The Buggy Path to The Development of Programming Expertise

Roy D. Pea, Elliot Soloway, Jim C. Spohrer

Focus on Learning Problems in Mathematics
Winter Edition 1987, Volume 9: Number 1

The Buggy Path to The Development of Programming Expertise

Roy D. Pea
Bank Street College of Education

Elliot Soloway
Jim C. Spohrer
Yale University

In the U.S. over a million precollege students take computer programming courses each year, and more than 50,000 teachers are involved with programming instruction. Unlike mathematics education, in which decades of research have led to a deepening understanding of the development of early number acquisition, algebraic cognition, and geometric problem-solving, cognitive research on the development of programming skills is an infant field. Nonetheless, a broad range of investigations within the last five years, as well as informal reports from programming educators in practitioner journals and at national meetings of NECC and AERA, have begun to illustrate the host of conceptual difficulties programming novices encounter in learning to program.

For programming, as in other domains from mathematics to the physical and engineering sciences, students are engaged through their learning activities in **actively building** a knowledge system of concepts and procedural skills. This domain-specific constructivist orientation is a pervasive component of modern cognitive science theories of learning (e.g., Glaser, 1984) and has been applied successfully to school learning of various topics in mathematics and science education (e.g., Carey, 1985; Carpenter, Moser, & Romberg, 1982; Resnick, in press). One phenomenon that such an orientation is designed to explain is the nature of student "errors" or "misconceptions," i.e., faulty answers to problems posed in the domain of study. Although some "errors" are slips in the mechanics of problem solving, it now appears that most faulty answers arise from systematic applications of the knowledge a student currently **does** have to the problem at hand. If one looks closely enough at the distribution of a

student's answers to problem types, one can see that the student has what various investigators have called a "theory," "belief system," or "schema," in terms of which they understand the phenomenon at hand, and which leads to the answers he or she gives. Such a view leads to a **positive** characterization of a student's current understanding in terms of the knowledge he or she is utilizing to **make sense** of the problem solving activities in computer programming. While such answers are "errors" from the canonical perspective, they are sensible generalizations from what the student currently knows.

By analogy to computer programs that do not run to specification, students' noncanonical answers have been called "buggy." These responses are due to "bugs" in the sense that if the students' knowledge structures used to generate answers are revised, canonical performances can be expected to result.

Our aim in this essay is to provide an overall scheme of interpretation that can "raise the consciousness" of programming instructors and others interested in programming education, so that the kinds of conceptual bugs students develop and manifest in programming can be "seen" more readily. We will address the following questions: What are the types of programming bugs in early programs and in programs of intermediate complexity, and what are their likely sources? What might instructors do to identify these bugs and to help students develop better programming understanding so that these buys are no longer manifest? What kinds of knowledge does programming instruction need to convey to help students overcome these buggy tendencies?

As in the case of mathematics learning, many instructors are likely to be aware of such bugs only tacitly, and may view them as "mistakes" on the student's part rather than in terms of systematic applications of the student's current understanding of programming concepts and procedures. By making the tacit explicit, we hope to provide a better definition than is now available of the pedagogical problem facing programming educators. If these "bugs" persist, we expect programming students are more likely to lose interest in further developing their programming skills because of their recurrent failures in making their programs run.

Charting the buggy terrain: An overview

Many research projects are underway in artificial intelligence, cognitive science, and software psychology to identify types and examples of programming bugs, to trace their sources and mechanisms of construction by the learner, and to work on defining their optimal paths to remediation. But little synthesis is available to offer signposts pointing to bug locations for the observer of students' programming practices. What we begin to provide in this paper is a sketchmap of

the terrain: what types of bugs might the programming instructor expect to encounter and what might be their sources? Although such a map is likely to need elaboration, refinement, and much more experimental testing, we can already begin to see some patterns of bug clustering, and get some sense of why novices' early programs are rife with bugs.

Bugs in program creation and program comprehension will be surveyed in three major categories. First, we will discuss **programming language-independent conceptual bugs,** that is, those which appear to arise in consequence of learning to implement goals in a formal programming system per se, rather than being due to specific features of the programming language being learned. Secondly, we consider the category consisting of **knowledge unavailability bugs,** arising from an inadequate knowledge base, and ranging from semantics, syntax, and plans, to whether students have a realistic view of flow of both control and data as programs are interpreted by the computer. Finally, the third category, which comprises **knowledge retrieval bugs** will be outlined. In this category, students often have available the relevant knowledge for solving a programming problem but do not access that knowledge.

Programming language-independent conceptual bugs

The prospective programmer quickly meets what might be called the "conversational metaphor" for writing programs (Hutchins, Hollan, & Norman, 1986). One creates a set of instructions according to the rules of an input language, and the computer interprets these instructions and executes them according to the programming language's conventions for command interpretation. This conversational metaphor is a powerful one indeed, since the programmer is a natural language user who has communicated throughout an entire lifetime in a conversational manner.

But in contrast to programming discourse, various techniques have evolved among natural language communities for managing and repairing misunderstandings that arise when the utterances a speaker offers as communicative acts are not understood by the listener. The listener can say "Huh?," ask the speaker questions, or use any one of a number of other natural language devices (e.g., elaborations, contradictions) to try to establish the intentions of the speaker. And often a listener can correctly impute meaning to the speaker's utterance in a conversation without the speaker ever elaborating on what he or she has said — the listener fills the gap, and the communicative act is successful, conveyed **without explicitness.** At the extremes of mutual understanding achieved in intimate communication, truncated speech is the rule, as Vygotsky (1962) reminds us in his reference to Kitty and Levin's laconic conversations in Tolstoy's **Anna Karenina.**

A central component of such human activities of discourse repair and communicative interpretation is that the listener has a deep understanding of what the speaker **could** mean in a given context. Typically, the two participants in the discourse share a common field of objects and events, the conversation has a history in which certain knowledge is assumed as shared, and the speaker-listener pair also bring a powerful background of interpretive frames that help guide their activities of speech interpretation and repair in a conversation.

We thus see a major discrepancy between conversation in natural language, and the writing/reading of programs in terms of a conversational paradigm. Whereas the speaker-listener pair can negotiate what was meant interactively, in terms of a rich common knowledge base of interpretive conventions, the "listener" side of the computer programming task is seriously impoverished. This point of contrast between the natural language and programming discourse is a major hurdle the novice programmer must overcome, and as we shall see, the solitary path to explicating all of one's meaning to make a program run as intended is bug-ridden. In sum, whereas the "debugging" of natural language discourse is socially accomplished, and often inexplicit, the novice programmer must go it alone in total explicitness, since the program interpreter requires complete specification in programming code of the programmer's intent.

What we find in novice programming are a host of bugs that appear as consequences of the programmer's attempts to **generalize** from natural language to programming discourse. These bugs are evident in the early programs of novice programmers from elementary school to adulthood, and they can even appear on occasion when the programmer has developed expertise.

The pragmatic strategies for creating and understanding natural language serve them poorly when they begin to program, because computers interpret their programs of instructions by means of mechanistic rules. For the programming languages typically learned by novices (e.g., BASIC, Pascal, Logo), there are deterministic rules for interpreting commands in a specific sequential order defined by how flow of control is dealt with in the language. While people are intelligent interpreters of conversations, programming languages are not. Humans fill the gaps of inexplicitness, helping repair the ambiguities of speaker utterances by supplying background knowledge on what the speaker would be likely to mean in that interpretive context.

This nondiscursive feature of programming thus violates human conversational maxims, such as the principles of cooperation outlined by Grice (1975) and developed in theories of sociolinguistics and natural language pragmatics (Cole, 1981; Searle, 1983). The computer cannot infer what a speaker means if she is totally explicit,[1] whereas the listener cooperates to make possible an interpretation of the speaker's talk.

Pea (1986) has outlined three major classes of programming students' conceptual bugs that appear to derive from the "superbug" of negative transfer of such natural language interpretive conventions to

the formal domain of computer programming. The classes of language-independent conceptual bugs that spring from this generalization are Parallelism Bugs, Intentionality Bugs, and Egocentrism Bugs. Unless noted, observations reported below took place in research with students learning Logo (ages: 8-12 year-olds, 14-17 year-olds) or BASIC (high school students).

Parallelism Bugs

The parallelism bug appears in various contexts. Its central feature is that the student assumes that different program lines can be active or known by the computer at the same time, in other words, in parallel. One often finds the parallelism bug in programs where conditional statements (IF... THEN) occur outside loops. For example, picture a case in which the conditional statement appears early in a program:

IF SIZE = 10, THEN PRINT "HELLO"

Then, a countup loop comes later in the program: the variable is incremented by one each time until it has a value of ten:

FOR SIZE = 1 TO 10, PRINT "SIZE"
NEXT SIZE

Now we may ask what students think the computer will do as it interprets this program. If they understand the control structure of the programming language (BASIC), they know that the IF statement is first evaluated for its truth. If SIZE is equal to ten, HELLO is printed, and control passes to the next statement. If the variable is not equal to ten, nothing is printed, and control passes to the next statement. After the test of the IF line of the program is executed, that line of code is inactive, irrelevant to whatever the remaining lines of the program instruct since control never returns there.

But high schoolers in their second year of computer science faced with such a problem predicted a surprising result. In one study, eight out of the fifteen students interviewed predicted that during the looping process, when the variable SIZE becomes equal to ten, HELLO **would** be printed. When explaining why, it was noted that since variable SIZE was **now** equal to ten (i.e., within the loop) and the IF statement was "waiting for" the SIZE to be equal to ten, it could now print HELLO. But in fact, once the IF statement was evaluated and found false, the computer never read it again. It appears these students believe that all program lines are simultaneously active, and that the program has the intelligence to monitor the action status of every program line at once. Similar findings have been reported for novice Pascal programmers, where as many as a third of the college students mistakenly assumed for simple Pascal programs that the actions in the **while** loop were continuously monitored for the exit condition to become true (Bonar & Soloway, 1983; Soloway, Bonar, Barth, Rubin, & Woolf, 1981).

We can see in these cases how students are biased by natural language conversational strategies, where expectations of what will come later can guide the interpretation of what occurs early in a conversation or text. Apart from quasiprocedural natural language such as building plans, spatial directions, and recipes, there is often no reason not to skip ahead for interpretation. In natural language, one hardly violates a text's meaning by reading parts of it out of order; we even teach the reading strategy of scanning ahead for text structure. But the strict flow of control for command interpretation defined for the specific programming language must be adhered to. To forecast program outcomes, the student must ask only what conditions regarding inputs are in effect as each line is executed.

Intentionality Bugs

Another class of bugs is revealed when the student attributes goal directedness or foresightedness to the program. By analogy to the natural language listener, the novice programmer assumes the computer can go beyond the information given in the code to a program interpretation. The novice treats the complex system represented by the programming language from an "intentional stance" (Dennett, 1978), granting it human interpretive capacities.

Kurland and Pea (1985) asked preadolescents judged to be "good programmers" by their teachers to think out loud as they draw on graph paper what the graphics pen will draw as the following tail-recursive Logo program is executed. When one types SHAPE 40, the program draws a large square, a medium-sized square inside it, and then stops. More specifically:

```
TO SHAPE: SIDE
IF: SIDE = 10 STOP
REPEAT 4 [FORWARD: SIDE RIGHT 90]
SHAPE: SIDE/2
END
```

The program draws a square with a variable side that, when initialized on the first call, is 40 units long. The second line of the program is a conditional counter with the purpose of stopping the drawing after two squares are drawn. The third line draws a square, the side of which is the length of the variable SIDE (i.e., 40). The fourth line divides the variable SIDE by two. Since the program begins with a conditional statement that says when the variable SIDE equals 10 stop, the program draws the two squares (of size 40 and 20) and terminates, because the variable SIDE then equals 10.

When predicting the effects of the program's second line, some students mistakenly suggest that the program will draw a box of size 10. Their explanations reveal intentionality bugs. They have looked

ahead in the program to a familiar programming plan — the third line that usually results in the drawing of a square: REPEAT 4 [FORWARD (SOME DISTANCE) RIGHTANGLE TURN (90 DEGREES)]. They then read the IF statement in the second line as if the program is **commanding** the computer to draw a square with sides equal to ten, because "it will draw a square," or "because it wants to draw a square." Other students recognize that the variable value at the IF statement is 40, but then say that the program "sees" the box statement line ahead which it wants to draw, but has to stop at 10. In each case, the student imbues the program with the status of an intentional being which has goals, and knows or sees what will happen elsewhere in itself.

Sleeman, Putnam, Baxter, and Kuspa (1986) describe similar "deep" errors for high-school novice Pascal programmers after a semester-long introductory course. Students often erroneously inferred the function of a program from only a few instructions, or even from the name of the program, assuming they needed to read no further to find the intention of the program. Kuspa and Sleeman (1985) report comparably inappropriate, semantically-driven interpretations of procedure functions among Logo learners. Another common misconception Sleeman et al. found was that a READ statement used with a meaningful variable name causes the program to select a value based on the name's meaning from a list of values in the DATA statement.

Egocentrism Bugs

Whereas intentionality bugs are revealed in comprehending and tracing what a program will lead the computer to do, egocentrism bugs emerge in creating a program to do something. Both bug types presuppose that the computer can do what it has **not** been told to do in the program. "Egocentrism" is an overemphasis on one's own perspective relative to that of others. It is a widespread trait of children's thinking, in early spatial cognition (Piaget & Inhelder, 1967), communication (Flavell et al., 1968), and other problem domains. Under the strenuous cognitive demands of a new task environment, it may also surface as a characteristic of the performances of adolescent or adult novice programmers. We are thus not surprised to find egocentric biases in novice programming. Egocentrism bugs reveal students' beliefs that more of their intention expressing what they want to accomplish is resident in the programming code than is actually present. For example, they omit lines of code, variable names or values, and other key instructions, assuming the computer "knows" or can "fill in," as a human listener can, what the student wishes it to do.

Students revealing egocentrism bugs do not say outright that the program knows what to do. Such bugs are virtually perceptual — the students' current conceptions do not guide their attention to these omissions as causally responsible for the bugs in their programs. A common problem of this kind is the omission of punctuation or control characters, and the nonprovision of values for variables (e.g., Kuspa & Sleeman, 1985). Lest these omissions be thought of only as careless work, one can then probe the students to test this hypothesis, which attributes more significance to these lacunae than clerical oversight. When one asks students to predict the outputs of programs they have written with these omissions, they gloss over the specific commands in a line of Logo code just written, e.g., asserting that a line of graphics code draws a square when they have included a move comand to send the turtle forward but **no** turn command for making the necessary right angles:

REPEAT 4 [FORWARD 30]

It is as if they do not **see** that the necessary specifications are missing. They have provided only a program skeleton, trusting that the computer can fill the gaps.

Carver and Klahr (in press) found two similar tendencies in their examinations of 8-year-old Logo programmers' debugging. Children would make predictions for the graphics turtle that matched their goal rather than what the program code said, and would skip commands that called other procedures, assuming the turtle would ignore any commands which they did not understand.

Bonar and Soloway (1983) have documented egocentrism bugs for college students writing Pascal programs, again involving the student's beliefs that the programming language can know more about intentions than it possibly can, given available code. Soloway, Ehrlich, Bonar, and Greenspan (1982) found novice Pascal programmers incorrectly using the same variable for more than one role, e.g., to store a value being read in [read (X)] and to hold a running total [X: = X + X]. It was as if the students assumed that the computer would recognize that the same variable played two different roles and could know when to change the role of the variable to make the program work as intended. Such multiple-valued variables are also common conceptual bugs by high school novice programmers in BASIC (Putnam, Sleeman, Baxter, & Kuspa, 1985) and Pascal (Sleeman, Putnam, Baxter, & Kuspa, 1986).

The "Hidden Mind" Superbug

These three classes of language-independent conceptual bugs appear to stem from what might be called a **superbug:** the idea that there is a hidden mind in the machine that has intelligent, interpretive powers. This benevolent being knows what has happened or will happen in program lines other than the one being executed; it can go

beyond the information given to help the student achieve his or her goals in writing the program.

But do students literally believe that the computer has a mind, can think, and interpret the unstated? Novice programmers will vehemently deny that the computer has these mentalistic traits. Instructors are very good at explaining that computers are dumb and can do nothing but what they are told. However, students' behaviors when working with programs betray their denials, for they act as if the programming language provides more than a mechanistic route to intention-expression. The student's **default strategy** for making sense when encountering difficulties of program interpretation or when writing programs is to resort to the powerful analogy of natural language conversation, to treat the computer as a disambiguating mind that can understand. The central point is that this analogy is predictable rather than bizarre behavior, for the students have no other analog, no other procedural device than the "person" to which they can give instructions that are then followed. Rumelhart and Norman (1981) have similarly emphasized the critical role of analogies in early learning of a domain — making links between the to-be-learned domain and known domains perceived by the student to be relevant. But in the case of programming, mapping conventions for natural language instructions onto programming results in error-ridden performances.

Metacognitive aspects of computer programming

Another language-independent set of conceptual problems that can arise in learning to program surrounds metacognitive skills. Brown (1978), Flavell (1976), and Garafalo and Lester (1985) characterize two distinctive components of metacognition. The first concerns "executive skills," that is, those skills involved in regulating and controlling one's mental activities during problem solving. The second component consists of one's beliefs and knowledge about cognition. Flavell (1979) observed that students prior to middle school reveal little knowledge of cognitive processing, and rarely monitor their memory and comprehension. Schoenfeld (1985) found that college students solving mathematical problems engage in little self-monitoring of their problem solving processes.

There are various kinds of bugs in learning to program that either result from or are perpetuated by the minimal use of metacognitive processes during programming. Difficulties have been documented for program writing, reading, and debugging. For example, Kurland and Pea (1985) had middle school age Logo programmers with 50 hours of programing behind them hand-simulate a progressively more complex series of brief recursive Logo graphics programs, predicting what will be drawn when the program is run. Students often did not adequately monitor their comprehension of the program, in

that they would skip lines and not check their work. Similar findings emerged for high school students learning Logo programming (Kurland, Clement, Mawby, & Pea, in press; Kuspa & Sleeman, 1985). Using a similar method, Carver and Klahr (in press) found that 8 year-old Logo programmers made many "placekeeping" errors in both a program comprehension task and a debugging task. These included redoing commands, skipping commands (especially those calling other procedures), doing the wrong number of iterations in a REPEAT statement, forgetting the final turn in a REPEAT statement, and losing track of the current variable value in a recursive procedure. They note that few students used written placekeeping marks to note their progress.

One possible pedagogical response to these difficulties is to use explicit prompts in early stages of programming instruction aimed to provide explicit metacognitive "scaffolding" for novice programmers. For example, in Logo graphics programming, McBride (1985; also see Hillel, 1985) recommends using strategy prompts such as "Is a setup needed?" to block the novice's tendencies to pay exclusive attention to the object to be drawn while ignoring the necessary reorientation of the turtle to the appropriate startup state. Such external modeling of what would be self-regulatory activity for an expert programmer is consistent with Vygotskian approaches to instruction (Rogoff & Wertsch, 1984) and has been applied to early programming (Delclos, Littlefield, & Bransford, 1985).

Metacognition in programming has been little studied to date, and the above remarks have emphasized self-regulatory components of metacognition. It is likely that metacognitive activities involving reflection on the state of one's knowledge and skills in programming are central to the development of programming expertise. For example, careful program documentation should emerge after a programmer discovers the difficulty in tracing program bugs without it. Explicit awareness of the strengths and limitations of one's program design and program debugging strategies should lead one to seek out help in order to learn to more effective program.

Goal/Plan Merging Bugs: Tackling Complex Programs

In the previous section we described language-independent bug types that were typically encountered when the students were in the early phases of learning to program. In this section, we describe a prevalent type of bug that arises as students are attempting to write more complex programs. However, we need to first develop some new vocabulary to describe this new class of bugs.

Goal/Plan Analysis: An Example

In moving from a problem to a coded program, there is strong empirical evidence that experts (and novices) develop an intermedi-

ate level representation that facilitates the mapping of problem to program. This intermediate representation is referred to as *a goal/plan analysis*. In the rest of this section we offer an example of a goal/plan analysis. Our method is to present a "mental trace" of an Idealized Expert Programmer (IEP) writing a program for a problem.

Consider, then, the following problem:

> Write a Pascal program that will read a set of integers and output the average of those numbers. Stop reading input when the number 99999 is read.

The first task to perform would be to abstract the essential goals of the problem. Using knowledge about what characteristics of a problem statement indicate major problem goals, the following two key goals would be abstracted from the problem specification:

Goal: compute average - - - > Goal: output average

This notation indicates that achieving the goal of computing the average is a necessary precondition of being able to achieve the goal of outputting the average. Notice that the stopping condition on the loop is not yet considered. A loop stopping condition is a detail that can be handled once other decisions about how to realize the main goals have been made.

Once the major goals have been decided upon, these can be further transformed. In particular, world knowledge about the definition of an average would generate the following subgoals:

Subgoal: compute sum of numbers input —
Subgoal: compute count of numbers input —
- - > Subgoal: compute sum divided by count

Again, we have goal enablement: to achieve the subgoal of doing the division one must compute the sum and compute the count. It is important to separate out the types of knowledge being used. The knowledge about averages comes from world knowledge, and is not programming specific. However, to straighten out the data types of the average, count, and sum, programming specific knowledge plus world knowledge must be used. Programming knowledge suggests that a count of discrete items should be of type integer. World knowledge, on the other hand, suggests that dividing two numbers might not necessarily result in a whole number; programming knowledge would suggest that the result variable be of type real. The problem statement itself says that the numbers read in are integers.

The next task is to retrieve from memory a programming plan that will achieve the subgoal of computing the sum of the numbers input. The activity of summing successively read inputs is one that is used in many programs, so it is not surprising that IEP has a "compiled" routine for carrying out such an activity. We have called this plan the

running total loop plan. The name we use facilitates our understanding of the plan, but the human expert probably retrieves the plan via key features such as function, stereotypicality, etc. A representation of the *running total loop plan* is given below:

```
goal: achieve initialize (running total: = 0)
loop until stopping condition = true
   goal: achieve input (new value)
   goal: achieve update (running total: = running total + new
         value)
end loop
```

The plan is not specific to any programming language. Rather, a form of goal-language/pseudocode is used. Transforming from a language-independent representation into Pascal requires more than just a knowledge of the syntax and semantics of Pascal; we will see that it also requires knowledge of the pragmatics of Pascal.

The *running total loop plan* has buried in it two variable plans: a *new value variable plan* and a *running total variable plan*. The former plan specifies a variable whose role it is to hold each successive input value. The latter plan specifies a variable whose role it is to hold the accumulated sum. The definition of a running total variable entails the fact that the new value is to be accumulated into it.

Still pursuing the subgoal of computing the sum of the input, IEP returns to the problem specification to determine what the stopping condition of the loop is — reading the value 99999. This form of loop termination is a standard one: it uses a sentinel value. The default programming knowledge about sentinel values is that they should not participate in the actual computation (i.e., the sentinel should not be included in the sum). Since sentinel termination is a standard way to stop a loop, instead of customizing the more general plan, a specific sentinel controlled running total loop plan may be retrieved.

The translation of this abstract plan into Pascal requires knowledge of the "pragmatics" of Pascal. While any of Pascal's looping constructs (e.g., for, repeat, and while) could be coerced into use, knowledge of pragmatics suggests that one construct is better suited than the others: while is especially suited for an unspecified number of iterations, with the possibility of terminating before performing any iterations at all. This sort of knowledge goes beyond syntax and semantics in that one needs to understand the goal behind each of the constructs; we have labeled this sort of knowledge as pragmatic knowledge: knowledge of *when to use* a construct in contrast to knowledge of *how to use it* (Soloway, Bonar, & Ehrlich, 1983).

The translation of the goal to get input from the user also requires additional programming knowledge. While the problem specification makes no mention of it, a prompt is required before reading in values from a user in an interactive run-time environment. The additional prompt goal goes beyond the goals mentioned explicitly in the prob-

lem specification and is based on both programming and world knowledge. After generating subgoals, retrieving plans, and transforming abstract plans into code the following code would result for the sentinel controlled running total plan:

```
total : = 0;
writeln ('please input a number');
read(new);
while new <> 99999 do
  begin
    total : = total + new;
    writeln('please input a number');
    read(new);
  end;
```

Now, IEP turns to the subgoal of counting the number of numbers read in. The programming plan retrieved for this subgoal is the *counter loop plan*, and it is structurally very similar to the running total loop plan. Transforming the plan to Pascal results in the following code:

```
count : = 0;
writeln('please input a number');
read (new);
while new < > 99999 do
  begin
    count : = count + 1
    writeln('please input a number');
    read(new);
  end
```

A major problem solving feat is about to take place. IEP realizes that since each plan assumes the data will be available, the *sentinel controlled running total loop plan* cannot occur separately before the *sentinel controlled counter loop plan*. The data is coming in as a stream, and thus can only be read (accessed) once. This understanding suggests to IEP that the two looping plans need to be combined. This is in effect an optimization for the computer's benefit, since the more natural human procedure is to make two separate passes over the data. Here we see for the first time the need to integrate two plans. We do not mean "compose" two plans; rather, we see that two plans need to be woven together into one piece of code. In order to weave together these two plans, we claim that IEP must "decompile" both plans and reason causally about the elements of each plan. The following are examples of the type of reasoning that IEP needs to carry out in order to be assured that the two plans can be integrated:

- IEP must realize that the goal of each loop plan is the same: namely, process just one item on each iteration.
- IEP must realize that it is OK to put the counter update anywhere in the loop for the following reason: the flow of data in the loop is such that the counter update will not interfere with any calculation in the loop.

The reader may rightfully point out that if IEP is really an expert then a compiled looping plan with both sum and count would already exist. Thus, the integration scenario just described would not take place. However, for the following reasons we felt it to be critically important to include, in a discussion of the knowledge used in programming, a description of the process of integrating two plans:

- In the initial creation of a looping plan that did combine both the summing and counting actions, the expert would most likely have had to go through the sort of reasoning process outlined above.
- We felt it instructive to see how plans are integrated together, since most programs are not built from totally standard components that can simply be composed together; some new construction must take place.
- We wanted to illustrate the legitimate use of "double duty," i.e., a situation in which one piece of code is used for two purposes. The illegitimate use of a double duty can lead to code that is hard to understand (Soloway & Ehrlich, 1984).
- As we shall see in more detail in the next section, *many bugs in novice programs appear at just those points in the design of the program that requires the integration of two plans.* Some bugs arise because the integration of plans is done incorrectly (Spohrer, Soloway, & Pope, 1985), and other times because integration was necessary and the novices failed to do it (Johnson et al., 1983).

Another issue with respect to integrating the two loop plans concerns knowledge of programming discourse rules — rules of style. While on theoretical grounds the counter update and the running total update could go anywhere inside the loop, in practice they are grouped together. To do otherwise would violate a rule of programming discourse that says group code together that has a common role. Both the counter plan and the running total plan have subgoals that correspond to the role of updating a variable inside the loop, so they should be grouped together.

After achieving the subgoals of computing the sum and count of the numbers input, the subgoal of dividing the sum by the count is enabled. To prevent a run-time error of division by zero, programming knowledge gives rise to another implicit subgoal: guard the average calculation. The division and the output it enables must be nested inside the division by zero guard. Integrating the output plan

with a division by zero guard plan requires that something be printed out when a division by zero might occur. Thus, what appears to be a simple plan to output a value becomes more complicated because another print statement is needed in the leg of the guard plan in which the error might occur:

```
count : = 0;
total : = 0;
writeln('please input a number');
read(new);
while new <> 99999 do
  begin
    total : = total + new;
    count : = count + 1;
    writeln('please input a number');
    read(new);
  end;
if count > 0 then
              begin
                average : = total/count;
                writeln('average is ', average);
              end
            else
              writeln('no valid inputs; no average calculated');
end.
```

As presented above, the program is now complete: it will solve the problem it was intended to solve.

Bugs Arising From Goal/Plan Merging

The vocabulary developed in the previous section provides us with important tools for describing classes of novice programming bugs (Spohrer, Soloway, & Pope, 1985). In particular, we have found that a large percentage of bugs arises because students have difficulty coordinating and composing plans to achieve goals (Spohrer & Soloway 1986a, 1986b). Essentially, the students seem to understand the pieces that make up a program, e.g., the language constructs themselves, but they have difficulty in putting the pieces together.

Merging is a complex plan composition strategy. Put simply, merging occurs when novices decide to achieve two different goals with a single, integrated plan. Often novices think that the merged plan will be shorter and more efficient than the unmerged, separate plans. Possibilities for merging arise when two goals share some similar subgoals. Unfortunately, there are often some differences as well as similarities, and novices may over-look the differences. Thus, one type of bug that can result from merging is subgoal drop-out bugs.

For example, consider the Tax Problem in which multiple records of information had to be processed (giving rise to an outer loop goal), and when each record of information was input it had to be checked for errors (requiring a valid data entry goal). The outer loop goal would be responsible for asking the user "Do you have more tax data to be processed?", and the input record consisted of marital status and income information for an individual.

To see why these two goals present an opportunity for merging we must examine the subgoal structure of each goal. First, consider the three subgoals of a sentinel controlled outer loop goal: (1) an initial input of the loop control variable before the while loop, (2) a test of the loop control variable to see if it is the sentinel value, and (3) inside the loop at the bottom the input must be read in again. These three subgoals are very similar to the three subgoals of a standard valid data entry goal, which ensures that input read into a program is valid. The three subgoals of a valid data entry goal are: (1) an initial input that reads in a value, (2) a test of the value to determine if it is valid or invalid (usually a while-do loop), and (3) a retry input inside the loop in case the input value is invalid. A language independent representation of the plans of these two goals is shown below:

plan for sentinel controlled outer loop goal
goal: achieve get first value of loop control variable
loop until value = sentinel value
  ...
  goal: achieve get next value of loop control variable
end loop

plan for valid data entry goal
goal: achieve get first value of input variable
loop until value = valid value
  goal: achieve get new value of input variable
end loop

Note that both goals deal with similar tasks — loops — and that the three subgoals of each of the goals are also similar in function.

When the two goals are achieved using separate plans, the valid data entry goal is achieved inside the outer loop goal. However, when novices merge the goals they create the buggy plan shown below:

```
01 program Tax (input, output);
02 var Status: char; Income, Tax: real;
03
04 begin
05 +-------------------------------+
06 | readln(Status)                |
07 | while ((Status = 's') or (Status = 'm'))
08 |   do begin                    |
09 |     Goal: valid data entry for income
10 |     Goal: compute tax         |
11 |     Goal: output tax          |
12 |     readln(Status)            |
13 |   end;                        |
14 +-------------------------------+
15 end.
```

BUGGY MERGE OF
GOAL: OUTER LOOP
AND
GOAL: VALID DATA ENTRY

The bug in this merged plan is that the retry subgoal of the valid data entry goal is not achieved — the subgoal has been "dropped-out". In other words, instead of asking the user to retry entering valid data, the program will just skip the loop when an illegal value of marital status is entered. The novice was able to capture most, but not all, of the functionality of the two goals in the merged plan.

There are many explanations for why novices make this bug when they merge: losing the final subgoal of valid data entry could be an undetected loss of information from working memory due to the overload imposed by merging (see Anderson & Jefferies, 1985), maybe the novices were aware of the bug but rationalized away the need for the subgoal or changed the over-all goal since merging seemed more efficient and required less code ("deplanning": Noss, 1984), or maybe the novice thought the retry subgoal had been achieved by the "get next value" of the loop control variable subgoal, since it would "get the next value if it were needed."

Perhaps the student consciously carried out the merge on the two goals. When students try to produce efficient, optimized code, they often will consciously attempt to merge goals. However, in this instance we think it more likely that the student simply did not distinguish enough subgoals — and may have been lured into this way of thinking just because the two goals had similar subgoals. Finally, of the 43 college subjects who attempted to do the Tax Problem, 36 attempted to merge the main, sentinel loop, with the marital status process; of those 36 only 4 were able to do so correctly on their first syntactically correct program. Thus, students have a propensity to attempt merging, and moreover, they have a poor track record for being able to carry it out.

We refer to the above situation as a *bug prone pattern*. That is, we have observed a "merge subgoals, drop out last subgoal" bug in other programs. In particular, whenever a program to be solved has a structure where subgoals of two goals are similar, there is a great risk that students will commit a merging bug, where the last subgoal is

dropped off. Because opportunities for merging depend on similar subgoal structures and not any particular language constructs, bugs that result from merging are language independent. Furthermore, the process of merging goals is not limited to just programming. When one needs groceries, one does not make a separate trip to the store for each item, but instead merges each of the separate trips into a single integrated plan. The common subgoals of going to the store and returning from the store permit merging. However, if the store did not supply shopping bags to carry the grocery items all at once, then the merging might have led to a bug (i.e., too much to carry). We are in the process of looking to see if similar bugs arise in non-programming situations.

Language-dependent conceptual bugs

In this section, we describe two major sources of bugs, and offer examples for each of a number of different categories of bugs that have been observed. The distinction we will make — between bugs due to **knowledge unavailability** and bugs due to **knowledge inaccessibility** — is a common-sensical one. Sometimes students do not have knowledge they should, and the bugs we see in their programs derive from gaps in their understanding. On other occasions, problems with students' programs derive from **knowledge inaccessibility bugs,** where sometimes students have available the relevant knowledge for solving a programming problem but do not access it.

Knowledge unavailability bugs

The varieties of knowledge unavailability are as vast as the varieties of knowledge, but a brief list is nonetheless helpful. Students may lack knowledge of the programming language syntax, such as how to use primitives, how to assign and use variables (Hillel & Samurcay, 1985; Kuspa & Sleeman, 1985), and how to define and call procedures (Sleeman, Putnam, Baxter, & Kuspa, 1986).

Students may lack knowledge of the semantics of the programming language commands. For young students of graphics programming in Logo, these difficulties can involve spatial cognition and such bugs as turning from the absolute angle rather than the relative angle specified by the turtle's position (Carver & Klahr, in press; Fein & Scholnick, 1985; Gregg, 1978; Roberts, 1984). Sometimes semantic errors are due to a confusion of the technical meaning of the programming language command and its natural language meaning (e.g., STOP and END in Logo: Kurland & Pea, 1985, Kuspa & Sleeman, 1985; WHILE in Pascal: Soloway et al., 1981).

Students may not have available an adequate mental model of the information processing that transpires between the input of a program and its output. In all languages studied, students appear to construct noncanonical models of how computers interpret programs. Kurland and Pea (1985) found that child Logo programmers interpreted embedded recursion as a looping construct. Bayman and Mayer (1983) describe bugs in novice BASIC programmers' mental models of flow of control and data. These included fallacious beliefs that "INPUT A" means the letter A is input and stored in memory, reading of conditional GOTOs as unconditional GOTOs, confusing LET assignment statements with equation storage in memory, and confusions of "PRINT C" and "PRINT 'C' ". The assignment confusions lead some BASIC students to argue that counters (i.e., a common programming plan expressed as "LET C = C + 1") are **impossible** (Putnam, Sleeman, Baxter, & Kuspa, 1985)! To overcome such problems, DuBoulay, O'Shea, and Monk (1981), Mayer (1984), Mioduser, Nachmias, and Chen (1985), among others, have recommended explicit training with a concrete model showing the important computer locations (e.g., memory spaces, input stack), visible enactments of program statements that involve transactions of control and data, and student "role-playing" of command interpretation by the computer.

Students may lack an understanding of programming discourse, those tacit rules that distinguish programs that (only) run from those run but are also well-structured so that they communicate well to program readers (Joni & Soloway, in press). Students may also not know the types of recurrent plans that are embodied in programs for that language, such as **numeric** plans like the RUNNING-TOTAL LOOP PLAN in Pascal (Soloway & Ehrlich, 1984), or plans in LISP with **lists** as their main data type (Soloway, 1985).

Furthermore, students may lack the knowledge of program structure required for using program cues to narrow their search space during debugging for the source of a programming bug. This contributes to the common haphazard or linear approach to debugging observed by researchers and programming instructors (Carver & Klahr, in press; DuBoulay, 1986; Kurland et al., in press; Sleeman et al., 1986), and the "deplanning" phenomenon, whereby students modify their goal in preference to debugging their program (Noss, 1984).

Knowledge inaccessibility bugs

Novice programmers may often have knowledge to bring to bear on a programming problem but not use it. Among other reasons, this may be due to the storage of that knowledge in such a form that its applicability to the problem at hand is not obvious. For example, many young Logo programmers will use a REPEAT command for

squares (a common activity) but do not utilize it for other multisided figures. Perkins and Martin (1985) have called this phenomenon "fragile knowledge."

Analogical reasoning is central to good programming, since one often has worked on a similar problem before, and a retrieval of what was done in that case can significantly contribute to success with the current problem. One possible pedagogical strategy is to encourage analogical thinking in programming by example. An instructor could exemplify a newly-introduced command or programming plan across a wide range of discrepant problem contexts, thereby diminishing the likelihood that a student will encode the command or plan as specific to one type of problem. Such undergeneralization is a general feature of procedural learning by novices (Langley, 1985). Students could also be encouraged to share programming code and plans, discussing in a group the broader applications of selected student ideas.

Another case of knowledge inaccessibility is the well-known tendency of child Logo programmers to write **simple** programs using Logo primitives rather than hierarchically organized superprocedures that call other procedures, even after examples of superprocedures and discussions of their merits for saving work have been offered (Hillel, in press; Hillel & Samurcay, 1985; Kurland, Pea, Clement, & Mawby, in press; Leron, 1983; Pea, 1983). The reasons for this preference for nonmodular, "linear programming" (Kurland, Clement, Mawby, & Pea, in press) are not currently understood.

Conclusions

We have canvassed the available literature on the kinds of conceptual bugs students manifest as they learn to program. Examples were offered of language-independent bugs arising from a "hidden mind" superbug that overgeneralizes the human conversational metaphor to programming, of "goal-plan merge" bugs, and of bugs due to insufficient metacognitive activity during programming. We also described language-dependent bugs due to either knowledge unavailability or lack of knowledge retrieval. Beyond pedagogical recommendations made within these sections, what are some further implications of these findings for programming instruction?

Once programming instructors realize the prevalence and systematicity of student bugs in understanding programming, the need for new kinds of programming instruction become apparent. We believe bug persistence is in part linked to the infrequency with which they are explicitly confronted by students and teachers. Teachers can better snare bugs if they adopt research methods. These include having students do hand-simulation (line by line) of the predicted outputs of programs designed to elicit specific misunderstandings, debugging activities with the same aim, and think-aloud explanations of a pro-

gram (defined to include a target programming construct) as it is being written by the student. Having students in a clinical interview situation explain what output will be produced by relatively short (5 to 15 line) programs is the technique used by many programming cognition researchers, and could also be used to advantage by programming instructors — both as a diagnostic tool for learning and as an expository method in teaching.

There are additional pedagogical complexities in addressing the "hidden mind" superbug: The programmer does not have to specify every operation to be carried out, since programming languages **automatically** carry out many things (e.g., physical address management; stack storage allocation). So the novice programmer has to learn a **subtle** lesson: some meanings do not need to be explicitly expressed in his or her code, while others do. Since the boundaries of required explicitness vary across programming languages, the learner must realize the necessity of identifying in exactly what ways the language he or she is learning "invisibly" specifies the meaning of code written.

More research is needed on how to help students see that computers read programs through a strictly mechanistic and interpretive process. There is some evidence that this goal can best be achieved by providing "virtual machines," i.e., clear physical models (DuBoulay, O'Shea, & Monk, 1981) or computer simulated physical models (Mioduser et al., 1985) that show how the processing of control and data is regulated by the specific programming language under study. These explanations can be supported by explicit think-aloud examples of how the facile programmer thinks about and makes decisions with respect to program creation and program understanding, similar to those that have been effective for written language understanding (Palincsar & Brown, 1984).

Finally, the programming language-independent goal-plan merging bugs that have been identified point to the need for new kinds of knowledge programming instructors should explicitly aim to convey. Students need more help learning that **goals and plans** are important intermediates between the problem statement and the completed program, they need to be taught recurrently-useful plans, they need to learn about the pragmatic rules for programming discourse, and they need to monitor and test subgoal satisfaction when integrating programming plans — recognizing this common situation as a "bug prone pattern." By making the tacit explicit, and going beyond today's primary emphasis on teaching syntax and semantics of programming languages to the goal-directed and planful aims of programming action, we can expect more effective understanding and practices of programming.

Acknowledgements

The research discussed in this article by the first author and colleagues was supported by the Spencer Foundation and the National

Institute of Education (Contract No. 400-830016). The studies by the second and third authors and their colleagues were supported by the National Science Foundation under Grant MDR-8470150.

FOOTNOTES

1. Several counterexamples are Teitelbaum's DWIM (Do What I Mean) systems added to the Interlisp programming environment, which corrects spelling errors by using syntactic context, and commercially available syntax-correcting compilers. Such painless error revisions are the subject of feverous debates among programmers.

References

Anderson, J.R., & Jefferies, R. (1985). Novice LISP errors: Undetected losses of information from working memory. *Human Computer Interactions, 1*(2), 107-131.

Byman, P., & Meyer, R.E. (1983). A diagnosis of beginning programmers' misconceptions of BASIC programming statements. *Communications of the ACM, 26*(9), 677-679.

Bonar, J., & Soloway, E. (1983). *Uncovering principles of novice programming.* Paper presented at the 10th Annual Symposium SIGPLAN-SIGACT on Principles of Programming Languages, Austin, TX.

Brown, A.L. (1978). Knowing when, where, and how to remember: A problem of meta-cognition. In R. Glaser (Ed.), *Advances in instructional psychology* (Vol. 1) (pp. 77-165). Hillsdale, NJ: Lawrence Erlbaum.

Carey, S. (1985). *Conceptual change in childhood.* Cambridge, MA: MIT Press.

Carpenter, T.P., Moser, J.M., & Romberg, T.A. (Eds.). (1982). *Addition and subtraction: A cognitive perspective.* Hillsdale, NJ: Lawrence Erlbaum.

Carver, S.M., & Klahr, D. (in press). Assessing children's Logo debugging skills with a formal model. *Journal of Educational Computing Research.*

Cole, P. (1981). *Radical pragmatics.* New York: Academic Press.

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

Dennett, D. (1978). *Brainstorms.* Montgomery, VT: Bradford Books.

DuBoulay, J.B.H. (1986). Some difficulties of learning to program. *Journal of Educational Computing Research, 2*(1), 57-73.

DeBoulay, J.B.H., 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.

Fein, G., & Scholnick, E. (1985, June). *Computing space.* Paper presented at the Fifteenth Annual Symposium of the Jean Piaget Society, Philadelphia, PA.

Flavell, J.H. (1976). Metacognitive aspects of problem solving. In L.B. Resnick (Ed.), *The nature of intelligence* (pp. 231-235). Hillsdale, NJ: Lawrence Erlbaum.

Flavell, J.H. (1979). Metacognition and cognitive monitoring. *American Psychologist, 34*, 906-911.

Flavell, J.H., Botkin, P.T., Fry, C.L., Wright, J.W., & Jarvis, P.E. (1968). *The development of role-taking and communication skills in children*. New York: Wiley.

Garafalo, J., & Lester, F.K., Jr. (1985). Metacognition, cognitive monitoring, and mathematical performance. *Journal for Research in Mathematics Education, 16*, 163-176.

Glaser, R. (1984). Education and thinking: The role of knowledge. *American Psychologist, 39*, 93-104.

Gregg, L.W. (1978). Spatial concepts, spatial names, and the development of exocentric representations. In R.S. Siegler (Ed.), *Children's thinking: What develops?* (pp. 275-290). Hillsdale, NJ: Lawrence Erlbaum.

Grice, H.P. (1975). Logic and conversation. In P. Cole & J.L. Morgan (Eds.), *Syntax and semantics, Vol. 3: Speech acts* (pp. 41-58). New York: Academic Press.

Hillel, J. (1985). On Logo squares, triangles and houses. *For the Learning of Mathematics, 5*(2), 38-45.

Hillel, J. (in press). Mathematical and programming concepts acquired by children, aged 8-9, in a restricted Logo environment. *Recherches en Didactique des Mathematiques.*

Hillel, J., & Samurcay, R. (1985, October). *Analysis of a Logo environment for learning the concept of procedures with variable* (Tech. Rep.). Montreal, Quebec: Concordia University, Mathematics Department.

Hutchins, E.L., Hollan, J.D., & Norman, D.A. (1986). Direct manipulation interfaces. In D.A. Norman & S.W. Draper (Eds.), *User-centered system design: New perspectives in human-computer interactions* (pp. 87-124). Hillsdale, NJ: Lawrence Erlbaum.

Johnson, W.L., Soloway, E., Culter, B., & Draper, S. (1983). *Bug Catalog: I* (Tech. Rep. No. 298). New Haven, CT: Yale University, Department of Computer Science.

Joni, S.A., & Soloway, E. (1986). But my program runs!: Discourse rules for novice programmers. *Journal of Educational Computing Research, 2*(1), 95-125.

Kurland, D.M., & Pea, R.D. (1985). Children's mental models of recursive Logo programs. *Journal of Educational Computing Research, 1*(2), 235-243.

Kurland, D.M., Clement, C.A., Mawby, R., & Pea, R.D. (in press). Mapping the cognitive demands of learning to program. In J. Bishop, D. Perkins, & J. Lochhead (Eds.), *Thinking: Progress on research and teaching*. Hillsdale, NJ: Lawrence Erlbaum.

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.*

Kupsa, L., & Sleeman, D. (1985). *Novice Logo errors* (Tech. Rep.). Palo Alto, CA: Stanford University, Department of Computer Science.

Langley, P. (1985). Learning to search: From weak methods to domain-specific heuristics. *Cognitive Science, 9,* 217-260.

Leron, U. (1983). Some problems in children's Logo learning. In *Proceedings of the 7th International Conference of the PME* (pp. 346-351). Shoresh, Israel.

Mayer, R.E. (1981). The psychology of how novices learn computer programming. *Computing Surveys, 13*(1), 121-141.

McBride, S.R. (1985, June). *A cognitive study of children's computer programming* (Tech. Rep. No. 8502). Newark, DE: University of Delaware, Cognitive Science Program.

Mioduser, D., Nachmias, R., & Chen, D. (1985, February). *Teaching programming literacy to nonprogrammers: The use of a computerized simulation* (Tech. Rep. No. 15). Tel Aviv, Israel: Tel Aviv University, Center for Curriculum Research and Development, School of Education.

Noss, R. (1984). *Children learning Logo programming* (Interim Rep. Nos. 1 & 2). Hatfield, England: Advisory Unit for Computer-Based Education, Chiltern Logo Project.

Palincsar, A.S., & Brown, A.L. (1984). Reciprocal teaching of comprehension-fostering and comprehension-monitoring activities. *Cognition and Instruction, 1,* 117-175.

Pea, R.D. (1983, April). *Logo programming and problem-solving.* Paper presented at the American Educational Research Association, Montreal, Canada. (Available as Technical Report No. 12, Center for Children and Technology. New York, NY: Bank Street College of Education.)

Pea, R.D. (1986). Language-independent conceptual "bugs" in novice programming. *Journal of Educational Computing Research, 2*(1), 25-36.

Pea, R.D., & Kurland, D.M. (1983). *On the cognitive prerequisites of learning computer programming.* (Tech. Rep. No. 18). New York, NY: Bank Street College of Education, Center for Children and Technology. (ERIC Document Reproduction Service No. ED 249 931)

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

Perkins, D.N., & Martin, F. (1985). *Fragile knowledge and neglected strategies in novice programmers.* (Tech. Rep.). Cambridge, MA: Harvard University, Educational Technology Center, Graduate School of Education.

Piaget, J., & Inhelder, B. (1967). *The child's conception of space.* New York: W.W. Norton.

Putnam, R.T., Sleeman, D., Baxter, J.A., & Kuspa, L.K. (1985). *A summary of misconceptions of high school BASIC programmers.* (Tech. Rep.). Palo Alto, CA: Stanford University, Department of Computer Science.

Resnick, L.B. (in press). The development of mathematical intuition. In M. Pelmutter (Ed.), *Minnesota symposium of child psychology* (Vol. 19). Hillsdale, NJ: Lawrence Erlbaum.

Roberts, R.J. (1984). *Young children's spatial frames of reference in simple computer graphics programming.* Unpublished doctoral dissertation, University of Virginia.

Rogoff, B., & Wertsch, J.V. (Eds.). (1984). *New directions for child development, No. 23: Children's learning in the "Zone of proximal development."* San Francisco, CA: Jossey-Bass.

Rumelhart, D.E., & Norman, D.A. (1981). Analogical processes in learning. In J.R. Anderson (Ed.), *Cognitive skills and their acquisition* (pp. 335-359). Hillsdale, NJ: Lawrence Erlbaum.

Schoenfeld, A. (1985). *Mathematical problem solving.* New York: Academic Press.

Searle, J.R. (1983). *Intentionality.* Cambridge, England: Cambridge University Press.

Sleeman, D., Putnam, R.T., Baxter, J.A., & Kuspa, L.K. (1986). Pascal and high-school students: A study of errors. *Journal of Educational Computing Research, 2*(1), 5-23.

Soloway, E. (1985). From problems to programs via plans: The content and structure of knowledge for introductory LISP programming. *Journal of Educational Computing Research, 1*(2), 157-172.

Soloway, E., Bonar, J., Barth, J., Rubin, E., & Woolf, B. (1981). Programming and cognition: Why your students write those crazy programs. In *Proceedings of the National Educational Computing Conference* (pp. 206-219). Toledo, OH: NECC.

Soloway, E., Bonar, J., & Ehrlich, K. (1983). Cognitive strategies and looping constructs: An empirical study. *Communications of the Association for Computing Machinery, 26,* 853-861.

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

Spohrer, J.C., & Soloway, E. (in press). Novice mistakes: Are the folk wisdoms correct? *Communications of the Association for Computing Machinery.*

Spohrer, J.C., & Soloway, E. (1986). *Analyzing the high-frequency bugs in novice programs.* In E. Soloway & S. Iyengar (Eds.), *Empirical studies of programmers* (pp. 230-251). Norwood, NJ: Ablex, Inc.

Spohrer, J.C., Soloway, E., & Pope, E. (1985). A goal/plan analysis of buggy Pascal programs. *Human Computer Interactions, 1*(2), 163-207.

Vygotsky, L.S. (1962). *Thought and language*. Cambridge, MA: MIT Press.

Focus on Learning Problems in Mathematics
Winter Edition 1987, Volume 9: Number 1