Tacit Programming Knowledge

Elliot Soloway
Kate Ehrlich

Cognition and Programming Project
Computer Science Dept.
Yale University
New Haven, Ct. 06520

1. Introduction

[1] The goals of the Cognition and Programming Project at Yale University are:

- empirically explore the issues surrounding programming
  - what does an expert programmer know, and how does this compare to what a novice does (and doesn't) know [Soloway et al., 1982a, Ehrlich & Soloway, 1982],
  - what makes a programming language construct "cognitively appropriate" — and can we design such constructs [Soloway et al., 1981a]
  - what is the relationship between algebra knowledge and programming knowledge [Erhlich et al., 1982, Soloway et al., 1982]
- build AI based computer environments which can aid the novice programmer in learning to program [Soloway et al., 1981b, Soloway et al., 1982b].

In this short paper, we will describe some techniques we employ to investigate the first issue: what do programmers know.

2. Programming Plans: The Tacit Knowledge in Programming

A number of researchers have replicated the chess experiments of deGroot [deGroot, 1965] and Chase & Simon [Chase and Simon, 1973] in the domain of programming; consistent with those earlier experiments with master and non-master chess players, it appears that expert programmers also have more knowledge which is more highly chunked than novice programmers [Shneiderman, 1976, Adelson, 1981, McKeithen, Reitman, Rueter and Hirtle, 1981],

Building on this work, our goal is to identify the *specific* knowledge which expert programmers appear to have and use. The problem is that experts are often unaware of using this sort of knowledge — hence the term *tacit* knowledge. Collins [Collins, 1978], Larkin [Larkin et al., 1980], Rissland [Rissland, 1978], etc. have argued for the importance of tacit knowledge in various domains; our objective is to identify the tacit knowledge in programming.

To this end, we have developed a first order theory of the programming knowledge underlying simple looping programs which we feel experts have and use. Knowledge in this theory is encoded in terms of *plans*: stereotypic chunks of knowledge. For example, we posit that there are control flow plans and variable plans; in Figure 1, we would suggest that the body of the program is an implementation of the Running Total Loop Plan: new values are successively generated, in this case by a Read, and are added to a Running Total Variable, Sum. Also, there is Counter Variable, Count, which keeps track of the number of numbers generated. Our approach to programming plans is similar in spirit to that of Rich [Rich, 1980] and Waters [Waters, 1979].

```
Problem  Read  in  a set of integers and print out their average  Stop
reading numbers when the number 99999  is  seen.  Do  NOT  include  the
99999 in the average
```

[1] This work was supported in part by the National Science Foundation, under NSF Grant SED-81-12403.

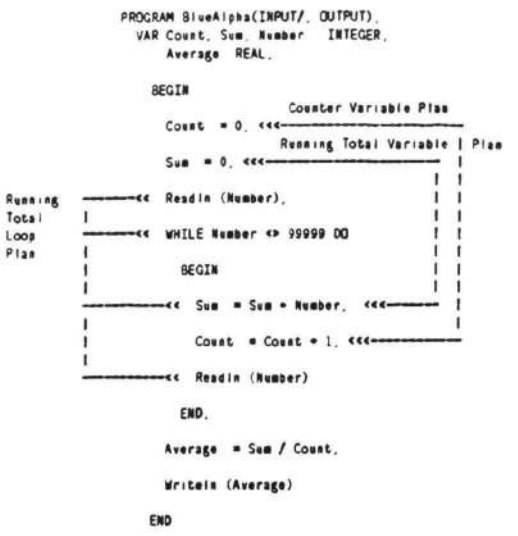

Figure 1: Examples of Plans

How does one go about testing a theory of this sort? Simply asking programmers whether or not they use the Running Total Loop Plan would not be too illuminating: the claim is that they are often unaware of having and using this type of knowledge. Below we describe techniques which we have found useful in this regard.

3. The Fill-in-the-blank Technique

The first technique we have used draws on work done in exploring the reality of scripts in text understanding. For example Bower, Black and Turner found that, in response to questions about a story, subjects would "fill in" from their "script" knowledge, information which was not explicitly given in the text. Similar in flavor, we give programmers a program in which a line of code has been left out, and ask them to fill it in. We purposely do not tell the subjects what the program is supposed to do; our objective is to have subjects use their experience with previous programming problems in order to recognize what line of code is most appropriate in the particular situation. If subjects didn't have plan structures, we would expect the answers they give to be arbitrary, and thus vary wildly from subject to subject. As we discuss below, the answers which novices give typically do vary significantly, while the answers which advanced programmers give do in fact exhibit a significant degree of consistency.

We also add an extra twist to the above design in order to more precisely home in on plan knowledge. We create two versions of the test program; in the first version, the information needed to fill in the blank line is more or less unambigious, while the second version contains *conflicting* information. For example, the programs in Figures 2 and 3 are both intended to produce the square root of N. Since N is in a loop which will repeat 10 times, 10 values will be printed out. The question is: how should N be set? The technique will be to compare the performance of programmers on the program which does not contain the plan conflict (Figure 2), with their performance on the program which does contain the conflict (Figure 3).

Please complete the program fragment given below by filling in the blank line (indicated by a box). Fill in the blank with a line of Pascal code which in your opinion best completes the program.

```
program VioletAlpha(Input/, Output).
  var N  real.
      I   integer.
  begin
  for I  = 1 to 10 do
    begin
    ------------------------------
    |                            |.
    ------------------------------
    if N < 0 then N  = -N.
    Writeln ( Sqrt(N) ).
       (* Sqrt is a built-in
         function which returns the
         square root of its argument*)
    end.
  end
```

Figure 2: Problem VioletAlpha:
The influence of a single Plan

In the program in Figure 2, N is a New Value Variable, since its function is merely to hold successive values. The plan for this type of variable does not present an overriding constraint on how it should be set in the blank line: a Read(N) or a N := N + SomeValue would both be acceptable. However, context does provide a strong constraint. Notice the If test preceding the Sqrt(N) instantiates the *"guard a portion of a program from improper data"* plan by protecting the Sqrt from negative integers (the Sqrt function can only work on positive integers). This test specifies an important constraint: N should take on values that could possibly be negative, otherwise the If test would be totally superfluous. Thus, N should

Please complete the program fragment given below by filling in the blank line (indicated by a box). Fill in the blank with a line of Pascal code which in your opinion best completes the program.

```
program VioletBeta(Input/, Output).
  var N  real.
      I   integer.
  begin
  N  = 0.0
  for I  = 1 to 10 do
    begin
    ------------------------------
    |                            |.
    ------------------------------
    if N < 0 then N  = -N.
    Writeln ( Sqrt(N) ).
       (* Sqrt is a built-in
         function which returns the
         square root of its argument*)
    end.
  end
```

Figure 3: Problem VioletBeta:
A conflict between Plans

not be set via an assignment statement to some simple function of N and/or the index variable I, e.g., N:=N+I, N:=I, N:=N+1. Rather, by setting N via a Read statement, negative values have the possibility of entering the program. This argument is based on a principle of tacit communication which states: ***include only necessary code in a program.*** By including a test for negative values, an experienced programmer is informing the reader that it is possible that such numbers could be generated; if such numbers could not possibly enter the program, then the inclusion of this test would violate this unwritten rule of communication.

The blank line in the program in Figure 2 is strategically placed: we wanted to explore the degree to which programmers are sensitive to the contextual relationship which obtains between the guard plan and the initialization aspect of New Value Variable Plan.

Program VioletBeta in Figure 3 is exactly the same as that in Figure 2 except that now N is given a value of 0 before the loop. Previously the New Value Variable Plan was neutral with respect to how N should be set. However, since N was initialized via an assignment statement to 0, the general rule of relating initialization to update should come into play, and direct that N be updated via an assignment. On the other hand, the If test, which realizes the "guard plan" and protects the square root operation, still sets up the expectation that N will be read in. If N will be set via a Read in the loop, the setting of N to 0 initially is superfluous. Thus, in Program VioletBeta we have purposely created a situation in which two plans are in conflict: the New Value Variable Plan expects N to be updated via an assignment, since it was initialized via an assignment, but the guard plan on the Sqrt operation expects that N will updated via a Read, so as to permit negative values to enter the program.

We felt that novices, with their limited experience, would be more sensitive to the constraint that obtains between a variable's initialization and update, as compared to the constraint that obtains between a guard plan and a variable's update. Hence, we predicted that the proportion of novices who Read in the value of N would decline when there was a conflict between plans. On the other hand, we felt that more advanced programmers would have had sufficient experience in both, and know when each is most appropriate, e.g., non-novices would realize that the test for a negative N should take precedence over the initialization of N to 0, since the "guarding" of the input is usually very important to the correct running of the program. Thus, we predicted that non-novices would fill in the blank with Read(N) equally often in both versions of the problem.

	NOVICES			NON-NOVICES	
	ALPHA no conflict	BETA conflict		ALPHA no conflict	BETA conflict
Category 1: Set N via Read	44	30		20	26
Category 2: Set N via assignment	7	15		4	4
	chi-squared = 5.20, p < 0.05			chi-squared < 1, N.S.	

Table 1: Fill-in-the-blank Responses

The responses of novices and non-novices on these programs, shown in Table 1, support our predictions. Non-novices chose to set N via a Read in the non-conflict case (VioletAlpha), and also chose to set N via a Read in the conflict case (Beta). This is consistent with our hypothesis that non-novices could use contextual information — the guard plan constraint — to override the variable plan constraint in the conflict case. In contrast, novices chose Read significantly less often in the conflict case than in the non-conflict case (chi-squared= 5.20, $p < 0.05$). This is consistent with our hypothesis that novices were more influenced by the familiar variable plan constraint than by the less familiar, contextual guard plan constraint.

4. Reading Time Studies

We also wanted to see how reading time was effected by the no conflict/conflict situations. Thus, we carried out studies which tracked the time a programmer started reading the program to the time he began to fill in the blank. For the programs in Figures 2 and 3, we found that novice programmers took effectively the same amount of time to respond in Program VioletAlpha as in Program VioletBeta (see Table 2). In contrast, while the advanced programmers responded quicker than the novices on Program VioletAlpha, they took significantly longer than the novices to respond to Program VioletBeta. We feel these data also support our hypothesis that Program VioletBeta contained a conflict between plans, to which only the advanced programmers were sensitive, while there was no similar conflict in Program VioletAlpha.

Alpha	Beta	
109	150	Novices n s
72	193	Non-novices p < .05

Mean Reading Times
In Seconds

Table 2: Reading Times Study

5. Concluding Remarks

Tapping into the tacit knowledge which programmers seem to have and use is a complex task. The basis for our experimental methods rests squarely on our, albeit preliminary, theory of programming knowledge. That is, we needed the theory in order to create the programs which serve as our stimulus materials. We are currently working on extending that theory to more complex programming problems and constructions.

We are also carrying out fill-in-the-blank studies and reading time studies with unplan-like programs, and programs which contain bugs. One objective is these studies is to explore the extent to which programs can be perturbed and still have people recognize the correct underlying intentions.

A longer range goal is the development of measures of program complexity based not just on features of the program text itself, but rather on the cognitive demands which the program makes on the programmer. Black and Sebrechts [Black & Sebrechts, 1981] have argued quite persuasively that measures of program complexity based on textual features (e.g. number of operations, length of variable names) cannot be effective measures, in the same way that the old measures of reading complexity, based also on textual features, were not effective measures. Such measures can capture only "surface" information. In contrast, effective measures must be based on the types and number of inferences which a programmer must make in order to understand the program. By cataloging the types of inferences which programmers do make, we have taken a first step in this enterprise.

Acknowledgements

We would like to thank Chuck Rich for his help in developing the stimulus materials used in this experiment, John Leddo for running the reading time study, and Joost Breuker and Valerie Abbott for their help in analyzing the data.

References

Adelson, B. Problem Solving and the Development of Abstract Categories in Programming Languages. *Memory and Cognition*, 1981, *9*, 422-433.

Black, J.B. & Sebrechts, M.M. Facilitating human-computer communication. *Applied Psycholinguistics*, 1981, *2*, 149-178.

Chase, W.C. and Simon, H. Perception in Chess. *Cognitive Psychology*, 1973, *4*, 55-81.

Collins, A. *Explicating the Tacit Knowledge in Teaching and Learning*. Technical Report 3889, Bolt Beranek and Newman, Cambridge, Mass., 1978.

deGroot, A.D. *Thought and Choice in Chess*. Paris: Mouton and Company 1965.

Ehrlich, K., Soloway, E. *An Empirical Investigation of the Tacit Plan Knowledge in Programming*. Technical Report 82-30, Dept. of Computer Science, Yale University, 1982.

Ehrlich, K., Soloway, E., Abbott, V. *Styles of Thinking: From Algebra Word Problems to Programming Via Procedurality*. Cognitive Science Society, Univ. of Michigan, Mich., 1982.

Larkin, J., McDermott, J., Simon, D. and Simon H. Expert and Novice Performance in Solving Physics Problems. *Science*, 1980, *208*, 140-156.

McKeithen, K.B., Reitman, J.S., Rueter, H.H., Hirtle, S.C. Knowledge Organization and Skill Differences in Computer Programmers. *Cognitive Psychology*, 1981, *13*, 307-325.

Rich, C. *A Library of Plans with Applications to Automated Analysis*. Technical Report 294, MIT AI Lab, 1980.

Rissland, E. Understanding Understanding Mathematics. *Cognitive Science*, 1978, *2(4)*, .

Shneiderman, B. Exploratory Experiments in Programmer Behavior. *International Journal of Computer and Information Sciences*, 1976, *5,2*, 123-143.

Soloway, E., Lochhead, J., Clement, J. Does Computer Programming Enhance Problem Solving Ability? Some Positive Evidence on Algebra Word Problems. In R. Seidel, R. Anderson, B. Hunter (Eds.), *Computer Literacy*, New York, NY: Academic Press, 1982.

Soloway, E., Bonar, J. and Ehrlich, K. *Cognitive Factors in Programming: An Empirical Study of Looping Constructs*. Technical Report 81-10, Department of Computer Science, University of Massachusetts , 1981.

Soloway, E., Woolf, B., Barth, P., and Rubin, E. *MENO-II: Catching Run-Time Errors in Novice's Pascal Programs*. International Joint Conference on Artificial Intelligence, Vancouver, B.C., 1981.

Soloway, E., Ehrlich, K., Bonar, J., Greenspan, J. What Do Novices Know About Programming? In Directions in Human-Computer Interactions, B. Shneiderman and A. Badre, Eds., Ablex, Inc. in press.

Soloway, E., Rubin, E., Woolf, B., Bonar, J. MENO-II: A AI-CAI Programming Tutor. Proceedings of the ADCIS Conference, Vancouver, B.C., in press.

Waters, R.C. A Method for Analyzing Loop Programs. *IEEE Trans. on Software Engineering*, May 1979, *SE-5*, 237-247.