DOCUMENT RESUME

ED 295 620 IR 013 327

AUTHOR Perkins, David; And Others
TITLE Loci of Difficulty in Learning to Program. Technical Report 86-6.
INSTITUTION Educational Technology Center, Cambridge, MA.
SPONS AGENCY National Inst. of Education (DHEW), Washington, D.C.
PUB DATE Jun 86
CONTRACT 400-83-0041
NOTE 22p.; For a related report, see IR 013 324.
PUB TYPE Reports - Research/Technical (143)

EDRS PRICE MF01/PC01 Plus Postage.
DESCRIPTORS Difficulty Level; Error Patterns; High Schools; *Knowledge Level; *Problem Solving; *Programing; Sex Differences; Student Attitudes; Teaching Methods

ABSTRACT

To learn more about the specific nature of the teaching and learning problems involved, researchers conducted a clinical study of 20 high school students enrolled a 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. A coding system was used to record the type of difficulty students encountered, the amount of help needed, and the correctness of solutions. Experimenters noted whether errors were omissions of a necessary element, inappropriate migrations of an element from one command to another, errors in sequencing the elements, or other mistakes. Data analysis provided information about loci of difficulty in three aspects of programming behavior: attitudes, knowledge base, and problem-solving strategies. Inadequacies in students' knowledge base about BASIC were revealed, with most errors occurring at the level of application. Findings suggest that programming instruction might place greater emphasis on encouraging students to prompt themselves with strategic questions about problematic situations, on helping them achieve more consolidated knowledge of the details of computer languages, and on addressing attitudinal and confidence factors in programming. (32 references) (Author/MES)

86-6

LOCI OF DIFFICULTY IN LEARNING TO PROGRAM

Technical Report

June 1986

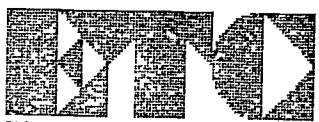
Educational Technology Center
Harvard Graduate School of Education
337 Gutman Library Appian Way Cambridge MA02138

86-6

LOCI OF DIFFICULTY IN LEARNING TO PROGRAM

Technical Report

June 1986



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

Loci of Difficulty in Learning to Program

Technical Report
June 1986

prepared by D. N. Perkins, Fay Martin, and Michael Farady

Programming Group Members

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

Preparation of this report was supported in part by the Office of Educational Research and Improvement (Contract # OERI 400-83-0041). Opinions expressed herein are not necessarily shared by OERI and do not represent Office policy.

Loci of Difficulty in Learning to Program

The evidence continues to accumulate that students in primary and secondary school studying programming do not learn to program very well. Studies of Logo programming conducted at Bank Street showed that students emerged from instruction of a nondirective sort with very limited mastery of the basics of Logo (Pea & Kurland, 1984a, 1984b). Linn (1985), reporting on an extensive survey of BASIC instruction, noted that student learning was quite limited at most sites, although better results appeared at a few sites with higher ability students and especially mindful instruction. In a recently completed study, Bank Street researchers examined the competence of high school students involved in a two year computer science sequence involving considerable programming instruction as well as related aspects of computer use (Kurland, Pea, Clement, & Mawby, in press). At the end of the instruction, many students still displayed surprising shortfalls in their competence.

To be sure, education aspires to greater achievement in virtually every area of the curriculum. Still, the gap between achievement and aspiration seems unusually large in the case of programming. Students emerge from many straightforwardly and seemingly competently conducted programs of instruction with much more limited mastery than one would have expected. The circumstances pose a puzzle both for psychology and pedagogy: How to explain the difficulties of programming that stand in the way of student progress?

One possible answer turns to the demands programming makes on problem-solving ability. Programming is a problem-solving intensive subject. While students may manage well enough in many school subjects by learning the facts and rote procedures and delivering them back in homework and quizzes, programming instruction routinely asks students to create their own programs to do a variety of tasks. Accordingly, programming routinely presses students for a variety of analytical and compositional competencies that other subjects, as normally taught, do not demand. Paradoxically, most programming instruction does little to teach such competencies directly; rarely does programming instruction explicitly address the sorts of heuristics or managerial strategies that might abet the problem-solving process. There is some evidence to suggest that when the instruction includes explicit attention to such matters, students do better (Clements, 1985; Clements & Gullo, 1984). Outside the area of programming, there is evidence to suggest the explicit teaching of heuristics and managerial strategies can improve problem solving substantially (Schoenfeld, 1982; Schoenfeld & Herrmann, 1982).

One might also seek an understanding of the difficulties of programming in a second direction: As well as a problem-solving intensive subject, programming is a precision-intensive subject. An able programmer needs to know the key commands, understand the purposes that they serve, grasp what actions they effect in the world of the computer, and respect the syntactic requirements of their proper use. To be sure, such knowledge is merely the low-level "database" that supplies information for the higher order problem solving students of programming need to do. However, it is by no means clear that students have that "database" well in hand. One conspicuous shortfall is the difficulties students commonly display in hand-executing given programs or programs they have already written (Pea & Kurland, 1984a, 1984b;

Perkins, Hancock, Hobbs, Martin, & and Simmons, in press). Hand-execution requires no problem solving; it is a purely algorithmic task calling only for a precise grasp of what actions the various commands effect. Yet students routinely have difficulty with tasks of this sort, demonstrating that the database of commands and their effects in the world of the computer needs extension and repair.

Perhaps it is somewhat surprising to think that the low-level knowledge of commands may be a major source of students' difficulties. But one does well to remember that the circumstances of programming call for an exceedingly high accuracy rate. If one recalls with accuracy 90% of the dates on a history quiz and fails to retrieve or retrieves erroneously the rest, one has turned in a creditable performance. If one handles with accuracy only 90% of the commands in a particular program, almost certainly the program will fail. A computer program is a highly interdependent structure, where errors in one locale have effects that propagate to spoil the whole performance of the program. Consequently, although the total amount of rote information a student needs to master for programming may be substantially less than that a student needs to master for history or a foreign language, the fullness of mastery the student requires to do well is much higher.

In summary, prior research suggests that programming may prove a difficult subject for students both because of its problem-solving intensive character and its precision-intensive character. But such a proposal demands explication. What sorts of problem-solving strategies might be needed? If students have trouble mastering commands, what forms does such trouble take? Can it be said that either problem-solving skills or shortfalls in knowledge base lie at the heart of the matter, difficulties with the one primarily reflecting difficulties with the other?

With such questions as these in mind, we undertook a clinical study of high school students in the second semester of a BASIC course. We sat with students as they attempted certain programming problems and helped them in systematic ways when they encountered difficulties. The students' interactions with the computer and our interactions with the students were consolidated into protocols and subjected to several sorts of analyses. The results illuminate the loci of difficulty in learning to program.

Method

Subjects

Twenty high school students participated in the study. Ranging from 10th to 12th graders, they consisted of 11 girls and 9 boys. The students were enrolled in the second semester of an elective first-year BASIC course. They were all taught by the same teacher, who had a professional level of mastery of programming. Each student participated for one 45-minute session. The instruction in the BASIC course seemed clear, solid, and straightforward, with considerable practice time and individual attention. Presentation of new commands and concepts proceeded in a very structured way, with systematic explanation and workbook exercises. The primary student activity was working through sets of programming problems; the problems tended toward business matters such as record keeping or processing a data file. Toward the end of the second semester, the students were studying two-dimensional arrays.

Programming tasks

In the experimental sessions, the investigator showed each subject a series of eight programming problems. The problems were sequenced from relatively easy to fairly difficult, each new problem building on the one before. The sequence of eight was based on the FOR-NEXT loop, asking the students to write programs that would produce various patterns of stars (asterisks) on the screen. Although the students had been exposed in class to all the concepts and primitives needed to write the programs, they may not have encountered them in the context of producing a graphic design.

To convey the character of the problem sequence, we describe several of the problems. Problem 1 asked for a program that would produce a column of 10 stars; problem 3 called for a program that would ask the user for an input (a number) and then print a column of that many stars. Problem 4 called for a program that would ask for an input and then print that many stars horizontally. Problem 5 requires a program that asked for an input and then printed a square of stars. With an input of 5, the program would produce:

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

Problem 8 called for a program that would print a hollow square of stars determined by the user's input. With an input of 5, the program would produce:

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

Procedure

Each session was held during the student's regular BASIC class period. The experimenter and student worked together in a separate room using a Digital Rainbow personal computer. The experimenter explained the purpose of the study -- to understand what aspects of programming are easy or hard to learn. The experimenter explained that there were programming problems for the student to work on; while the student worked, the experimenter would watch, take notes, and help if the student had trouble. The experimenter also reminded the student that the session would be audiotaped.

The experimenter then presented the sequence of problems and asked the student to choose one to begin that was "not too easy, but not too hard a challenge." As the student worked, the experimenter watched, recording on paper the student's code and other notes such as the student's affect during the session. Occasionally the experimenter asked questions to track the student's thinking. When the student encountered a difficulty he or she was

unable to resolve alone, the experimenter intervened by asking questions or providing information. The experimenter worked with the student until the program ran successfully. Then the student was asked to work on the next problem in the sequence. This procedure was repeated until time ran out.

When the student encountered a difficulty, the experimenter first asked questions designed to provoke strategic thinking. We called this type of question a "prompt." By definition, prompts were questions that did not require a knowledge of the true nature of the difficulty, questions that in principle students could have thought to ask themselves. Some typical prompts were: "What's the first thing you need to tell the computer to do?"; "What does this semicolon do?"; "How would you describe the problem to yourself?" As the examples suggest, some prompts were general and could be used in any problem-solving situation, whereas others were specific to programming. 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 offered more specific assistance. This type of intervention was called a "hint." By definition, hints were questions or comments that reflected the experimenter's understanding of the solution. A hint could be a leading question or a bit of information. Hints sounded like this: "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?"; "Is there some way you could use 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 proceed with the rest of the programming. These were called "provides." Characteristic provides were: "Write INPUT 'How many stars do you want', N"; "use a FOR-NEXT loop"; "Put a semicolon after the print statement." The experimenter attempted to explain answers when providing them. Note that not all "provides" directly gave the student code. For instance, if the student's immediate problem was to retrieve the name of a relevant command, and prompts and hints failed, the experimenter would provide the name of the command, but not details on how to use it. The experimenter would then wait to see whether the student could follow through with the command, before intervening further with prompts, hints, and provides.

The escalations from prompt to hint to provide 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 alone. A successful prompt suggested that the student could benefit from learning to self-prompt in the same way. At the other extreme, a provide preceded by several unsuccessful prompts and hints indicated very limited knowledge and understanding of 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 the student's increasing frustration indicated that more direct help was needed to maintain attention and involvement.

Data

Data collected during each session included notes taken by the experimenter, code written by the student that was transcribed by the experimenter, and an audiotape of the conversation. The audiotape was transcribed later and the notes and code interpolated to yield a verbatim protocol of the session.

Coding system

The experimenters developed a coding system to apply to the protocols. The coding system classified the students' success or difficulties with the steps we identified as necessary to complete the program in question -- the "critical components" discussed below. The students' success or difficulties were inferred by the scorers from the students' verbally expressed ideas and actions at the keyboard that furthered the development of the program or that the student hoped would further its development. Comments like "I need some kind of a loop" or "I can use INPUT" would lead to one or more scores. Side remarks like, "This is a snap" or "This is tough" would not.

The coding system sought to capture three aspects of the students' programming behavior: (1) the level of organization at which students evinced a difficulty, for instance overall organization of a program versus details of the handling of a particular command; (2) how much help a student needed with a particular step -- none, a prompt, a hint, or a provide; (3) the correctness of a student's handling of a step -- was it correct, was the student unable to provide any response, or what sort of error did the student's response display?

Presenting a sample interaction with a student will facilitate describing the coding system. For such a sample, suppose that a student is addressing problem 4, which called for a program to print a horizontal row of stars of length determined by the user's input. The student recognizes the need to get information into the program, mentions using READ, and then appears stuck. The experimenter says, "Can you think of another command you might use?" The student says, "Oh yeah, INPUT."

Critical components of a program. To facilitate coding students' responses for each programming problem, we identified those components of a solution that were, we felt, critical to resolving the problem. For example, we established four critical components for problem 4: an INPUT statement, a FOR-NEXT loop with a variable containing the input number for its upper limit, a PRINT followed by a star and semicolon, and spaces between the stars, effected by a statement such as "PRINT '* ';" rather than "PRINT '*';". For each student who completed problem 4 or nearly completed it (interrupted by the end of the period), the student's performance on each one of these critical program components was scored. Accordingly, the sample episode above would be scored as part of the INPUT component. Similar critical components were defined for all the problems and students were scored accordingly on any problem they completed or almost completed.

Level. The performance of each student on a given program component was coded according to the student's handling of four major programming levels: decomposition, need formulation, command finding, and application. Decomposition referred to realizing the need for the program component -- in the case of Problem 4 and INPUT, realizing that some way of getting information in was required. Need formulation referred to formulating in

sharper terms what the component should do -- in the case of Problem 4 saying something like, "I need a command that reads an input from the user." Command finding referred to retrieving an appropriate command -- INPUT in the case of the example. Application referred to following through with correct code using the command -- a correct INPUT line in this case. Considering again the sample episode, the student's effort to retrieve INPUT is an example of command finding.

The level scheme provided one approach to gauging the extent to which students suffered from weaknesses in their BASIC "database" versus higher order difficulties as in decomposing the programming problem. The four levels were considered a logically necessary sequence for any program component. All four levels were given distinct scores for each critical component, even though a student might not display any overt sign of passing through a particular level. Scoring each level was possible because (a) a student always completed the component, even if with help from the experimenter, and (b) in those cases where a student gave no overt evidence of a level, it was scored by inference in light of the student's performance on a subsequent level. For example, if a student immediately wrote a correct command line -- an act at the application level -- by inference the scorers classified the decomposition, need formulation, and command finding levels as "spontaneously correct" (see below for the meanings of these terms). For a contrasting example, a student requiring the experimenter's help with each level in turn would be scored accordingly at each level.

Help. Students' responses were classified according to the help required to elicit them -- spontaneous, or with the help of a prompt, hint, or provide. This classification did not assume that a subject's response to a prompt, hint, or provide was correct; by definition, responses to provides were always correct since the experimenter directly provided the needed information, but responses to prompts and hints were often incorrect. Note that the classification of help from the experimenter as a prompt, hint, or provide was done from the protocols, without reference to the experimenter's original intentions, which were in any case not indicated in the protocols. Referring to the sample case again, the students' retrieval of INPUT would be classified as in response to a prompt, because the experimenter's question, "Can you think of another command you might use?" is a question not betraying any particular knowledge of the answer, a question a student under similar circumstances might well ask himself or herself.

Correctness. Finally, the correctness of a response was classified into one of several categories: correct; omissions -- failure to include a needed element in a response, as for instance in omitting a semicolon on PRINT or leaving out the FOR-NEXT altogether; migrations -- cases where a syntactic element from one command intruded into another, as for example in PRINT S STEP 2; sequence errors -- where an element was mispositioned in a program, as in placing one after another FOR-NEXT loops that should be nested; mistakes -- a catch-all category for erroneous responses not otherwise classified. In the example, the subject's response of INPUT would be classified as correct; the subject's prior response of READ would be classified as a migration (of READ into the territory of INPUT). The scoring would also note that the student spontaneously rejected this migration.

The coding process

The principal scorer went through each protocol, looking for responses and classifying them into a set of 4 x 4 charts, one for each critical component of each problem, with the help dimension across the top and the level dimension down the side. An entry in a chart included a letter code to indicate correctness and an annotation to indicate what response in the protocol the entry referred to. A second scorer scored a randomly selected subsample of the protocols independently of the first scorer. After the adequacy of interjudge agreement was established (see below), the principal scorer's classifications were adopted for all further analyses.

Teacher's grades and ratings

The experimenters collected from the teacher at the end of the term the students' final course grades. They also collected the teacher's ratings for each student on a three point scale (low, medium, high) for three attitudinal factors: motivation, enjoyment, and persistence. The teacher was encouraged to take these terms in their everyday senses and was given no more specific guidance about their interpretation. Both the grades and the attitude ratings inevitably raise issues about interjudge agreement, since there was no second grader or rater for a crosscheck. However, we felt that the teacher's professional role and year-long familiarity with the students put her in a good position to assign objective grades and ratings. Also, as the analysis will reveal, substantial relationships emerged between the teacher's grades and ratings and the data gathered by the experimenters, forming a coherent picture that argues indirectly for the soundness of the grades and ratings.

Results

Coding reliability

Occasionally, the first scorer coded a response missed by the second scorer or vice versa, but this did not happen frequently. Disagreement as to which critical component a response belonged to was quite rare. Disagreement as to level of response was somewhat more common. Degree of agreement was calculated by treating the sequence decomposition, need description, command finding, application as a four-point scale and calculating a Pearson correlation coefficient; the result was .92 ($N=185$, $p<.001$). Degree of accord regarding help was calculated in the same way; the correlation was .82 ($N=183$, $p<.001$). Agreement as to the correctness of a response was very high, but this reflected the fact that very often students' responses were straightforwardly correct and both scorers would so classify them. To provide a more stringent test, the scorers' agreement was examined on those responses that the scorers both classified into one or another of the noncorrect categories; there, the second scorer agreed with the first 76% of the time. Although this figure is not high, it is much greater than the chance rate of 25%, assuming equal frequency of the categories. In general, then, a very satisfactory level of interscorer agreement was obtained.

Relations between performance on interview and in class

It is natural to wonder whether the students' programming performance during the interviews adequately reflected their general programming performance. Might not the unusual conditions of interviewing -- a strange interviewer, the frequent questions, even the help offered by the interviewer -- distort students' normal rank order of programming performance? To examine this matter, indices of the students' programming performance during the interview were compared with their final course grades. For programming performance during the interview, we used three indices: the number of responses that were spontaneously correct, the number of resolutions that had to be provided, and the number of the final problems a student attempted. The figures were drawn from applications level responses, because, as will be seen later, subjects made the most mistakes at this level and hence their number of correct responses would be a more individuating measure at that level than for responses pooled across levels.

The first row of Table 1 presents the correlation coefficients calculated between these variables. As the table shows, the correlation coefficients cluster around .57, indicating a distinct relation between the students' interview performance and their overall course performance. To be sure, this magnitude of correlation is far from perfect, but, given such intervening factors as the interview format itself, students' response to instruction subsequent to the interview, and accidents of performance both on the interview and in the course, these correlations argue that at least the interview format did not "reshuffle" students' relative levels of performance. It may be objected that the interview format perhaps led many students to perform considerably below their norms, while preserving relative levels of performance. We have no formal evidence on this point, but can remark informally that while some students appeared nervous, others did not, and that, since the interview format itself provided direct support for students' problem-solving processes, a net disruptive effect appears unlikely.

Insert Table 1 about here

Performance in relation to attitudinal factors

As noted earlier, the teacher provided attitudinal ratings for each student on the dimensions motivation, enjoyment, and persistence. The remainder of Table 1 displays the correlations between these ratings and the measures of performance during the interview. The correlations fall in the neighborhood of .5. The results suggest that student's effectiveness during the interview related to the student's general attitudes to programming. As one would expect, more competent programmers are more motivated, enjoy programming more, and display more persistence.

[illegible] adding that the attitudinal variables correlated even more [illegible] course grade, in the neighborhood of .7. This means that they re[illegible] more strongly to overall course performance. This appears to give[illegible] evidence of a connection between attitudes and performance. However, [illegible] point of caution, one should remember that all ratings were provided [illegible] the teacher without any second rater. Although the pattern of

Table 1

Correlation Coefficients between Class Performance and Interview Performance

Class Performance	Interview Performance		
Teacher's Assessments	Number of Spontaneously Correct Responses	Number of Solutions Provided by Experimenter	Number (1-8) of Final Problem Attempted
Course Grade	.57****	-.58****	.57****
Motivation	.51**	-.65****	.39*
Enjoyment	.38*	-.56****	.25
Persistence	.48**	-.52***	.39*

* p<.05, ** p<.025, *** p<.01, **** p<.005, one-tailed test, N=20.

results is plausible, conceivably the attitude ratings reflect a halo effect from competence or vice versa or both.

The concentration of difficulties at various levels

As mentioned earlier, one aim of the present experiment was to determine the degree of difficulty presented by the four levels of programming identified -- decomposition, need formulation, command finding, and application. This would help determine the extent to which students suffered from inadequacies of their BASIC "database" versus higher order difficulties. One relevant calculation examined at each level the percentage of critical components resolved spontaneously, rather than by way of a prompt, hint, or provide. The percentages for decomposition, need formulation, command finding, and application respectively were 92, 78, 83, and 64.

We asked whether these figures differed genuinely or merely reflected variations in sampling the same underlying distribution. A 4 x 2 chi square test would have been inappropriate because of implicit constraints on the marginals resulting from details of the scoring system. Instead, we obtained a best bet underlying distribution by pooling the tallies over decomposition, need formulation, command finding, and application; we performed chi square tests comparing this theoretical distribution with each of the four actual distributions, rejecting the null hypothesis that all these percentages reflected the same underlying distribution (largest chi square = 39, df=1, $p<.001$). To underscore the significance of the percentages, 92% of the time subjects resolved a matter spontaneously at the decomposition level; in contrast, they resolved a matter at the application level spontaneously only 64% of the time. The numbers demonstrate that subjects encountered the most difficulties at the application level.

Another approach to comparing levels was to consider what percentage of subjects' difficulties occurred at the decomposition, need formulation, command finding, and application levels. These figures, respectively, were 10, 25, 18, and 47%. A chi square test reflecting the null hypothesis of an equal distribution (25, 25, 25, 25) rejected the null hypothesis (chi square = 73, df=3, $p<.001$). In summary, nearly half of subjects' difficulties overall occurred at the application level.

It is natural to ask whether the more able subjects encountered relatively fewer difficulties at the application level and more at the decomposition, need formulation, and command finding levels. This might occur because the more able programmers would have mastered the details of commands better and so encounter fewer difficulties at the applications level; also, they would tackle the more difficult problems in the sequence, which presented more challenges of decomposition and need formulation. This question was examined by computing the percentage of difficulties at the application level for each subject and correlating this figure with course grade and with the problem number (1-8) of the final problem attempted, two reasonable indices of overall competence as indicated earlier. The correlations, .06 and -.31 respectively, did not approach significance, showing that the difficulties of the more able subjects occurred no less often at the applications level than those of the less able subjects.

Another natural question concerns whether those subjects who dealt successfully with at least the early phases of approaching the programming problems -- decomposition and need formulation -- approached the problems in

the same way. After all, at least in principle, the programming problems could be decomposed in more than one way, leading to divergent approaches. This question was pursued not by a formal analysis but simply by inspecting over half the protocols for any sign of divergent decompositions of the problems. No such cases emerged. It seems appropriate to conclude that divergent approaches were either not apparent or not attractive to the students.

Effectiveness of prompts and hints

The analysis examined the extent to which prompts and hints were effective in helping students over difficulties they encountered. Pooling the data over all four levels, we examined the percentage of time for the entire sample that a prompt, a hint, or a provide resolved a difficulty. The figures were 33%, 15%, and 52% respectively. In summary, about half the time a prompt or a hint assisted a student in resolving a difficulty, while the rest of the time a provide proved necessary. The effectiveness of prompts and hints indicated that subjects possessed "inert knowledge" that they had not initially been able to use, since both prompts and hints required subjects to add information of their own in order to resolve the difficulty.

We also examined whether the more able programmers were more responsive to prompts and hints than the less able programmers. For each student, the percentage of difficulties resolved by prompts was calculated. Also, for each student the percentage of difficulties not resolved by prompts but resolved by hints was calculated. These figures were correlated with final course grade for a measure of programming ability. The correlation coefficients were .28 and .25, neither of which differed significantly from 0. It appears that the more able programmers were not significantly more responsive to prompts. Of course, one must remember that the more able programmers were reacting to prompts given in the context of more difficult problems.

The types of difficulties that occurred

The data were pooled to examine the relative frequency of the four error types. There were 359 errors classified in all, distributed as follows: omissions, 37%; migrations, 27%; sequence errors, 8%; mistakes, 28%. We examined whether this distribution of error types varied according to whether the error occurred spontaneously or in response to a prompt or hint. No significant difference among these three conditions emerged.

We also examined whether the distribution of error types varied with level: decomposition, need formulation, command finding, and application. There were substantial differences, but they were not psychologically interesting, reflecting instead the structure of the task and coding system. Only one migration occurred at the decomposition and need formulation levels, because before reaching the command finding level subjects rarely had mentioned any commands or command structures, so there was nothing that could be said to have migrated. Sequence errors only occurred at the applications level, because the placement of commands in the program was considered to be an applications-level matter.

The distribution of error types indicates that one cannot interpret the students' problems with their BASIC knowledge base just as a matter of

outright knowledge gaps about some commands. Omissions accounted only for 37% of the data. Migrations, sequence errors, and mistakes all indicated garbled rather than entirely missing knowledge about the program elements in question.

Confidence and competence in relation to gender

In light of concerns about males' and females' attitudes and achievement in such traditionally male areas as mathematics and computers (cf. Badger, 1981; Hawkins, in press), we examined the data for contrasts between the males and females on final grade, motivation, enjoyment, persistence, responsiveness to prompts, responsiveness to hints, final problem attempted, and initial problem. For all but the last variable, no significant differences emerged. There were indications that females found a prompt helpful a slightly lower percentage of times than did males, but this was at a marginal level of significance, especially considering the number of tests performed.

A marked contrast appeared between the initial problem chosen by females with that chosen by males. The average number of the initial problem selected by the females was 1.7 and by the males 4.2, a statistically significant difference (*t* test, $p<.003$). There was, however, a nonsignificant difference between females' and males' final problem, the females averaging slightly lower. To test whether females indeed chose easier initial problems without displaying a difference in ability to cover problems, we conducted an analysis of variance on initial problem with final problem as a covariate. The results confirmed the conclusions from the *t* test ($F=7.3$, $df=1$, $p<.015$). It appears that the females were less confident, the males more so, in their selection of an initial problem. This points to an attitudinal difference that does not appear when males and females are compared on the other indices of ability or attitude.

Discussion

The analysis of the data provided information about loci of difficulty in three aspects of student's programming behavior: attitudes, knowledge base, and problem-solving strategies.

Attitudes

The analysis disclosed the relation one would expect between measures of programming competence and attitude: More competent programmers, as measured by indices derived both from the interview and course grade, tended to display more motivation, enjoyment, and persistence as rated by their teacher. In addition, an interesting gender contrast appeared. While not differing in competence or other respects, female students evinced less confidence than male students by selecting starting problems that were less challenging. This finding is consonant with the idea that, in areas such as mathematics and computers, attitudinal factors may distinguish males from females in our culture more so than do matters of ability (cf. Badger, 1981; Hawkins, in press).

However, the relations between attitude and competence allow no simple inference as to cause. It is certainly possible that students' attitudes

reflect their underlying competence in dealing with the subject matter of programming, which may in turn reflect general intellectual competence; Linn (1985) reports that success in programming relates considerably to IQ. On the other hand, more motivation, enjoyment, and persistence could signal the presence of a more active conception of learning that in turn fosters more competence (Dweck & Bempechat, 1980; Dweck & Licht, 1980; Zelman, 1985). Further support for this idea comes from our observation that some students tend to be "stoppers," disengaging from a problem as soon as difficulties appear, while others tend to be "movers," continuing to attack a problem that presents difficulties, although not necessarily effectively (Perkins, Hancock, Hobbs, Martin, & and Simmons, in press). In general, it seems unlikely that the causal arrow would run only in one direction between attitudes and competence. Most plausibly, attitudes and competence would reinforce one another in a kind of loop. The results suggest what should be axiomatic in education in any case: Instruction needs to take into account not only how students perform but how they feel about their learning activities.

Knowledge base

It would be easy to consider programming as a challenge principally to students' problem-solving abilities, since, by the measure of any natural language, programming languages present relatively few primitive terms to be mastered. As noted at the outset, programming is precision-intensive, but this might not pose that much of a problem in light of the limited "database" required. However, the analysis offered ample evidence that students are troubled considerably by inadequacies in their knowledge base. In particular, students displayed more difficulties at the application level, where details in the knowledge of BASIC figured more, than at any other level. Omission errors and migration errors were commonplace, pointing to problems of missing and garbled knowledge of BASIC.

The contemporary research on expertise leads one to expect knowledge-base problems in performance. Good problem solving in a domain appears to depend upon a repertoire of schematic problem types and solution types which are specific to the domain (e.g. Chase & Simon, 1973; Chi, Feltovich, & Glaser, 1981; Larkin, McDermott, Simon, & Simon, 1980; Schoenfeld & Herrmann, 1982). The importance of such schemata has been demonstrated for programming specifically (Schneiderman, 1976; Soloway & Ehrlich, 1984). However, it must be emphasized that the knowledge problems detected in this experiment did not seem to be of the order of the schemata detected in such studies. Rather, many subjects fall afoul of quite elementary aspects of writing individual command lines.

The analysis also demonstrated that characterizing students' shortfall in knowledge as a matter of "missing knowledge" would be simplistic. The students often had relevant knowledge but failed to bring it to bear, as demonstrated by the frequent success of prompts and hints in marshalling students' "inert knowledge." Inert knowledge is known to be a problem in other contexts of composition besides programming (Bereiter & Scardamalia, 1985). Inert knowledge may be considered a case of failure of transfer, attributable not only to the knowledge representation but to whether the learner mindfully seeks opportunities for transfer (Belmont, Butterfield, & Ferretti, 1982; Salomon & Perkins, 1984; Perkins & Salomon, in press). Sequence errors and migration errors also point to problems of knowledge that are not straightforwardly interpretable as missing knowledge.

Elsewhere, we have discussed a four-way classification of students' knowledge shortfalls in programming as missing knowledge, inert knowledge, misplaced knowledge, and conglomerated knowledge. Missing knowledge refers to knowledge that a student lacks access to, even upon prompting. Inert knowledge is knowledge that surfaces upon prompting or hinting or in other ways but was not initially retrieved. Misplaced knowledge means knowledge that intrudes into inappropriate contexts, as with the present migration category. Conglomerated knowledge refers to students' occasional use of anomolous combinations of command elements pulled from two or more contexts. We introduced a new term, "fragile knowledge," to summarize the four and underscore the point that the weaknesses in students' knowledge base requires a sharper characterization than "missing knowledge" alone affords (Perkins & Martin, 1986).

Problem-solving strategies

The effectiveness of prompts in helping students over difficulties argues that students fail to use important elementary strategies for problem solving. By definition, prompts were advice a student might well give himself or herself. Also, the prompts amount to elementary problem-solving strategies -- asking oneself what the current problem is, what commands might help, whether what one has just written does what one supposes, and so on. The elementary character of these strategies deserves emphasis. We are not addressing the somewhat more sophisticated strategies often discussed in contexts of mathematical problem solving, such as constructing a visual representation of a problem, reducing a problem to prior problems, or checking a solution through special cases (Polya, 1954, 1957; Schoenfeld, 1980; Wickelgren, 1974). To be sure, such strategies are certainly relevant to the context of programming; indeed, Soloway and Ehrlich (1984) have commented on the role of some of them. However, the present findings suggest that students do not help themselves enough with quite elementary attention-focusing strategies in dealing with programming problems.

It must be acknowledged that the efficacy of prompts in the interview contexts does not in itself demonstrate that students could learn to prompt themselves and thereby improve their autonomous performance. However, at least the finding suggests that this might be possible. A contrary finding -- that prompts had no impact on subjects' performance -- would hold out little hope of students learning to guide their problem-solving processes with more art.

Conclusion

The results of this experiment argue that novice programmers' difficulties are far from monolithic. One might suppose at first that programming principally challenged students' problem-solving abilities. In fact, the results give evidence of loci of difficulty in attitudes and mastery of the relevant knowledge base as well as in elementary problem-solving strategies.

The results also argue that students are troubled by some difficulties of a rather simple character. As noted earlier, other research has shown the relevance of a schematic repertoire of problem and solution types and of certain problem-solving heuristics to activities like programming, problem solving in mathematics and physics, and chess play. However, our findings

suggest that, at least in programming, shortfalls in knowledge and strategies much simpler than those usually discussed may impair novices' efforts considerably. We may need to reassess our intuitions about what is difficult; matters that one would not initially think of as difficult turn out to be so for the novice. It is even conceivable that an approach to programming instruction which sought to inculcate more sophisticated schemata and strategies, without addressing these elementary matters, would falter for lack of attention to them. Perhaps curriculum design efforts aiming to help students with the learning and problem-solving processes involved in programming should begin at quite an elementary level.

In particular, students might be encouraged to self-prompt themselves with the sorts of questions that focus attention on helpful aspects of problematic situations. Instruction might be designed so as to help students to achieve more consolidated, less fragile knowledge of the details of the computer language in question. This may require giving students a better mental model of the computer (DuBoulay, O'Shea, & Monk, 1981; Mayer, 1976, 1981) as well as systematic frameworks for learning (Perkins, 1986). Finally, the findings recommend instruction in programming that recognizes the attitudinal problems that students may have and tries to boost students' attitudes directly by whatever means seem likely. In further research at the Educational Technology Center, we are investigating these avenues.

References

Badger, E. (1981). Why aren't girls better at maths? Educational Research, 24, 11-23.

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.

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

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

Clements, D. H. (1985a, 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.

DuBoulay, B., O'Shea, T., & Monk, J. (1981). The black box inside the glass box: Presenting computing concepts to novices. International Journal of Man-Machine Studies, 14, 237-249.

Dweck, C. S., & Bempechat, J. (1980). Children's theories of intelligence: Consequences for learning. In S. G. Paris, G. M. Olson, & H. W. Stevenson (Eds.), Learning and motivation in the classroom (pp. 239-256). Hillsdale, New Jersey: Lawrence Erlbaum Associates.

Dweck, C. S., & Licht, B. G. (1980). Learned helplessness and intellectual achievement. In J. Garbar & M. Seligman (Eds.), Human helplessness. New York: Academic Press.

Hawkins, J. (in press). Computers and girls: Rethinking the issues. Journal of Sex Roles.

Kurland, D. M., Pea, R. D., Clement, C., & Mawby, R. (in press). A study of the development of programming ability and thinking skills in high school students. Journal of Educational Computing Research.

Larkin, J. H., McDermott, J., Simon, D. P., & Simon, H. A. (1980). Modes of competence in solving physics problems. Cognitive Science, 4, 317-345.

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

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

Mayer, R. E. (1981). The psychology of how novices learn computer programming. Computing Surveys, 13(11), 121-141.

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. (1986). Knowledge as design. Hillsdale, New Jersey: Lawrence Erlbaum Associates.

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. N., & Martin, F. (1986). Fragile knowledge and neglected strategies in novice programmers. In E. Soloway & S. Iyengar (Eds.), Empirical studies of programmers. Norwood, New Jersey: Ablex.

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

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.

Schneiderman, B. (1975). Exploratory experiments in programmer behavior. International Journal of Computer and Information Sciences, 5, 123-143.

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.

Zelman, S. (1985, April). Individual differences and the computer learning environment: Motivational constraints to learning LOGO. Presented at the American Educational Research Association Annual Meeting, Chicago, Illinois.