plans such as nested FOR-NEXT loops or recursion for dealing with particular types of situations (Anderson & Reiser, 1985; Soloway & Ehrlich, 1984)? Do they know the symptoms that signal when a particular operation or programming plan might serve? One can speak roughly of a continuum between low-level knowledge of the particular commands a language offers and rather abstract and general tactics of problem solving. With such a continuum in mind, is the shortfall principally in low-level knowledge or high-level strategic repertoire?

The question has some interest not only for the pedagogy of programming in itself but for the possibility explored by many that learning to program might impact on the learner's general cognitive skills (e.g. Feurzeig, Horwitz, & Nickerson, 1981; Linn, 1985; Papert, 1980). Consider two extreme cases. Perhaps novice programmers have ample general cognitive skills for the relatively easy problems they face, but their mastery of the primitives of the language is so poor that they cannot apply those skills effectively to solve programming problems. In this case, typical instruction in programming cannot be expected to boost learners' cognitive skills until it carries students beyond the initial phases of learning into problems that pose higher order challenges. On the other hand, perhaps novice programmers have the details of the language under control but lack the general cognitive skills required to marshall their knowledge. In this case, from the first programming affords a natural training ground for the development of those higher order skills, although it may be questioned whether students will develop them without strong mediation (Delclos, Littlefield, & Bransford, 1985; Perkins, 1985; Salomon & Perkins, this issue).

The broad issues seem clear enough. However, in pursuing them one has to recognize a certain oversimplification in the questions: Knowledge comes across as a "you have it or you don't" sort of thing. The student may know some things about the language, but not other things and perhaps not enough. However, common experience testifies that often a person does not simply "know" or "not know" something. Rather, the person sort of knows, has some fragments, can make some moves, has a notion, without being able to marshall enough knowledge with sufficient precision to carry a problem through to a clean solution. One might say that learners in such a state have fragile knowledge.

Understanding in what ways students' knowledge of programming might be fragile -- neither here nor there, reliable nor random, possessed nor lost -- could help in grasping the nature of students' difficulties and designing instruction that affords better learning opportunities. In particular, under the general label of fragile knowledge a number of questions can be addressed. Do students have knowledge of operations they do not succeed in retrieving? Is the problem of retrieval radical, or can strategic prompts trigger retrieval? Is it that students do not have certain knowledge or do they get confused about what knowledge to use when?

In this paper, we report on a series of clinical case studies done with high school students taking a first year BASIC course. Working one-on-one with a number of students, the experimenter observed and interacted in defined ways as the students attempted to solve programming problems. While a few students managed the problems well, most evinced considerable difficulty at some points. We interpret the students' difficulties as manifesting fragile knowledge of BASIC commands in four senses. Partial knowledge is the straightforward case of an impasse due to knowledge the student has not retained or even never learned, as revealed by clinical probes failing to reveal signs of the knowledge. Inert knowledge refers to situations where the student fails to retrieve command knowledge but in fact possesses it, as revealed by a clinical probe. Misplaced knowledge designates circumstances where a student imports commands structures appropriate to some contexts into a line of code where they do not belong. Conglomerated knowledge signifies situations where a student produces code that jams together several disparate elements in a syntactically or semantically anomalous way in an attempt to provide the computer with the information it needs. The four will be more sharply defined and distinguished and their connections to the literature examined as they are discussed in turn.

We also made tallies of certain fragile knowledge events that show to what extent general strategic prompts helped students over their difficulties. This, along with some features of the case studies, allows appraising the relative contributions of fragile knowledge and lack of general problem-solving skills to students' difficulties and points to prospects for a better pedagogy of programming.

A Clinical Methodology

To investigate the locus of novices' programming difficulties, we devised a procedure that would reveal whether particular difficulties reflected the failure of high-level problem management skills or a poor understanding of particular commands in the programming language. The experimenter presented a student with a choice of eight increasingly difficult programming problems. The problems built on one another, each preparing the way for the next. Each student selected a problem that seemed manageable, attempted it in interaction with the experimenter until completing it, and proceeded to the next problem, continuing this process until the end of the session. Some details follow.

Subjects. Twenty high school subjects enrolled in the second semester of a year-long first BASIC course participated in the study. The students ranged from 10th to 12th graders, including 11 girls and 9 boys. Each student participated for one 45 minute session. In our view, the instruction at the site was quite careful and conscientious, from a teacher with a very good understanding of BASIC in particular and programming in general. Nonetheless, while some of the students had developed considerable programming skills others displayed substantial difficulties, as will emerge.

Programming tasks. The sequence of eight programming tasks, ranging from easy to difficult, centered on the FOR-NEXT loop. All the problems asked for programs that produced patterns of stars (asterisks) on the screen. For example, problem 1 called for a program to produce a column of ten stars; problem 3 called for a program that would ask for a number and then print a column with that many stars; problem 4 did the same except that the row of stars was to be horizontal. Problem 5 required a program that asked for a number and then produced a square of stars. For instance, with an input of 5 the program would produce:

```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

Problem 8 called for a hollow square of stars of any size.

It should be noted that the exact character of the challenge posed by these problems depends somewhat on the programming environment and the commands students know. In particular, we designed the sequence knowing that the students in our sample did not have at their disposal cursor control commands and consequently needed to produce the patterns of stars through print statements, line by line from the top to the bottom of the pattern. This prevents, for example, solving the hollow square of stars problem by way of a program that guides the cursor around the sides of the square, printing asterisks along the way.

Procedure. The experimenter explained the purpose of the study -- to come to understand how people learn programming, what difficulties they have, and how to help people to overcome them -- and explained that the experimenter would watch and help as the student worked some programming problems. The experimenter introduced the student to the sequence of problems and invited the student to choose one to begin with that would be "challenging, not too easy, but not too hard."

The experimenter watched and asked occasional questions to track the student's thinking until and if the student encountered a significant difficulty. Then the experimenter intervened, asking questions and providing information to help the student to overcome the difficulty. The experimenter worked with the student until the student attained a program that performed the task in question. Then the experimenter asked the student to attempt the next problem, and so on until time ran out.

When the student faced an impasse, the experimenter's first questions, called prompts, were high-level strategic questions one might ask oneself. By definition, prompts were questions that did not require any foreknowledge of the true nature of the difficulty: People in principle could prompt themselves. Some typical prompts were, "What's the first thing you need to tell the computer to do; how would you describe the problem to yourself; what does this (e.g., a semicolon) do?" As the examples suggest, some prompts were phrased generally and could be used in any problem-solving situation, even one outside of programming, while others were particularized to mention semicolons or other elements of the programming situation. The prompts were generated by the experimenter according to the experimenter's judgment of the level of specificity needed.

If a couple of prompts did not help the student to overcome the difficulty, the experimenter resorted to "hints." Hints by definition reflected the experimenter's understanding of the solution, nudging the student toward a resolution with leading questions or bits of information. Some characteristic hints would be, "Can you think of a command to get the computer to ask you for a number; your problem is to repeat something several times, so do you know a command for that; why don't you try a semicolon?"

If a couple of hints did not provoke progress, the experimenter provided an exact solution to the immediate dilemma so that the student could get on with the rest of the program. These were called "provides." Characteristic provides were: "Write INPUT 'How many stars per side' N; use a FOR-NEXT loop; put a semicolon after the print statement." The experimenter did not just provide answers, of course, but also attempted to explain them.

The escalation from prompts to hints to provides not only helped the student but served as a probe of the student's level of mastery and understanding. The more support the student required, the less the student could accomplish solo. A successful prompt suggested that the student could succeed by learning self-prompting strategies -- good questions to ask oneself in programming situations. At the other extreme, a provide preceded by several unsuccessful prompts and hints indicated very limited knowledge and understanding relevant to the particular difficulty.

Although the experimenter generally attempted the progression from prompts to provides, sometimes the experimenter moved directly to hints or provides. This occasionally happened by mistake, but more often because the general performance of the student and spiraling frustration suggested that more direct help was needed to keep the student's attention and involvement.

Data collection. The data collected during a session included notes taken by the experimenter, code written by the student and transcribed by the experimenter, and an audiotape of the conversation. The audiotape was transcribed later and notes and code interpolated to yield a verbatim protocol of the session. Case studies were drawn from the protocols; a number of examples will be discussed below. In addition, the protocols were scored for certain events as discussed later.

We turn now to a discussion of particular phenomena of fragile knowledge that occurred frequently throughout the clinical interviews.

Partial Knowledge

Knowledge might be fragile in many ways, each telling us something about students' shortfalls of understanding and pointing to ways to remedy them. Perhaps the simplest sort of fragile knowledge is partial knowledge: A student knows something about a command or other element of programming but has minor gaps that impair the student's functioning. Since this sort of fragility is so straightforward, we will not treat it extensively but simply mention a couple of examples.

The programming tasks that asked the students to produce one or more horizontal lines of stars of varying length all called for the use of a semicolon at the end of a PRINT "*" to suppress the usual carriage return. While some students recalled this tactic spontaneously or upon prompts or hints, others found themselves at a loss. When the experimenter provided the semicolon, some even showed no familiarity with its function although all the students had been exposed to it in their class.

For a more advanced example, the problems calling for multiple rows of stars required a bare PRINT statement after the NEXT of the inner FOR loop to force a carriage return after each line of stars. The students in their class had used bare PRINT statements to create blank lines in formatting output. However, apparently many associated a bare PRINT with blank lines specifically, not recognizing its general function of outputting a carriage return, which could also be used to terminate a line.

In general, numerous examples suggestive of partial knowledge occurred throughout the interviews. If this were the only sort of difficulty students manifested, there would be little point of speaking of fragile knowledge: One [illegible] ust as well refer to partial knowledge specifically. However, the [illegible] flicting students generally proved more complex. To focus only on [illegible] wledge would be both to miss much about the structure of students' knowl[illegible] and to underestimate how much knowledge they have. With this in mind, we turn to other species of fragility.

Inert Knowledge

One particularly straightforward kind of fragility has been called "inert knowledge." This refers to knowledge that a person has, but fails to muster when needed. For example, Bereiter and Scardamalia (1985) discuss the problem of inert knowledge in the context of writing. They note that youngsters asked to write on a topic typically only manage to access a fraction of their relevant knowledge. Conventional tactics of fluency such as brainstorming ideas seem to offer little help; however the strategy of listing words that might be used in an essay considerably increases students' retrieval of relevant information. The authors suggest that this occurs because the bare terms activate a network of associations more effectively than lists of points, which have a more particular nature.

Broadly speaking, the problem of inert knowledge is a problem of transfer. Knowledge acquired on one occasion fails to bridge the slight or substantial gap to another occasion of application. Belmont, Butterfield, and Ferretti (1982) emphasize that knowledge often tends to remain bound to the context of initial learning unless the learner deploys self-monitoring strategies that help to carry the knowledge across to other applications. Salomon and Perkins recently have presented a general theory of the mechanisms of transfer that identifies both "high road" and "low road" ways that transfer can occur. The former requires mindful abstraction and application in new contexts, the latter skills practiced to near automaticity on a variety of cases, so that new contexts spontaneously evoke the skills in question (Perkins & Salomon, in press; Salomon & Perkins, 84; Salomon & Perkins, this issue). In typical instructional situations, neither the conditions for low nor high road transfer are met. Consequently, knowledge that otherwise might serve the learner in a new context remains inert.

In sum, the phenomenon of inert knowledge occurs routinely throughout much of learning. Identifying situations where it does mischief helps to define what remedy to apply: Students may need their knowledge represented more generally and better tools of retrieval and abstraction to access knowledge and break it free from narrow contexts. But how does all this apply to the context of programming specifically? In our methodology, an instance of inert knowledge appears when a learner fails to respond with an appropriate solution, but a prompt or hint triggers success, demonstrating a problem of access as opposed to a problem of ignorance.

Consider these examples, for instance. Brenda was working on a program to print a column of ten stars. She had coded:

```
10  X = "*"
20  FOR X = 1 TO 10
30  PRINT X
40  NEXT X
```

When she ran the program, she received the error message, "Type mismatch in 10." She asked what a type mismatch was and the experimenter directed her to look at line 10.

E: What kind of symbol is the star, a number or a character?

Brenda: A number, no a character.

E: Okay, and what is X? What does it stand for?

Brenda: Oh, a number.

She then recoded: 10 X$ = "*". So the experimenter's hint lead Brenda to retrieve knowledge that in fact she possessed but had not accessed.

Dennis was working on the more advanced problem 5, which called for a solid square of stars. He had coded two nested FOR loops but his output resulted in a horizontal row of stars. Dennis pondered his output for a while.

E: How many stars did you get?

Dennis: Twenty-five. That's the right number.

E: What do you need to do?

Dennis: Put them into a block.

E: Right.

Dennis then coded a bare PRINT after the first NEXT statement to force a carriage return and make rows of stars. Here the experimenter's general queries led Dennis to see through to the nature of the problem and retrieve a command that would solve it.

These examples illustrate inert knowledge that when accessed provides what the student needs for a solution. But another part of programming skill involves a critical filter that allows the student to reject candidate solutions. We term the activity of reading back expressions in a computer language to discern exactly what they tell the computer to do "close tracking" (Perkins, Hancock, Hobbs, Martin, & Simmons, in press). There is a strategic side and a low-level knowledge side to close tracking. On the strategic side, students need to attempt to close track in order to apply the critical filter. On the knowledge side, even when a student tries to close track, problems of fragile knowledge can stand in the way.

For an example of fragile knowledge, Abby began a program to print out a column of N stars this way:

```
3  INPUT "How many stars do you want"; N
4  PRINT
5  N = 8
10 FOR X = 1 TO N
```

When she ran the program with an input of 5, she could not figure out why she got 8 stars. The experimenter then asked her to describe what the program did line by line.

> Abby: Okay, at line 3 it's going to input how many stars do you want and then it's going to stop so I can put it in. On the next line it's going to skip a line 'cause of the print. Then on number 5 I'm telling it how many I want, the little stars. I'm telling it how many stars I want to print out, and the next line is the loop.

Even in reading through the program, Abby did not realize that her assignment of 8 to N would overwrite her input. Yet this would appear obvious once pointed out. Odd as such slips may seem, they proved quite common, preventing students from filtering out their errors by close tracking.

With these examples in mind, what appears to explain the occurrence of inert knowledge in programming? One obvious cause is the failure to execute certain strategic actions that marshall particular knowledge. A programmer may fail to close track an expression to check it, for example. On the other hand, even when students ask themselves appropriate higher order questions, the retrieval process may fail for any number of reasons. Abby, for instance, seemed to be answering what each line did by reading in her intention to get N set to a reasonable value rather than thinking about the exact actions prescribed by the code. For another instance, a student failing to retrieve a bare PRINT to force a carriage return at the end of a row of stars may know in principle that such a print statement outputs a carriage return but associate the action strongly with making blank lines. In general, one should recognize that an experienced programmer will have a rich network of associations linking various commands and programming plans; if retrieval does not succeed by one route, it will probably succeed by another. In contrast, the network of connections in the novice inevitably is sparse; if retrieval by one route fails for whatever reason, there may be no other ready way.

Misplaced Knowledge

Another phenomenon of fragility might be called misplaced knowledge. Here, knowledge suitable for some roles invades occasions where it does not fit. Like inert knowledge, misplaced knowledge occurs commonly in human experience. For instance, one's steering habits lead to trouble when one's car skids, since the best corrective calls for steering with rather than against the direction of the swerve. Toddlers frequently overgeneralize the application of new words and individuals learning a second language experience interference from terms and syntactic structures in their native language (de Villiers & de Villiers, 1978). Functional fixedness, where people have difficulty in applying an object in an unconventional way, and the classic Einstellung effect, where problem solvers carry forward a solution method for a series of problems to new problems allowing a much simpler solution, offer other examples (Adamson, 1952; Luchins, 1942). As with inert knowledge, the connection with transfer should be plain. All these examples amount to instances of negative transfer, where the knowledge or know-how in question impairs rather than abets performance through application in an unsuitable context.

Misplaced and inert knowledge display another connection: The one can cause the other. In the previous section, we limited our discussion to cases where fairly pure problems of retrieval left knowledge inert. However, often relevant knowledge remains inert because misplaced knowledge has intruded. For example, Stan began working on the program to print a vertical column of N stars. He coded a FOR loop and then paused, pondering how to handle the print statement. When asked about his worry, he said he wanted to set up the format line in order to "print out the stars the way you want it." He was referring to formatted printing with the print using command. The experimenter quickly steered him away from this cumbersome method.

E: "What if you didn't use a format. Is there any other way you can think of to print a star?

Stan: Print and then just write an asterisk.

For another example, Dan was working on the program to print out a row of N horizontal stars. He coded:

```
10  INPUT "How many stars do you want?"; S
15  LET S$ = "*"
20  PRINT S STEP 2
```

When asked how the program would work, he explained that STEP 2 would print across so that somehow the star would be printed S times horizontally. Clearly Dan misplaced the STEP command from a FOR loop. Moreover, he was unable to retrieve the FOR loop itself even after the experimenter hinted. When the experimenter prompted him to think of another way to print across, he did produce PRINT S$;, but still no FOR loop. Moreover the semicolon also migrated inappropriately: When the experimenter directly suggested using a FOR loop, Dave coded 12 FOR X = 1 TO S;, with the semicolon at the end of the FOR statement.

As with inert knowledge, we should ask what explains the occurrence of misplaced knowledge. Overgeneralization -- or underdifferentiation to put it another way -- provides one obvious cause. Recall how Dan had not sharply differentiated the proper applications of STEP or the semicolon. Recency of learning often figured in such difficulties, the students apparently feeling that whatever they had studied lately must somehow apply. This was probably the cause of Stan's misapplication of print using. Throughout the course of data collection, we noted how different intrusions seemed to crop up as a function of topics in class. Early in our observations, we saw print using and read data statements appear in students' code. A few weeks later the step command was interpreted as causing both horizontal and vertical printing, somehow always in association with print statements rather than loops. In the final sessions, we began to see students try to apply arrays in coding the square designs.

In situations where the learner has difficulty finding a reasonable solution, misplaced knowledge may amount to a desperation measure. Precedent for this appears in mathematical and other problem solving contexts, where the general point has been made that students generally try to provide some kind of a response, even if a dubious one (Davis, 1984). For example, Alice was having great difficulty programming the vertical column of ten stars. She realized that she needed to print a star ten times going down, but could not retrieve an appropriate command for doing so. Her first idea for repeating involved the use of a GOSUB, but the experimenter steered her toward the idea of a FOR loop. Then to take care of the printing she suggested using READ, but realized there were no data to read. Finally, she coded a print statement inside the loop but become stuck over whether to code ten print statements instead of just one. Ultimately the experimenter had to provide Alice with the correct code. It seems fairly clear that in this instance Alice's misplaced knowledge reflected her being at a loss.

Conglomerated Knowledge

Another manifestation of fragile knowledge might be called "conglomerated knowledge." This appears when the young programmer composes code that expresses loosely the intent without following the strict rules that govern how the computer actually executes code. Dan's use of STEP in the midst of a PRINT statement offers an example not only of misplaced but also conglomerated knowledge.

Another example independently produced by several students concerned the problem of printing five stars in a horizontal row. In one episode, Gail coded:

```
10  INPUT "How many stars do you want"; X
20  PRINT "*"; X
```

Her idea was that the star would print ten times across.

This example illustrates well the peculiar character of conglomerates. They certainly show signs of the programmer's mindful engagement in the activity of programming; the programmer plainly has sought to encode information the computer would need to carry out the task in question. On the other hand, conglomerates are syntactically or semantically ill-formed. Either they are far from being legal code in the language in question, as with STEP in the midst of PRINT, or although accidentally legal direct the computer to do something very different from the programmer's intent, as with PRINT "*"; X to print a row of X stars.

Consider another more complex example. Ellie was working on the same problem, but she coded:

```
 1  FOR X = 1
10  "Print how many stars do you want"; N
20  INPUT NUMBER
30  X = (*) * NUMBER
40  NEXT X
```

Her conglomerate aimed to multiply the asterisk times whatever number she put in. There are several other difficulties with her program as well, of course.

Of course, there is no sharp borderline between misplaced and conglomerated knowledge. Many conglomerates involve misplaced knowledge -- elements from one programming construct or context showing up in the midst of an expression from somewhere else entirely. But a broad distinction can be drawn. Pure cases of misplaced knowledge involve knowledge intruding into contexts without any sense of a conglomerate jammed together out of ill-fitting parts, as in the earlier example of seeking to use a print using statement in a situation calling for a simple print. Pure cases of conglomerates occur when the expression in question does not show components obviously misplaced from some other context. For instance, in the PRINT "*"; X example, one does not particularly feel that the X is misplaced from anywhere, a FOR-NEXT loop for example. Rather, the programmer simply means to let the computer know that X stars are required and hopes that putting the X in the print expression will do so.

As with inert and misplaced knowledge, one wants an explanation for conglomerates. Why do they occur? Plainly they reflect the active effort of young programmers to solve the problem. Unsure exactly how to command the computer, a programmer takes a stab at it, putting together code that provides the computer with at least some of the information the computer would need to perform the task. The remaining question asks why programmers take such stabs rather than doing the "right thing?" Several answers seem relevant. First of all, the "right thing" often involves knowledge inert or not possessed at all, leaving the programmer no proper recourse. Second, the programmer often works from an underdifferentiated knowledge base, leading to misplacements that yield conglomerates. Third, the programmer fails to close track tentative conglomerates or may be unable to do so with precision. Fourth, the programmer lacks the general critical sense that one simply cannot expect to throw things together in a programming language and have them work. That is, the programmer treats the programming language as much looser, less restrictive, more expressive and more like a natural language than in fact it is.

Prompts as a Gauge of Strategic Shortfall

One of our running themes in this paper has been the contribution to programming of relatively high level strategic knowledge versus relatively low level knowledge of the details of the language. Successful prompts in particular show that a student possesses knowledge the student might have retrieved and applied autonomously. In other words, prompts are the high level strategic questions one might ask oneself. To the extent that a student needs help but proves responsive to prompts, the student displays a strategic shortfall but sufficient lower level mastery. How often this happens will be taken up later. Here we examine effective prompts and consider a few cases in more detail to convey their flavor.

Particularly notable is the range of prompts. Here are some samples selected for variety: What's the first thing you need to tell the computer to do? Are there any other ways to make the computer (print across, repeat something, whatever)? How would you describe the problem to yourself? What is your plan? Do you know a command for repeating (after the student has indicated a need to repeat; otherwise this would be a hint)? What do you need to do next? What does a semicolon do (when the student is reading back a statement with a semicolon; otherwise this would be a hint)? Note how much these prompts vary in seeming generality and yet how alike they are. While some are phrased much more specifically than others, these bind a very general question to particular circumstances. For instance, "What does a semicolon do?" is a special case of "What does _this_ do?" where the _this_ might be a symbol in an algebraic equation or a part of a carburetor. In type, the question is a probe for the exact function of a part.

Of course, whether a prompt succeeds depends not just on the prompt itself but on the accessibility and organization of the knowledge the prompt seeks to activate. The very same prompt can succeed on one occasion and fail on another. For example, in attempting the filled-in square program, a number of students coded only one FOR loop and two PRINTs, one with a semicolon "to go across" and another without "to go down." Through a series of prompts, the experimenter usually got the students to realize that the program needed to repeat N rows of N stars per row. But this was not always enough. When asked, "Do you know a command that might help you repeat something," Dennis, for example, replied, "Oh yeah, FOR-NEXT." But Randy's response was "GOTO."

We have saved one type of prompt for separate discussion because it occupies a pivotal position in programming: The prompt to close track, which may be seen as a special case of the prompt to check one's work in relation to one's objectives. As mentioned earlier, close tracking ideally functions as a critical filter applied to candidate solutions. While some students had difficulty close tracking with precision when prompted to do so, others succeeded. For example, Naomi was working on a programming problem from our sequence that asked for a triangle of stars. She wrote a program involving nested loops but no use of the iteration variable of the first loop as the upper limit of the second, to yield lines of increasing length. But even before running the program she paused, pondered her program for a moment, and realized, "It's going to end up like a square." Naomi's inclination and ability to read back her program and forecast what it would do, rather than presuming it would do as she intended, enabled her to reject plans and seek other alternatives. This was relatively rare among the students.

In addition to forecasting, the critical filter of close tracking can serve as a debugging aid. One can adopt the general strategy of close tracking the whole program or likely segments of it not just to predict what the program will do but to try to explain bugs: Why did the program misbehave in exactly this way? Often students prompted to attempt this strategy managed to follow through. For example, Dick was working on the filled-in square problem with one loop and two print statements, PRINT "*"; to print across and PRINT "*" to print down. When he ran his program he got a column of pairs of stars. The experimenter prompted him to trace why that result occurred. Dick spent a few moments contemplating the program and then replied, "It prints that and that and then goes down, prints that, then that, goes down, prints that and that." The experimenter asked what Dick needed to get the desired result. Dick: "Keep on printing that row. So after it prints a row, it goes on to print the same row, that many times." Dick has reached a sharper formulation of what must happen.

The previous examples show how gauging the effectiveness of prompts allows appraising whether students would be helped by an enriched strategic repertoire. But up to now we have not provided any information on how often prompts help. We turn now to this question.

How Often Do Prompts Help?

A means of identifying and tallying prompts, hints, and provides was developed. Whenever a student's protocol showed that the experimenter needed to intervene significantly, a scorer assessed whether the experimenter offered prompts, hints, and/or provides, and which led to a correct resolution. A second scorer judged independently a subset of all the protocols with high interscorer agreement.

As mentioned earlier, the experimenter from time to time gave a hint without a prior prompt or a provide without a prior hint. Sometimes this occurred by mistake but most often because the student appeared to need immediate support. Assume for the moment that all provides and all successful hints, even those not preceded by hints and prompts respectively, were genuinely needed by students. This yields a conservative estimate of the percentage of time prompts and hints are effective. In particular, prompts led to a correct resolution of difficulties 32% of the time and hints an additional 17%, leaving 52% of difficulties requiring an answer provided by the experimenter.

Now consider only those 'provides' preceded by unsuccessful hints and successful hints preceded by unsuccessful prompts. These data include just the hints and provides that were surely necessary. One can then calculate a liberal estimate : the percentage of time prompts and hints are effective. Prompts led to a correct resolution 55% of the time and hints 16% of the time, leaving 28% to be provided.

The ambiguity introduced by the occasions the experimenter did not proceed regularly from prompt to hint to provide should not obscure the central point. Plainly, prompts have a substantial impact on students' thinking. From a third to half the time they help the student to marshall knowledge and resolve the difficulty at hand. This suggests that young students of programming could benefit from more strategic thinking in the form of general strategic questions to ask themselves.

It is also worth noting that hints, although providing information directly relevant to the difficulty, did not help that much over and above prompts. By both ways of estimating, hints resolved a difficulty only about 15% of the time. This suggests that if the knowledge is there to be marshalled, strategic questions usually suffice to marshall it. Finally, the considerable percentage of provides demonstrates a substantial problem of partial knowledge and of inert knowledge that could not be provoked by prompting and hinting.

These remarks apply to all the experimenter's prompting except prompts to close track, which we isolated for separate analysis in light of the importance of close tracking as a critical filter. When students close tracked, they did so accurately about 50% of the time, forecasting a problem or explaining a bug. However, only about 20% of the episodes of close tracking were spontaneous; the rest had to be prompted. This almost certainly underestimates the frequency of spontaneous close tracking, since we only tallied episodes where students were plainly close tracking; a couple of seconds staring at the program with no tracking-like comments would not be counted, although the student might have been close tracking rather than just generally looking over the program. Nonetheless, the numbers suggest that students do not spontaneously close track as often as they might benefit from it.

Issues of Knowledge and Strategy

We began this paper by noting the modest programming achievement reported by other investigators and asking whether students' lack of low-level programming knowledge or high-level strategies was to blame. Even in asking the question, we recognized its presumption: Both could be implicated and lack of low-level knowledge might put too simply the nature of students' knowledge difficulties. The case studies and general findings reported here suggest a perspective on novices' programming difficulties somewhat more subtle than knowledge versus strategies. That perspective and its implications lend themselves to discussion by way of four questions. What characterizes novices' difficulties with programming? Are the difficulties just a consequence of poor instruction? What are the implications for the teaching of programming? What are the implications for programming's impact on cognitive skills?

What characterizes novices' difficulties with programming?

The data reviewed here suggests this answer: fragile knowledge exacerbating a shortfall in elementary problem-solving strategies. As to fragile knowledge, both the case studies and the tallies of the effectiveness of prompts and hints in marshalling students' knowledge demonstrated that viewing novices' knowledge as partial was too simple. Besides problems of partial knowledge, students displayed inert knowledge that they could not readily muster, misplaced knowledge that migrated to inappropriate contexts, and conglomerated knowledge that mixed together commands in syntactically or semantically anomalous ways. The causes of such fragile knowledge seemed varied but comprehensible. Among the factors discussed were a sparse network of associations, underdifferentiation of commands binding of commands and programming plans to customary contexts without recognizing their generality, treating a programming language more like a natural language where one can say what one means in many ways, and, of course, underuse of general strategic questions to prompt oneself to better marshall one's knowledge.

The range of difficulties posed by fragile knowledge might seem dismaying, but there is another way to look at it. The phenomena of fragile knowledge say that students know more than you might think. To be sure, that knowledge is often inert, underdifferentiated, undergeneralized, and so on, but at least it is there in some nascent form. Moreover, the fragile knowledge phenomena of misplaced and conglomerated knowledge catch students in the midst of seeking to cope with the task in an exploratory way. If the misplacements and conglomerates will not do the job hoped for, at least they signal the students' efforts to muster what they know and apply it somehow. Note that while misplacements and conglomerates by definition will not work, they are never nonsensical. While PRINT "*" X will not print X *'s in a row, and a student with a good understanding of PRINT would know that, nonetheless one can see how such a format <u>might</u> perform such an action.

Now consider problem-solving strategies. The strategic shortfall implicated by the clinical work involves rather elementary strategies. Prompts in the spirit of "what now," "what other ways," "how can you describe the problem," "what's the plan," "do you know a command to do that," and "what will this command do," dominated. These concern several aspects of problem solving -- formulating goals, generating solutions, evaluating solutions, breaking set. However, they contrast with many efforts to enumerate problem solving heuristics that emphasize somewhat more sophisticated strategies, in effect taking such simple prompts as these almost for granted (Polya, 1954; Polya, 1957; Schoenfeld, 1980; Wickelgren, 1974).

Why elementary problem-solving strategies come to the fore here seems plain enough. The students' fragile knowledge will not sustain any very sophisticated problem solving. On the contrary, in light of their fragile knowledge, students' principal problem becomes how to muster that knowledge most effectively. Elementary prompts rather than sophisticated strategies that take much more for granted fit the bill. For this reason, we say that the strategic shortfall exacerbates the fragile knowledge problem. It is not enough just to conclude that students need both more robust knowledge and more strategies as though the two were independent dimensions of programming. One must appreciate how knowledge and strategies work together to support one another, weaknesses in either finding partial compensation in the other.

Are the difficulties just a consequence of poor instruction?

We have already noted that in our judgment the students we studied were taught in a careful and conscientious way by a teacher with excellent mastery of BASIC. Yet anyone familiar with programming must be taken aback by some of the errors students committed in our case studies. "How could any well-instructed student think that would work?" someone might ask. On this interpretation, one need only teach programming in the solid way we at least sometimes teach many another subjects and the difficulties will recede. In our view, this reading of the circumstances underestimates the difficulty of programming and learning to program. Although programming achievement of students certainly varies according to the ability of the students and the expertise of the teachers (Linn, 1985), it seems to us that too many students display substantial difficulties too often to justify attributing such troubles solely to teaching.

Consider for a moment what a challenge programming is. Unlike most school subjects, programming is problem solving intensive. One cannot even come close to getting by just by knowing answers. Moreover, as the phenomena of misplaced and conglomerated knowledge demonstrate, a freewheeling manipulation of one's knowledge will not suffice either, as it might to some extent in the arts or literature for instance. The demands of a computer that cannot discern what a program means are inevitably more stringent than the demands of a reader who not only can see through to meanings but may appreciate exploratory and playful extrapolations and stretchings of concepts. Moreover, programming calls for extraordinary perfection. If you get 90% of the words right on a spelling test, you score a 90; if you make 90% good points on an essay, you probably get an A. But if you get only 90% of your commands right in a program, it will not do anything like what it is supposed to. Moreover, the remaining 10% may well introduce several interacting errors that make tracking down the bugs a demanding and frustrating task.

For these reasons and no doubt others, we suggest that normally responsible and knowledgable instruction does not suffice to give students a reasonable mastery of programming, particularly the students who do not show a flair for programming. Just as programming makes extraordinary demands on students, so does the teaching of programming make extraordinary demands on pedagogy.

What are the implications for the teaching of programming?

With this challenge in mind, how should programming be taught? It would be glib to suggest that the present findings can offer anything like a full formula for so complex an enterprise. Nonetheless, three broad recommendations follow from our observations. First of all, teach programming so as to reduce the problem of fragile knowledge. This general directive translates into a number of particular objectives. One needs to highlight the functional roles of commands in their generality to work against inert knowledge. For instance, a FOR loop needs to be seen as a way of repeating anything any number of times that can be calculated as the loop is entered; PRINT needs to be seen as outputting a carriage return character that hence either terminates a line or creates a blank line. Also, one needs to convey an understanding of exactly what commands do. For instance, a FOR loop does not just repeat something in a holistic sense but goes through a particular iteration process with a particular end-test, making possible nonstandard applications such as transferring out of a loop before it is complete. One needs to caution students about freewheeling and treating the computer language as though it were a natural language that could get across an intended meaning in many ways.

Second and somewhat paradoxically, teach programming so as to preserve the exploratory use of the language. It would be a shame to convey such a stringent image of programming that students became fearful of making conjectures, as indeed some students seem to be (cf. the discussion of "stoppers" versus "movers" in Perkins, Hancock, Hobbs, Martin, & Simmons, in press). The solution to the paradox involves what might be called "controlled exploration," exploratory thinking filtered by a precise appreciation of what the programming language affords. For instance, we think this to be a good principle: "When in doubt, take a stab at a solution; but check your stab by close tracking." This encourages students to attempt cycles of invention and critical filtering from which they can learn. For example, a couple of times a student in our clinical work proposed PRINT "*" X or a slight variant, close tracked what would happen upon execution, and realized it would not behave as desired. We see such episodes not as unfortunate sidetracks but as important learning experiences wherein students enlarge their own understanding of the language by generating possibilities and testing those possibilities against the knowledge they already have. However, such exploratory learning cannot go well unless the students have enough knowledge to be able to close track well.

Third, _encourage the use of elementary problem-solving strategies._ As our data demonstrate, students would gain by prompting themselves more often with simple strategic questions such as "what does the program need to do next," "what command do I know that might help to do that," "what will what I have written really do," or "how did my program get that wrong answer?" The last two, prompts to close track, have special importance. As stressed before, close tracking is the critical filter that allows detecting programming errors with understanding. To be sure, running the program to see what happens also acts as a critical filter: If the program fails, there is something wrong. However, the critical filter of running the program provides far less information than the critical filter of accurate close tracking. While the former simply presents the programmer with the fact of an error, and perhaps an error message or some anomalous output, the latter leads the student through the program's action blow by blow.

What are the implications for programming's impact on cognitive skills?

The present findings in no way document an impact of programming on general cognitive skills. On the contrary, if anything they suggest that students of programming need stronger general problem solving skills in the first place in order to best build upon their fragile knowledge. Rather than expecting programming instruction of itself to boost cognitive strategies, one should teach cognitive strategies as part of better programming instruction.

However, the findings do support the idea that beginning programming is a natural arena for the development of general cognitive skills. To see this point, consider what the present study might have found instead. High level prompts could have proved quite ineffective. Only hints might have succeeded in marshalling students' fragile knowledge. Such a finding would suggest that even elementary problem-solving strategies had no very important role to play until students achieved a substantial mastery of the basics of the computer language in question. However, this was not the finding. Instead, it appears that certain general problem-solving strategies can contribute from early on.

In sum, we suggest a distinction between programming as an arena for the development of cognitive skills and programming as an activity whose pursuit automatically develops cognitive skills. Our data argue for the former and against the latter. Direct teaching or indirect encouragement of strategic self-prompting and other tactics should help students to learn to program better and increase the likelihood of transfer from programming as well (Cf. Salomon & Perkins, this issue).

References

Adamson, R. E. (1952). Functional fixedness as related to problem solving. Journal of Experimental Psychology, 44, 288-291.

Anderson, J. R., & Reiser, B. J. (1985). The LISP tutor. Byte, 10(4), 159-175.

Belmont, J. M., Butterfield, E. C., & Ferretti, R. P. (1982). To secure transfer of training instruct self-management skills. In D. K. Detterman & R. J. Sternberg (Eds.), How and how much can intelligence be increased? (pp. 147-154). Norwood, New Jersey: Ablex.

Bereiter, C., & Scardamalia, M. (1985). Cognitive coping strategies and the problem of inert knowledge. In S. S. Chipman, J. W. Segal, & R. Glazer (Eds.), Thinking and learning skills, Vol. 2: Current research and open questions (pp. 65-80). Hillsdale, New Jersey: Erlbaum.

Clements, D. H. (1985, April). Effects of Logo programming on cognition, metacognitive skills, and achievement. Presentation at the American Educational Research Association conference, Chicago, Illinois.

Clements, D. H., & Gullo, D. F. (1984). Effects of computer programming on young children's cognition. Journal of Educational Psychology, 76(6), 1051-1058.

Davis, R. B. (1984). Learning mathematics: The cognitive science approach to mathematics education. Norwood, New Jersey: Ablex.

de Villiers, J. G., & de Villiers, P. A. (1978). Language acquisition. Cambridge, Massachusetts: Harvard University Press.

Delclos, V. R., Littlefield, J., & Bransford, J. D. (1985). Teaching thinking through LOGO: The importance of method. Roeper Review, 7(3), 153-156.

Feurzeig, W., Horwitz, P., & Nickerson, R. (1981). Microcomputers in education (Report no. 4798). Cambridge, Massachusetts: Bolt, Beranek, & Newman.

Linn, M. C. (1985). The cognitive consequences of programming instruction in classrooms. Educational Researcher, 14, 14-29.

Luchins, A. S. (1942). Mechanization in problem solving. Psychological Monographs, 54(6).

Nachmias, R., Mioduser, D., & Chen, D. (1985). Acquisition of basic computer programming concepts by children (Technical report no. 14). Tel Aviv, Israel: Center for Curriculum Research and Development, School of Education, University of Tel Aviv.

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

Pea, R. D., & Kurland, D. M. (1984a). On the cognitive effects of learning computer programming. New Ideas in Psychology, 2(2), 137-168.

Pea, R. D., & Kurland, D. M. (1984b). Logo programming and the development of planning skills (Report no. 16). New York: Bank Street College.

Perkins, D. N. (1985). The fingertip effect: How information-processing technology changes thinking. Educational Researcher, 14(7), 11-17.

Perkins, D. N., Hancock, C., Hobbs, R., Martin, F., and Simmons, R. (in press). Conditions of learning in novice programmers. Journal of Educational Computing Research.

Perkins, D., & Salomon, G. (in press). Transfer and teaching thinking. In Bishop, J., Lochhead, J., & Perkins, D. (Eds.). _Thinking: Progress in research and teaching_. Hillsdale, New Jersey: Erlbaum.

Polya, G. (1954). _Mathematics and plausible reasoning_ (2 vols.). Princeton, New Jersey: Princeton University Press.

Polya, G. (1957). _How to solve it: A new aspect of mathematical method_ (2nd ed.). Garden City, New York: Doubleday.

Salomon, G., & Perkins, D. N. (1984, August). _Rocky roads to transfer: Rethinking mechanisms of a neglected phenomenon_. Paper presented at the Conference on Thinking, Harvard Graduate School of Education, Cambridge, Massachusetts.

Schoenfeld, A. H. (1980). Teaching problem-solving skills. _American Mathematical Monthly, 87,_ 794-805.

Schoenfeld, A. H. (1982). Measures of problem-solving performance and of problem-solving instruction. _Journal for Research in Mathematics Education, 13_(1), 31-49.

Schoenfeld, A. H. & Herrmann, D. J. (1982). Problem perception and knowledge structure in expert and novice mathematical problem solvers. _Journal of Experimental Psychology: Learning, Memory, and Cognition_, _8_, 484-494.

Soloway, E., & Ehrlich, K. (1984). Empirical studies of programming knowledge. _IEEE Transactions on Software Engineering, SE-10_(5), 595-609.

Wickelgren, W. A. (1974). _How to solve problems: Elements of a theory of problems and problem solving_. San Francisco: W. H. Freeman and Co.