THE BRIDGE FROM NON-PROGRAMMER TO PROGRAMMER

Jeffrey Bonar*
Elliot Soloway**

COINS Technical Report 83-18

*Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts, 01003

**Department of Computer Science
Yale University
P.O. Box 2158
New Haven, Connecticut 06520

Appeared in <u>Proceedings of the National Educational Computing Conference, 1984</u>

This work was supported by the National Science Foundation under NSF Grant SED-81-12403. Any opinions, findings, conclusions, or recommendations expressed in this report are those of the author, and do not necessarily reflect the views of the U.S. Government.

Abstract

Non-programmers bring to the learning of programming strategies that they have developed to solve day-to-day problems. Interestingly, programming language constructs often require strategies that conflict with these non-programming strategies. One can predict quite confidently that in these situations, novice programmers will have difficulty — and bugs in their programs will result. In this paper, we present evidence for the existence of *natural language specification strategies* that novices bring to programming, in the form of abstracts from verbal protocols taken from novice programmers as they are trying to program. These transcripts highlight the types the bugs and misconceptions that result when there is a mismatch between the strategies required by programming language constructs and the strategies that non-programmers bring to programming.

1. Introduction

Any interesting computerized task soon involves programming. Experience with statistics packages, word processing, and even microwave ovens shows that we always want our systems to be able to follow a step-by-step specification involving decisions and repeated actions. Even with a very intelligent computerized assistant, we would like to give it detailed instructions at an appropriate level of abstraction.

This ubiquity of programming presents a problem, however. It is widely known that programming, even at a simple level, is a difficult activity to learn. The seventies saw a revolution in the way that programming was practiced and taught. The phrase "structured programming" summarizes a whole new level of attention to the design, implementation, and testing of computer programs; attention that changed much of the thinking about how programming should be taught. We are now much clearer about how to teach powerful and effective programming, but do we know how to make programming maximally available? Do we really know how to make *programmers* ubiquitous? Apparently not:

> *Based on exam grades and on our studies (e.g., Soloway et al.1982), we estimate that more than 40% of the conscientious students never really understand the rudiments of programming.*

What's missing -- what is the way to build a bridge between non-programmer and programmer?

We begin by noting that programming is a cognitive skill, much like understanding math [Rissland, 1978] or solving physics problems [DiSessa, 1982]. Drawing from recent work in Cognitive Science, we are using a new methodology for looking at the acquisition of cognitive skills. There are two key parts to our methodology. First, we look *in great detail* at the errors of novice programmers. As experienced programmers, our tendency is to look at errorful novice programs only seeking to eliminate bugs as soon as possible. In our work, we have tried to see what was the specific (mis)information used by the novice to produce the bug. This is quite a powerful view: novice programmers have deep and interesting misunderstandings, as you will see below.

Second, in understanding the novice misunderstandings we try to view the situation in terms of specific bundles of knowledge possessed by the expert but not by the novice. What we find is that there is often a level of *tacit* knowledge [Collins, 1978] that is not explicitly taught, and often not even explicitly acknowledged.

In this report we present evidence for one major source of difficulty: *current programming languages often do not accurately reflect the problem solving strategies that non-programmers bring to programming.* That is, non-programmers have developed *natural language strategies* in order to cope with day to day problems. While experienced programmers have learned to modify or replace these strategies with ones more appropriate to computer programming, novices are often confused at this very basic level. Step-by-step natural language specification provides powerful intuitions for novice programmers using a programming language. We hypothesize that these intuitions take the form of bundles of knowledge we call *plans* - regular but flexible techniques for specifying how to accomplish a task. Programming knowledge also involves plans [Soloway et al, 1982] [Waters, 1979]. While an individual programming language plan may have many lexical and syntactic similarities to a corresponding natural language plan, the two plans often have incompatible semantics and pragmatics. Many novice programmer's misconceptions derive directly from these incompatibilities.

In this brief report we will show examples of natural language plans and programming language plans. We will then analyze transcripts of thinking aloud protocols taken with novice programmers who use natural language plans while attempting to write a computer program. We conclude with a brief discussion of the implications of this work for teaching programming.

Before proceeding, a methodological point is worth making. While the theoretical framework in this paper — and its conclusions — are the same as those in other papers we have published (e.g., Soloway et al. 1982, Soloway et al. 1981a), there is a key methodological difference between this paper and the others: previously we used statistical arguments based on written tests with large numbers of programmers as evidence for/against our hypotheses; however, in this paper we use anecdotes from thinking aloud protocols taken from individual programmers as evidence for our hypotheses. That is, in previous papers we have made claims about what we think our subjects were thinking. Statistical evidence is an indirect test of these sorts of claims. Verbal reports of subjects as they are programming provide a more direct window into the thought processes of our subjects. Thus, this paper provides the needed closure for our hypotheses: we have converging evidence from statistical-type group studies and from verbal reports with individual programmers that support our theory of the role of programming plans and the role of natural language plans (i.e., pre-programming plans) in programming.

2. Natural Language Plans and Programs

Consider the following problem:

> ***Problem 1:*** Please write a set of explicit instructions to help a junior clerk collect payroll information for a factory. At the end of the next payday, the clerk will be sitting in front of the factory doors and has permission to look at employee pay checks. The clerk is to produce the average salary for the workers who come out of the door. This average should include only those workers who come out before the first supervisor comes out, and should not include the supervisor's salary.

The following natural language specification for this problem, written by one of our subjects, is typical:

1. Identify worker, check name on list, check wages
2. Write it down
3. Wait for next worker, identify next, check name, and so on
4. When super comes out, stop
5. Add number of workers you've written down
6. Add all the wages
7. Divide the wages by the number of workers

There are several natural language specification plans used here. Note how steps 1 through 4 specify a loop: steps 1 to 3 describe the first iteration of the loop, indicating repetition with the phrase "and so on". Step 4 adds a stopping condition, assuming that this condition will act as a *demon,* always watching the action of the loop for the exit condition to become true. The specification also assumes *canned procedures* for counting inputs, step 5, and for summing a series of numbers, step 6. Note however, that these two procedures are both denoted with the word "add".

Now focus on the two actions performed in steps 1 and 2. The plan to describe these actions is *get a value (step 1), and process that value (step 2).* This plan is nearly universal in this sort of description. Unfortunately, many programming languages support a far less natural plan: *process the last value, get the next value.* Below, we discuss this problem in detail.

3. Examples of Novice Programming Difficulties

To show how the conflict in strategies effects novice programmers, consider a problem analogous to problem 1, but simpler and explicitly of a programming nature:

> ***Problem 2:*** Write a program which repeatedly reads in integers until it reads they integer 99999. After seeing 99999, it should print out the *correct* average. That is, it should not count the final 99999.

In Pascal, a popular novice programming language, the preferred correct solution to Problem 2 is:

```
PROGRAM Problem_2 Expert;
   VAR Count, Total, New : INTEGER;
   BEGIN
   Count := 0; Total := 0;
   Read (New);
   WHILE New <> 99999
     DO BEGIN
        Count := Count + 1;
        Total := Total + New;
        Read (New)
        END;
   IF Count > 0
      THEN writeln ('Average =',Total/Count)
      ELSE writeln ('No data.')
   END.
```

Notice the peculiar WHILE loop construction. Because a WHILE loop tests only at the top of the loop, it is necessary to have a Read both above the loop and at the bottom of the loop. Within the loop we see the plan *process the last value, read the next value*. This plan is part of the knowledge used by experienced Pascal programmers. Data we have gathered suggests that novice programmers do not easily acquire such a plan (Soloway et al. 1982, Soloway et al. 1983).

First of all, novices often want the WHILE to have a demon like structure. Consider, for example, the following transcript:

Subject:	How do I get [the WHILE loop][1] to do that over again? See, I guess I don't know, I thought I had it. What happens now, how do I get it to go back? ... I say to myself, why would it do [the WHILE test] after [the last line of the loop body]? It seems to me that it would do it as soon as the [variable tested in the WHILE condition] changes. ...
Interviewer:	So how will the WHILE statement behave?
Subject:	Again, total guess here, I'm saying the WHILE statement, here's a logical guess ... everytime [the variable tested in the WHILE condition] is assigned a new value, the machine needs to check that value ...

The subject's "logical guess" is that the condition in the WHILE loop is being *continually* tested, and that the loop will be exited as soon as the condition is true. This is not an unreasonable interpretation; it is is consistent with the meaning of "while" in English phrases such as "while you are on the highway, watch for the Northfield sign". In a group study with novice programmers, we found that 34% had this type of misconception about the test in the WHILE loop (Soloway et al [1981a]).

[1]Text in square brackets ("[" and "]") describes items pointed to by the subject. Usually the subject's actual words were "this", "here", or something similar. The brackets and words were used to make the transcripts more readable.

Novices also try to implement the *get a value, process that value* plan, even though they are programming in Pascal. Consider, the following novice program fragment,

```
  ...
VAR Count, Total, I : INTEGER;
   BEGIN
   Count := 0
   Total := 0
   Writeln ('Enter integer')
   Read (I)
   WHILE I <> 99999 DO
      BEGIN
      Count := Count + 1
      Total := Total + I
      Read ~~(I)~~ the subject has crossed out this line out
      END                  after writing it down
  ...
```

and a transcript of the subject discussing this program:

Subject: If I put a number in [at the top of the loop], it comes through [the loop body]. I don't think I want [the inside Read] read again, I want it read up [at the top of the loop] ... If I read it [at the bottom of the loop body], what's that going to do for me? It's not going to do anything for me. OK, if I come out of the loop, having entered [a value], finish all [the loop body], then if I read in another one [points to Read above the WHILE, traces a flow from that outside Read down through the loop]. I guess what I need to figure out is how do I get back up here [points to the Read above the WHILE].

The subject wants to put the Read at the top of the loop, making the test in the middle of the loop. This reflects the *get a value, process that value* plan. In a separate study Soloway, et al. [1983] show that a new Pascal looping construct supporting this plan significantly improved novice and intermediate performance with Problem 2.

Conflicts and problems can occur even when the novice appears to fully understand a program fragment. Consider, for example, the following novice. She is writing pseudo-code for the following problem:

Problem 3: Write a program which reads in 10 integers and prints the average of those integers.

After working on the problem for a few minutes, she had written the following:

```
Repeat
(1) Read a number (Num)
        (1a) Count := Count + 1
(2) Add the number to Sum
        (2a) Sum := Sum + Num
(3) until Count := 10
(4) Average := Sum div Num
(5) writeln ('average = ', Average)
```

Leaving aside some inconsistent pseudo-code notation, this is correct. At this point, the interviewer asks whether the statement on line 1a is the "same kind of statement" as that on line 2a. The subject seems to understands the role these two lines play in the program. She also recognizes the need for other associated statements to carry out those roles. Nonetheless, it appears that she thinks the Pascal translator knows far more about these roles than it does:

Interviewer: Steps 1a and 2a: are those the same kinds of statements?

Subject: How's that, are they the same <u>kind</u>. Ahhh, ummm, not exactly, because with this [1a] you are adding - you initialize it at zero and you're adding one to it [points to the right side of 1a], which is just a constant kind of thing.

Interviewer: Yes

Subject: [points to 2a] Sum, initialized to, uhh Sum to Sum plus Num, ahh - thats [points to left side of 2a] storing two values in one, two variables [points to Sum and Num on the right side of 2a]. Thats [now points to 1a] a counter, thats what keeps the whole loop under control. Whereas this thing [points to 2a], this was probably the most interesting thing ... about Pascal when I hit it. That you could have the same, you sorta have the same thing here [points to 1a], it was interesting that you could have, you could save space by having the Sum re-storing information on the left with two different things there [points to right side of 2a], so I didn't need to have two. No, they're different to me.

Interviewer: So -- in summary, how do you think of 1a ?

.Subject: I think of this [point to 1a] as just a constant, something that keeps the loop under control. And this [points to 2a] has something to do with something that you are gonna, that stores more kinds of information that you are going to take out of the loop with you.

This interview explains a result we have from an earlier written study. We found 100% of novices working on problems like 2 and 3 were able to correctly write the counter variable update statement ("`Count := Count + 1`"), while only 83% could correctly write the running-total variable update ("`Sum := Sum + Num`") [Soloway et al, 1982]. Why this difference with statements syntactically and semantically so similar? With this transcript, we now have some insight into the problem. Our subject seems to be keying on the role -- the *pragmatics* -- of the statements, noticing but not concentrating on the syntactic and semantic regularity. The running-total variable update is more difficult because it "stores information that you are going to take out of the loop with you". That is, it has implications outside the loop body.

4. Conclusions

The implication of these results is not simply to make syntactic fixes to programming languages. Instead, we are suggesting that the knowledge people bring from natural language has a key effect on their early programming efforts. Shneiderman and Mayer [1979] have proposed a model of programmer behavior based on language specific knowledge (which they call *syntactic*) and more general programming knowledge (called *semantic*). Our results suggest that there is a third body of *natural language step-by-step specification knowledge* which strongly influences novice programming behavior.

Miller [1981], Green [1981], and others have previously looked at step-by-step natural language specifications. They concentrated on looking at the suitability of natural language for directing computers. Based on the ambiguities and complexity limitations of natural language, they concluded it would be quite difficult to program in natural languages. Here, we are not contradicting that result, but extending it. We are finding that novice programmers *do* use natural language, even when they think they are using a programming language.

There are several implications of this work for programming education. First, we note that the power of the notions from structured programming will only be useful to students who have mastered the level of pragmatic and tacit programming knowledge highlighted in this paper. We need to address the problems students have very early in their programming education. The errors discussed here are barriers for many programming students. Only after a student has mastered writing a simple loop, for example, is he or she ready to see the power of a top-down design involving several loops.

We are beginning to explain many novice programming errors through the idea of natural language step-by-step specification plans. The quality of these explanations has proved important in the development of a tutor to do intelligent computer assisted instruction of programming [Soloway et al., 1981b]. In the future, we hope to extend the tutor to understand a stylized form of these natural language plans.

Though use of our plans cannot yet be fully automated, such plans can still play a part in a programming curriculum. As we stated earlier, the knowledge contained in such plans is usually tacit. Programming teachers, we feel, have much to gain by making that knowledge as explicit as possible as early as possible. We are presently developing an introductory course where students are taught both natural language step-by-step specification plans and programming plans from the beginning. Not only is the information in plans made explicit, but the differences between similar plans for different languages, in particular the natural language and the programming language being studied, can be made explicit. (These ideas are developed fully in Bonar [1983].)

Finally, what is the key to cognitively appropriate novice computing systems? Our work suggests that we need serious study of the knowledge novices bring to a computing system. For most computerized tasks there is some model that a novice will use in his or her first attempts. We need to understand when is it appropriate to appeal to this model, and how to move a novice to some more appropriate model.

5. References

Bonar, J., K. Ehrlich, E. Soloway, and E. Rubin, (1982) Collecting and Analyzing On-Line Protocols from Novice Programmers, in *Behavioral Research Methods and Instrumentation*, May 1982.

Bonar, J. (1983) Natural Problem Solving Strategies and Programming Language Constructs: Conflicts and Bridges. Ph.D thesis in preparation.

Collins, A. (1978) Explicating the Tacit Knowledge in Teaching and Learning, presented at the American Education Research Association (also Bolt Beranek and Newman Technical Report 3889).

DiSessa, A., (1982) Unlearning Aristotelian Physics: A Study of Knowledge-Based Learning, *Cognitive Science*, 6:1 (January-March), pp. 37-75.

Du Boulay, B. and T. O'Shea (1981) Teaching Novices Programming, in *Computing Skills and the User Interface* edited by M.J. Coombs and J.L. Alty, Academic Press, New York.

Green, T. (1981) Programming As a Cognitive Activity, in *Human Interaction With Computers*, edited by C. Smith and T. Green, Academic Press.

Miller, L. A. (1981) Natural language programming: Styles, strategies, and contrasts, *IBM Systems Journal*, 20:2, pp. 184-215.

Rissland, E. (1978) The Structure of Mathematical Knowledge. *Cognitive Science*, 2:4 (October-December 1978).

Shneiderman, B. and R. Mayer (1979) Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results, *International Journal of Computer and Information Science*, 8:3, pp. 219-238.

Soloway, E., J. Bonar, B. Woolf, P. Barth, E. Rubin, and K. Ehrlich (1981a) Cognition and Programming: Why Your Students Write Those Crazy Programs, appeared in Proceedings of the National Educational Computing Conference, pp. 206-219.

Soloway, E., B. Woolf, E. Rubin, J. Bonar, W. L. Johnson, (1983) MENO-II: An Intelligent Programming Tutor, *Journal of Computer-Based Instruction*, in press.

Soloway, E., K. Ehrlich, J. Bonar, J. Greenspan, (1982) What Do Novices Know About Programming?, in *Directions in Human-Computer Interactions*, edited by B. Shneiderman and A. Badre, Ablex Publishing Company.

Soloway, E., J. Bonar, and K. Ehrlich (1983) Cognitive Factors in Looping Constructs, *Communications of the ACM*, to appear.

Waters, R. C., (1979) A Method for Analyzing Loop Programs, *IEEE Transactions on Software Engineering*, SE-5:3, May.