LEARNING TO PROGRAM = LEARNING TO CONSTRUCT MECHANISMS AND EXPLANATIONS

Teaching effective problem-solving skills in the context of teaching programming necessitates a revised curriculum for introductory computer programming courses.

ELLIOT SOLOWAY

MOTIVATION AND GOALS

A revolution—fueled by the growing body of cognitive science research into the nature of expertise—is causing radical revision (e.g., see [24]) in science and mathematics curricula today. What has been taught in the past is by and large not what an expert actually knows. For example, geometry students typically understand each step in a proof, as the teacher puts it on the board, line by line. However, when attempting to do a proof for homework, students often have no idea where to begin. Why? Mathematicians do not develop proofs in such an orderly, linear fashion. Rather, developing a proof is a nonlinear, search process. Unfortunately, students are not told explicitly about the nonlinear nature of proof development: They see their teacher develop a proof line by line, and not surprisingly, they think they should be able to do the same. Today, teaching topics such as geometric proofs is being revised to include explicit instruction as to the heuristics that guide proof development.

This article continues with the curriculum redefinition effort and focuses on concepts that should be taught in an introductory programming course. Textbooks used in introductory programming courses typically focus on the syntax and semantics of constructs in a language. New research with novice programmers, however, suggests that language constructs do not pose major stumbling blocks for novices learning to program. Rather, the real problems novices have lie in "putting the pieces together," composing and coordinating components of a program [35, 36]. Expert programmers know a great deal more than just the syntax and semantics of language constructs (e.g., [2, 5, 13, 18, 27, 31]. They have built up large libraries of stereotypical solutions to problems as well as strategies for coordinating and composing them. Students should be taught *explicitly* about these libraries and strategies for using them.

Teaching programming in schools is a particularly hot topic now: On the one hand, it is argued that programming is merely a job skill, and that programming instruction should not be included in a general curriculum; on the other hand, it is argued that programming is a subject where one can learn effective problem-solving skills (e.g., [21]). Currently, there is little empirical support for the latter camp (e.g., [10, 17, 23, 33]). However, the lack of impact of programming on problem solving may be due to the fact that students are not taught the key ideas underlying programming: Why *should* learning where to put a semicolon in Pascal lead to enhanced problem-solving ability?

The focus on instruction of the syntax and semantics of programming language constructs leads to an emphasis on the program as the output of the programming process. However, a program has two audiences:

This work was sponsored by the National Science Foundation, under NSF Grants MDR-8470150, DPE-8470014, and IST-8505019.

- *The computer:* The instructions in a program turn the computer into a *mechanism* that dictates *how* a problem can be solved.
- *The human reader:* The programmer needs to have an *explanation* as to *why* the program solves the given problem.

In this view, *learning to program amounts to learning how to construct mechanisms and how to construct explanations.* The need to construct mechanisms and explanations transcends the domain of programming: In daily life, people constantly are developing mechanisms and explanations to solve problems. For example, giving instructions to a visitor on how to get from the airport to an office is a mechanism, and providing a justification for why that particular route is an effective choice is an explanation. In an introductory programming course, students should be taught what they really need to know about programming, and in doing so, they will learn effective problem-solving strategies that transcend the domain of programming.

UNDERLYING ASSUMPTIONS

Two key assumptions underlying the proposed curriculum are the following:

Tacit Knowledge. Experts are not necessarily conscious of the knowledge and strategies they employ to solve a problem, write a program, etc. ([7]). Often, experts refer to "intuition, gut feel, etc." as the sources of their inspiration. However, the scientists' job is to *make explicit that which was implicit*—to tease out the otherwise tacit knowledge that experts have and use. This enterprise has in fact been carried out in a wide variety of content domains such as mathematics (e.g., [20]), physics (e.g., [15]), etc. The reader will recognize immediately the majority of the concepts described in the next section, not as new types of knowledge associated with good programming and problem solving, but, rather, as simply making explicit that which was already there!

Whorfian Hypothesis. The belief that one should tease out tacit concepts and teach them explicitly rests to some degree on the Whorfian Hypothesis. In writing about language, Benjamin Whorf suggested that "language determines thought": That is, you can only think about something if you have a word for it. The strict, literal interpretation of Whorf's conjecture seems much too strong. However, a weaker claim that "language aids thought" surely has a great deal of appeal. That is, how can students learn a concept, when what they need to learn is not explicitly taught to them? The geometry proof example mentioned earlier is a case in point.

In what follows, then, a set of "vocabulary items" that, by and large, have remained as tacit knowledge in experts' heads, but that need to be explicitly taught, will be explained.

AN ENRICHED SET OF VOCABULARY TERMS

There is a lot of knowledge and many strategies that experts use that need to be made explicit and taught explicitly to students in introductory programming courses. There are two general categories of concepts: knowledge and strategies for using knowledge. The following list is not exhaustive; it is only meant to illustrate the basic theme.

Goals and Plans: the Heart of the Matter

What are the basic building blocks for analyzing problems and constructing programs? *Goals and plans*—stereotypical, canned solutions—are two key components in representing problems and program solutions. A chatty walk-through of a simplified problem decomposition will provide an intuitive understanding of the nature of goals and plans. More formal jargon will be introduced at the end of this section.

What, then, are the dominant elements in the following problem?

Averaging problem: Write a program that will read in integers and output their average. Stop reading when the value 99999 is input.

An average requires that a sum of the input numbers be taken, and that this sum must be divided by a count of the numbers summed. These two requirements are *goals* of the problem. In developing a programming solution for this problem, two additional goals (at least) must be put forth: The numbers must be input, and the average must be output. Expert programmers have developed standard techniques for realizing these four goals since they are common and appear in many problems. For example, programmers have a templatelike structure for reading in a stream of integers, summing them, and stopping when a sentinel value (99999) is input:

```
initialize a running total
ask user for a value
if input is not the sentinel value
      then
   add new value into running total
   loop back to input
```

This template is not language dependent—an expert can instantiate this template in Pascal, C, PL/1, etc. In our terminology, such canned solutions are *plans* ([25]). Programmers also have a looping plan for reading in and counting values that are input, again stopping when a sentinel value is input:

```
initialize a counter
ask user for a value
if input is not the sentinel value
      then
   increment counter
   loop back to input
```

A key observation that must be made is that the plan for realizing the summing goal and the plan for realizing the counting goal must be *merged*, since the data will be passed through only once:

```
initialize a running total
initialize a counter
ask user for a value
if input is not the sentinel value
      then
   add new value into running total
   increment counter
   loop back to input
```

Once the sum and count have been calculated, division must take place to arrive at the average. An expert programmer knows that division must be protected since a run-time error will occur if division by zero takes place:

```
if count is greater than 0 then
   then do the calculation
   else report the problem to the user
```

Note that this protection plan comes directly after the sum-and-count plans. This type of plan composition is *abutment*.

Although the preceding discussion may appear straightforward, the power of the goal/plan language can be seen in the following two examples:

- *Example 1:* Consider the following problem:

  Write a program that will output 'T' if all the inputs are 'T', but output 'F' if there is just one 'F' in the input sequence. Stop reading when a '#' is input.

  On the surface, this problem and the averaging problem appear different. However, this problem has an almost isomorphic *plan* structure to the averaging problem: A sentinel is used to stop reading the input; instead of accumulating totals as in the averaging problem, this problem requires a flag to be set/reset depending on the values input; since no input may be given, protection of the output must be implemented. In teaching programming—and problem solving in general—a key objective is to develop useful methods of abstraction: If every problem a student must solve appears to be new and different, then there is little reuse of experience. A hallmark of expertise is the ability to view a current problem in terms of old problems, so that solution strategies can be transferred from the old situation to the current situation. The language of goals and plans provides just such a language for characterizing problems and solutions ([29]).

- *Example 2:* Figure 1 depicts a buggy program generated by a student programmer for a more complicated version of the averaging problem. What would be a good description of this bug? Why might the student have created this bug? The goal/plan language again proves useful: The student has not merged a "valid-data-entry plan"—a filter loop plan that checks the validity of incoming data—with the sum-and-count plans. In fact, merging these plans requires considerable skill; analyses of literally thousands of student-generated programs suggest that students have significant difficulty in precisely those situations where merging must take place ([36])! Many more examples of the use of goals and plans can be found in [11] and [37].

It is not mere coincidence that a goal/plan language has utility in both developing and analyzing computer programs. Empirical data support the theoretical claim that experts in domains such as programming, chess, go, electronic circuits, mathematics, etc., have and use goals and plans in their problem-solving activity. The classic experiment in this area was carried out by Chase and Simon [6] with chess players:

> Master and novice chess players were shown a meaningful chess board, and asked to recall the pieces (once the board was taken away). The masters recalled more of the pieces than did the novices. Next, the masters and novices were shown a board in which the chess pieces were placed randomly on the board. What happened? The performance of the masters was the same as that of the novices.

Chase and Simon argued that the masters were better able to recall the meaningful chessboard because they were able to encode the pieces of the board in terms of *chunks* that represented meaningful units (e.g., a particular type of offense). In contrast, by definition novices do not have the degree of experience masters have, and thus the novices do not possess the domain specific chunks. Similar experiments have been carried out in the domain of programming [1, 19, 27] with comparable results. These sorts of experiments lend support to the theory that experts have and use domain specific *chunks*. These chunks are *plans* in the programming domain. Finally, the notion of a chunk (plan) is not useful only in technical subject domains: Cognitive psychologists have long posited the existence of

```
BUGGY VERSION

BEGIN
sum := 0;
count := 0;
writeln('please input a rainfall');
read(rainfall)
while rainfall < 0 do
  begin
    writeln('rainfall cannot be < 0;
             input again');
    read(rainfall);
  end;
while rainfall <> 99999 do
  begin
    sum := sum + rainfall;
    count := count + 1;
    writeln('please input a rainfall')
    read (rainfall);
  end;

etc.
END.
```

```
CORRECT VERSION

BEGIN
sum := 0;
count := 0;
writeln('please input a rainfall');
read(rainfall)
while rainfall <> 99999 do
  begin
    while rainfall < 0 do
      begin
        writeln('rainfall cannot be < 0;
                 input again');
        read(rainfall);
      end;
    sum := sum + rainfall;
    count := count + 1;
    writeln('please input a rainfall');
    read (rainfall);
  end;

etc.
END.
```

Problem: Read in integers that represent daily rainfall, and print out the average daily rainfall; if the input value of rainfall is less than zero, prompt the user for a new rainfall.

FIGURE 1. Sample Buggy Program

schemata as units of mental organization that play the same type of role in reading and writing stories as *plans* play in reading and writing computer programs (e.g., [3, 4, 9, 26]). Thus, there is a substantial body of psychological research that underlies the goal/plan idea.

Goals and plans can be given more concrete realizations. For example, the plan that "reads in and sums integers, stopping when 99999 is input" is called the SENTINEL-CONTROLLED RUNNING-TOTAL LOOP PLAN. Similarly, the plan that "reads in and counts integers, stopping when 99999 is input" is called the SENTINEL-CONTROLLED COUNTER-LOOP PLAN. The plan that protects the division in the average calculation is called the SKIP-GUARD PLAN. The intent is to convey some sense of the plan's goal in its name. Although plan names can be awkward, to say the least, they give *explicit labels* to structures that should be used repeatedly when solving problems. Assigning a label helps to establish a plan's reality. In fact, a whole library of plans has been built up to teach introductory programming. Students are explicitly taught to use the plans in this library and to construct plans of their own ([11, 30]). The goal/plan language is at the core of the proposed revised curriculum for teaching introductory programming.

Mechanisms and Explanations: the Products of the Programming Process

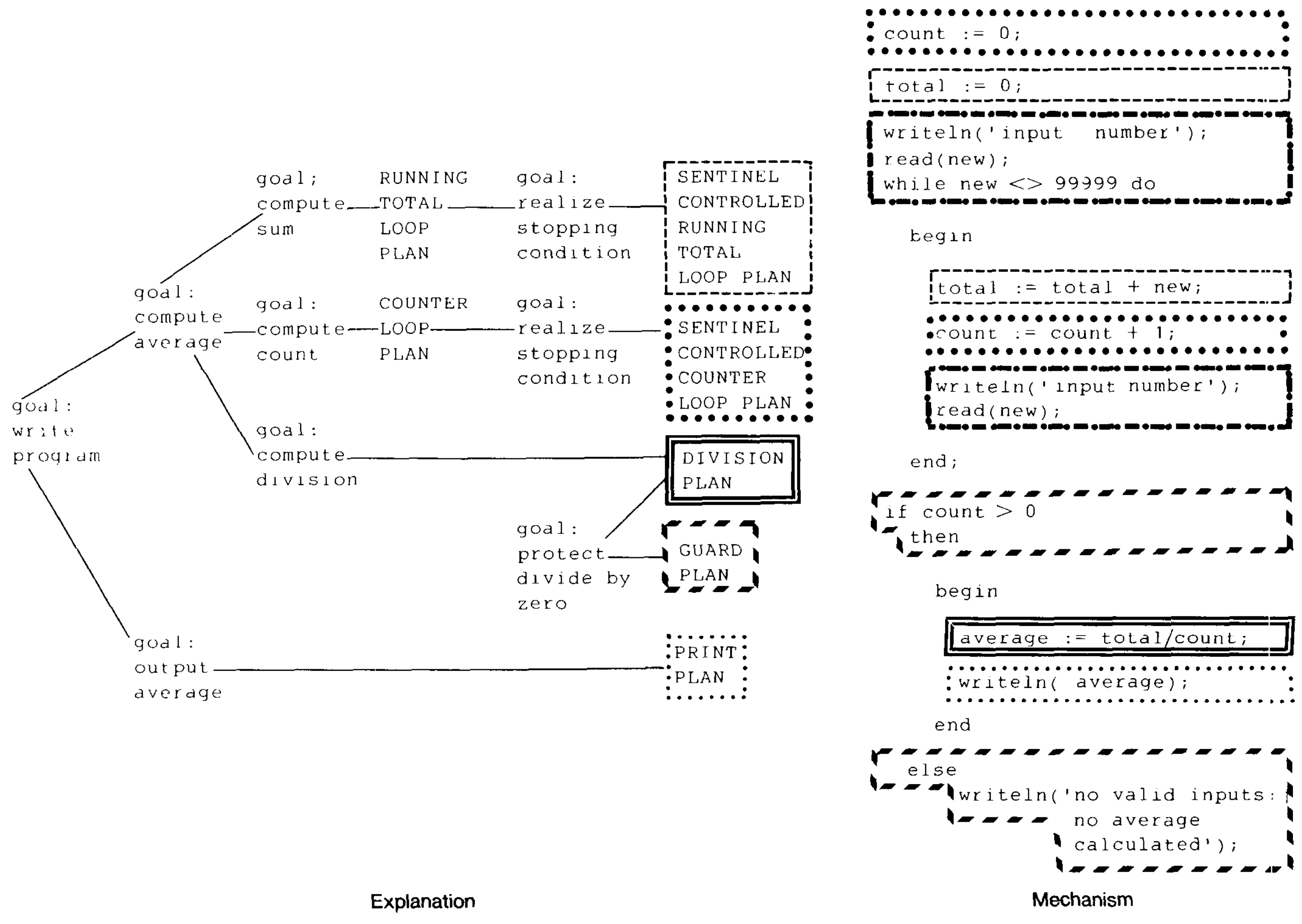
If introductory programming courses are to teach students something more than a job skill, the underlying abstractions of programming must be made explicit. That is, students must be taught what programming has in common with other problem-solving tasks. By focusing on programs as the output of the programming process, students are naturally led to think that what they have learned in "Computer Science 100" is relevant only to the production of programs. To facilitate the transfer of knowledge from "Computer Science 100" to other problem-solving activities, students must be taught explicitly that programming is a design discipline, and as such the output of the programming process is not a program per se, but rather an artifact that performs some desired function. Artifacts lead naturally to the concept of *mechanism*: A mechanism, whether it be instantiated as a transistor, a bolt, a gear, or a verbal instruction, specifies a chain of actions that, when set in motion, produces some desired effect. Artifacts that are instantiated in software are exceedingly malleable—change is the norm, not the exception. Programmers typically do not have the luxury of producing a one-shot artifact. Rather, they need to provide a trail of how and why the artifact was designed the way it was to enable the next programmer to carry out effective changes of the artifact. This trail of information is an *explanation.* The products of the programming process are really mechanisms and explanations; what students learn in an introductory programming course is the knowledge and skills to construct these objects.

The right half of Figure 2 is a Pascal program for the averaging problem. The program is the mechanism that instructs the computer as to how an average should be calculated.

How does the program in Figure 2 solve the desired problem? When executed by a computer (or by a human assuming the role of a computer), the program accepts numbers from a user, adds those numbers into an accumulator called `total`, etc.

The description in Figure 2 is only a partial representation of the mechanism. That is, in executing the mechanism, the computer turns a static program written on a piece of paper into a dynamic entity that exists over time. In this dynamic representation, notions such as the *causal relationship* between the statements become very important and must be used in describing how a program works: For example, the running total (`total`) and counter update (`count`) are performed *before* the next input value is read in; *after* the sentinel value is input, a test is made to see if the counter is greater than zero.

The left half of Figure 2 is the explanation for why that program computes the desired answer. The



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

Note that the relationship between a plan and its code is indicated by a box (e.g., the code that implements the SENTINEL-CONTROLLED RUNNING-TOTAL LOOP PLAN is enclosed by a box of dashes). In particular, the code realizing the merging of the SENTINEL-CONTROLLED RUNNING-TOTAL LOOP PLAN and the SENTINEL-CONTROLLED COUNTER-LOOP PLAN is enclosed by a box of dashes and dots.

Problem: Write a program that will repeatedly read in and sum data until a final stopping value of 99999 is input. Output the average of the numbers read in.

FIGURE 2. An Example: Mechanism and Explanation

goals of the English statement of the problem are identified, and stereotypical methods (plans) are used to attack and resolve each of these goals.

Why does the program effectively compute the average of the numbers read in? The SENTINEL-CONTROLLED RUNNING-TOTAL LOOP PLAN is used here *because* the problem requires that the loop be terminated following input of a specific value (the "sentinel-controlled" part of the plan), and *because* a running sum is needed to achieve the goal of summing the numbers entered by the user of the program. Also, *to* calculate the average, a merge of the SENTINEL-CONTROLLED COUNTER-LOOP PLAN with the SENTINEL-CONTROLLED RUNNING-TOTAL LOOP PLAN keeps track of the number of values input, *since* the input data will only be passed over once.

In the explanation, phrases such as *because* and *to* are used; these phrases signal the presence of information explaining the relationship between goals in the problem and plans instantiated in the program.

Just as the program description does not convey the entire mechanism, the goal/plan representation in Figure 2 does not include some important information. For example, how does the programmer know that the merging of the COUNTER-LOOP PLAN with the SENTINEL-CONTROLLED RUNNING-TOTAL LOOP PLAN will be correct? Or, why did the programmer choose not to read the numbers into an array, using a SENTINEL-CONTROLLED ARRAY INPUT LOOPING PLAN, and then process the data? This type of information must be encoded in a more complete explanation.

Understanding how to talk about mechanisms and explanations is still at an early stage. At least some vocabulary already exists for talking about mechanisms: language constructs, data and control flow, static and dynamic properties, etc. Playing an analogous role, the discussion of goals, plans, and their relationships begins to provide a concrete language for talking about explanations.

DESIGN STRATEGIES: CONSTRUCTING MECHANISMS AND EXPLANATIONS

Stepwise Refinement

Stepwise refinement is a planning technique that is intended to provide an orderly method for thinking about a problem. Almost all programming textbooks attempt to teach students how to use this technique. Typically students are told to *break down a problem into subproblems.* They are then shown how the author carries out this prescription on a set of examples. Most students can follow the argument quite well: They can understand the transition between each step. However, when asked to create a stepwise refinement for a new problem, most students are at a loss as to where to begin. When they attempt stepwise refinement on their own and confront the question of how to actually do it, troublesome issues arise: Why was the problem broken down into *those n* levels; why not *m* levels? And why were those particular routines used; why not some other routines? Textbooks typically do not answer those questions. Students are left to induce the strategies for doing stepwise refinement on the basis of the textbook's examples.

In teaching stepwise refinement, one crucial heuristic that everyone intuitively knows, but seldom makes explicit, must be added:

Break down a problem into subproblems, *on the basis of problems that you have already solved and for which you have canned (or almost canned) solutions.*

In teaching stepwise refinement, the image we try to paint is as follows:

Assume that you already possess a "barrel of canned solutions." Look up at the problem from that barrel of solutions, and try to see if some of those canned solutions can be used in the solution of the new problem. *Break the new problem down so that you can use those canned solutions.*

In other words, students must already possess the primitives into which the problem will be decomposed in order to carry out a stepwise-refinement strategy.

Canned solutions are programming plans—stereotypical methods for achieving goals. The objective of the stepwise-refinement planning strategy is to identify in the given problem statement various goals for which programming plans have already been developed. But what happens when a student does not have the plans that are relevant to a problem? For example, what happens when students just start learning to program? The answer is simple: Students cannot employ stepwise refinement as straightforwardly; they will do a substantial amount of floundering and searching, and tend to decompose a problem inappropriately. In fact, in a study with junior grade software designers this floundering behavior is precisely what was observed ([2]).

Recall that an explanation is made up in large measure by the goals and plans underlying the mechanism. Notice that the stepwise-refinement process actually produces the goal/plan decomposition that explains why the resultant mechanism is an effective solution to the problem: The goal/plan decomposition relates goals that need accomplishing to techniques (plans) for achieving those goals. As part of the goal/plan decomposition, the reasons why one goal and/or plan was chosen over another should also be made explicit; this too is part of the explanation.

Plan Composition Methods

Stepwise refinement describes the "macrostrategy" for developing a goal/plan decomposition for a given problem. However, rules for how plans at the "microlevel" should be woven together have also been identified. For example, in solving the averaging problem using stepwise refinement, there is a need for an output goal, to print out the computed value, and a calculate goal, to produce the average. How should the plans that realize these two goals be "glued together" (Figure 2)? Since the CALCULATION PLAN is completely finished before the OUTPUT PLAN can be performed, simply *abut* the two plans—the plan for computing the average outputs a value that is the input to the plan that prints out the value.

More generally, four strategies for gluing together plans have been identified:

- *Abutment:* Two plans are glued together back to front, in sequence, as illustrated in the averaging problem.
- *Nesting:* One plan is completely surrounded by another plan. For example, in the averaging program in Figure 2, the OUTPUT PLAN, the plan that realizes the goal of writing out the average, is nested within the SKIP-GUARD PLAN, which realizes the goal of not permitting division by zero to occur in the average calculation if no numbers are actually input.
- *Merging:* At least two plans are interleaved. For example, to solve the averaging problem, the plan that summed the input and the plan that counted the number of inputs were merged.
- *Tailoring:* Sometimes a canned plan that has already been developed is not quite what is needed in a problem. It must be modified to fit the particular needs of the situation. After all, we do call it "*soft*ware."

Plan merging is an exceedingly difficult activity to carry out effectively. (For example, see the buggy program in Figure 1, in which the student did not properly merge the VALID-DATA-ENTRY PLAN with the SENTINEL-CONTROLLED RUNNING-TOTAL LOOP PLAN and the SENTINEL-CONTROLLED COUNTER-LOOP PLAN.) In studying bugs that novice programmers make, it has been shown that when novices try to merge plans they almost invariably merge them incorrectly [37]. Such behavior is not all that surprising: Weaving together two or more plans requires great care and attention to details, and the ability to foresee all sorts of subtle interactions. From an instructional point of view, students must be alerted to the problems of producing programs in which plans are merged.

Providing students with four strategies for gluing plans together makes explicit a strategy for developing the explanation and mechanism components of a program at the microlevel. In particular, the explanation is furthered when students are told how the subgoals are stitched together to solve the overall goal, and the mechanism is furthered when the plans are instantiated in terms of actual programming language constructs.

Rules of Programming Discourse: Good Programming Practices

When plans are instantiated via actual code, there is still a great deal of flexibility as to how those plans should be realized: Different language constructs can realize the same plan, and/or the statements in a plan can have multiple orderings. For example, consider the program in Figure 3, which calculates and outputs a correct average. A programming instructor would certainly not want to give the writer of this program full credit in a test situation, since this program is written in a "poor style." Generally speaking, initializing variables to some strange value (e.g., typically one initializes a variable containing numeric values to either 0 or 1) is not a good habit. A goal/plan analysis of this program reveals the student's underlying intentions in performing these strange initializations:

```
Program Average
VAR count : INTEGER;
    sum, average, number : REAL;
BEGIN
  sum := -99999;
  count := -1;
  REPEAT
      writeln('please input a number');
      read (number)
      sum := sum + number;
      count := count + 1;
    UNTIL (number = 99999);
  average := sum/count;
  writeln('the average is:  ',average);
END.
```

This student program would not receive full credit because of its poor programming style.

FIGURE 3. Violating a Rule of Programming Discourse

In attempting to realize the goal of reading in, summing, and counting user input integers, the student has implemented the SENTINEL-CONTROLLED RUNNING-TOTAL LOOP PLAN with a `repeat` construct in Pascal. However, the `repeat` construct is not appropriate when implementing a loop in which the stopping value can be input on the first read; in such a case, the processing loop must not be executed even once. In Pascal, the `while` construct is the most appropriate loop for a SENTINEL-CONTROLLED RUNNING-TOTAL LOOP PLAN. (See [34].) Using a `repeat` construct allows the sentinel value to be added incorrectly into the running total (and also the counter is incorrectly updated). To compensate for this "off-by-one bug," the student has subtracted out the sentinel value (and the counter value) during the initialization phase of the program!

Surely, backing out a value to compensate for an off-by-one bug is not good programming practice.

Programming teachers, in grading a program such as the one depicted in Figure 3, often get the following response from students: "But my program runs. . . ." Often, teachers are hard put to say explicitly why a piece of code is not in good style, and the student is left in the dark as to why the program is so strictly graded. Aside from learning about stepwise refinement and plan composition methods, students must be taught to use *rules of programming discourse* ([12]), which are analogous to rules of conversational discourse. For example, consider the following interchange:

Mary: *Hi, John. Can you tell me what time it is?*
John: *Yes.*

John's response is not in accordance with accepted rules of discourse; he should have said something like this:

John: *Yes, it is 3:15.*

Analogously, there are rules of programming discourse that prescribe good programming practices. For example, the program in Figure 3 violated the following rule of programming discourse:

Do not do "double duty" with code, especially when the second function played by the code is obscure.

That is, the initialization of `Sum` and `Count` played two roles:

- *Role 1:* The variables were initialized to some starting value.
- *Role 2:* The starting values were chosen so as to compensate for the off-by-one bug that resulted from an incorrect looping structure.

Clearly, that second role was obscure: Students would have to understand the behavior of the loop to appreciate the programmer's choice of that particular initialization. Unfortunately, precious few programming textbooks (e.g., [14, 16]) explicitly teach students about the rules of programming discourse. Rather, students are expected to pick them up from observing examples of programs. However, as programming teachers know, students tend not to learn what counts as a program written in a good style.

From the computer's perspective, a program that violates the rules of discourse but computes the desired value is as good as a program that does not violate the rules of programming discourse. However, a program is not just for the computer: A program that follows the rules of discourse better enables a human reader to reconstruct the explanation that the program writer was following in initially developing the mechanism. Expert programmers use rules of discourse, albeit implicitly, in generating programs, and they assume that other programmers will do the same. Programs that violate the rules of discourse are often very difficult to comprehend. In fact, in a recent study, the performance of advanced programmers was reduced to that of novice programmers when the advanced programmers were asked to deal with programs that violated various rules of discourse ([31]).

Simulation: Making Explicit the Dynamic Properties of a Program

Although stepwise refinement and plan composition methods deal with putting pieces of solutions together, the resulting solution must in fact do what it is supposed to do. That is, the former three strategies deal with plans as *static objects*; however, the *dynamic*, run-time properties of the plans must be inspected, too, since unwanted interactions often arise. In studies, professional designers [2] and maintainers [18] have been observed repeatedly carrying out *simulations* of their design in progress, or the program being enhanced. (See also [13].) These professionals appear to seek and gain an understanding of the casual interactions among the components. The feedback provided by simulation then enables them to rework their design, in the case of the designers, or to be sensitive to a possible trouble spot, in the case of the maintainers. Moreover, those designers and maintainers who did *not* carry out simulations did *not* develop an effective design or an effective enhancement.

Simulation, then, is a strategy for uncovering unwanted causal interactions between components that occur as a result of the dynamic aspects of a program. Simulation has its detractors (e.g., [22]);

however, if simulation is not to be used, then some other means for understanding these dynamic effects must be developed. Until then, the simulation strategy should be explicitly taught to students.

CONCLUDING REMARKS

Teachers of introductory programming clearly know that teaching the syntax and semantics of a programming language is not enough. Students should be given *explicit* instruction in "vocabulary terms" such as mechanism, explanation, goal, plan, rules of programming discourse, plan composition methods, etc. As supported by a wide range of empirical studies (e.g., see [8, 28, 32]), these notions are, in fact, what expert programmers know and use. Moreover, these notions are applicable not only in a programming context. For example, people are constantly developing mechanisms and explanations to deal with problems arising in daily life. If a revised curriculum of this type enables students to transfer what they learned from programming to other problem-solving situations, then programming will be established as not just a vocational skill, but rather as a vehicle for learning effective problem solving—and the intense excitement surrounding programming for "just plain folks" would be justified!

Acknowledgments. The research upon which this article is based has involved significant contributions from a number of individuals: Beth Adelson, Jeff Bonar, Kate Ehrlich, Lewis Johnson, Stan Letovsky, David Littman, Jeannine Pinto, Warren Sack, Jim Spohrer, and Beverly Woolf. Important, helpful comments on earlier drafts of this article were made by the reviewers, as well as members of my research group, the Cognition and Programming Project, at Yale; assistance from David Littman, Jim Spohrer, and Warren Sack was especially useful. To all these people, I say a most heartfelt "thank you."

REFERENCES

1. Adelson, B. Problem solving and the development of abstract categories in programming languages. *Mem. Cognition 9* (1981), 422–433.
2. Adelson, B., and Soloway, E. The role of domain experience in software design. *IEEE Trans. Softw. Eng.* (Nov. 1975).
3. Bartlett, F.C. *Remembering.* University Press, Cambridge, Mass., 1932.
4. Bower, G.H., Black, J.B., and Turner, T. Scripts in memory for text. *Cognitive Psychol. 11* (1979), 177–220.
5. Brooks, R. Towards a theory of the comprehension of computer programs. *Int. J. Man-Mach. Stud. 18* (1983), 543–554.
6. Chase, W.C., and Simon, H. Perception in chess. *Cognitive Psychol. 4* (1973), 55–81.
7. Collins, A. Explicating the tacit knowledge in teaching and learning. Tech. Rep. 3889, Bolt, Beranek and Newman, Cambridge, Mass., 1978.
8. Curtis, B. *Tutorial: Human Factors in Software Development.* IEEE Computer Society, 1985.
9. Graesser, A.C. *Prose Comprehension beyond the Word.* Springer-Verlag, New York, 1981.
10. Howe, J.A.M., O'Shea, T., and Plane, J. Teaching mathematics through Logo programming. Tech. Rep. 115, Artificial Intelligence, Univ. of Edinburgh, Scotland, 1979.
11. Johnson, W.L. Intention-based diagnosis of errors in novice programs. Ph.D. thesis 246, Dept. of Computer Science, Yale Univ., New Haven, Conn., 1985.
12. Joni, S., and Soloway, E. But my program runs! Discourse rules for novice programmers. *J. Educ. Comput. Res.* To be published.
13. Kant, E., and Newell, A. Problem solving techniques for the design of algorithms. Tech. Rep. CMU-C S-82-145, Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa., 1982.
14. Kernighan, B., and Plauger, P. *The Elements of Style.* McGraw-Hill, New York, 1978.
15. Larkin, J., McDermott, J., Simon, D., and Simon, H. Expert and novice performance in solving physics problems. *Science 208* (1980), 140–156.
16. Ledgard, H., Hueras, J., and Nagin, P. *Pascal with Style: Programming Proverbs.* Hayden Book Co., Rochelle Park, N.J., 1979.
17. Linn, M.C. The cognitive consequences of programming instruction in classrooms. *Educ. Res. 14*, 5 (1985), 14–29.
18. Littman, D., Pinto, J., Letovsky, S., and Soloway, E. Software maintenance and mental models. In *Empirical Studies of Programmers*, E. Soloway and S. Iyengar, Eds. Ablex, New York, 1986.
19. McKeithen, K.B., Reitman, J.S., Rueter, H.H., and Hirtle, S.C. Knowledge organization and skill differences in computer programmers. *Cognitive Psychol. 13* (1981), 307–325.
20. Michener, E.R. Understanding understanding mathematics. *Cognitive Sci. 2* (1978), 283–327.
21. Papert, S. *Mindstorms, Children, Computers and Powerful Ideas.* Basic Books, New York, 1980.
22. Parnas, D. Software aspects of strategic defense systems. *Am. Sci. 73* (1985), 432–440.
23. Pea, R., and Kurland, D. Logo programming and the development of planning skills. Tech. Rep. 16, Center for Children and Technology, Bank Street College of Education, New York, 1984.
24. Resnick, L. Mathematics and science learning: A new conception. *Science 220* (1983), 477–478.
25. Rich, C. Inspection methods in programming. Tech. Rep. AI-TR-604, AI Laboratory, MIT, Cambridge, Mass., 1981.
26. Schank, R.C., and Abelson, R. *Scripts, Plans, Goals and Understanding.* Lawrence Erlbaum Associates, Hillsdale, N.J., 1977.
27. Shneiderman, B. Exploratory experiments in programmer behavior. *Int. J. Comput. Inf. Sci. 5*, 2 (1976), 123–143.
28. Shneiderman, B. *Software Psychology: Human Factors in Computer and Information Systems.* Winthrop Publishers, Cambridge, Mass., 1980.
29. Soloway, E. From problems to programs via plans: The content and structure of knowledge for introductory LISP programming. *J. Educ. Comput. Res.* (Summer 1985).
30. Soloway, E. Programming and problem solving in Pascal. In preparation.
31. Soloway, E., and Ehrlich, K. Empirical studies of programming knowledge. *IEEE Trans. Softw. Eng. SE-10*, 5 (1984), 595–609.
32. Soloway, E., and Iyengar, S. *Empirical Studies of Programmers.* Ablex, New York, 1986.
33. Soloway, E., Lochhead, J., and Clement, J. Does computer programming enhance problem solving ability? Some positive evidence on algebra word problems. In *Computer Literacy*, R. Seidel, B. Hunter, and R. Anderson, Eds. Academic Press, New York, 1982, pp. 171–215.
34. Soloway, E., Ehrlich, K., Bonar, J., and Greenspan, J. What do novices know about programming? In *Directions in Human–Computer Interactions*, A. Badre and B. Shneiderman, Eds. Ablex, New York, 1982.
35. Spohrer, J., and Soloway, E. Novice mistakes: Are the folk wisdoms correct? *Commun. ACM 29*, 7 (July 1986), 624–632.
36. Spohrer, J., and Soloway, E. Analyzing the high-frequency bugs in novice programs. In *Empirical Studies of Programmers*, E. Soloway and S. Iyengar, Eds. Ablex, New York, 1986.
37. Spohrer, J., Soloway, E., and Pope, E. A goal/plan analysis of buggy Pascal programs. *Hum.-Comput. Interaction 1*, 2 (1985).

CR Categories and Subject Descriptors: D.m [**Miscellaneous**]: *software psychology*; K.3.2 [**Computers and Education**]: Computer and Information Science Education—*curriculum*
General Terms: Design, Human Factors, Languages

Author's Present Address: Elliot Soloway, Cognition and Programming Project, Dept. of Computer Science, Yale University, P.O. Box 2158, New Haven, CT 06520.