Natural Problem Solving Strategies and Programming Language Constructs {1}

Jeffrey Bonar
Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

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. {2} What is it about this cognitive skill that is so difficult? Is it inherent in programming, or directly related to the nature of the programming tools currently used for novices? In this report we will present evidence that current programming languages do not accurately reflect human problem solving strategies developed in a context of step-by-step natural language specification. This evidence was gained by studying novice computer programs collected from their terminal sessions [Bonar et al, 1982], video-taped interviews of novices programming, and written studies focusing on specific aspects of novice programming techniques. {3}

Step-by-step natural language specification provides powerful intuitions for novice programmers using a programming language. We hypothesis that these intuitions take the form of frame-like plans - regular but flexible techniques for specifying how to accomplish a task. Programming knowledge also involves frame-like 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 an example of natural language and programming language plans. Using those plans we will discuss a transcripts of novice programmers using a natural language plans while attempting a programming language problem. We conclude with a brief discussion of the implications of this work.

{1} 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.

{2} Our own conservative estimate from several introductory programming courses is that more than 40% of the conscientious students never really understand the rudiments of programming.

{3} Du Boulay and O'Shea [1981] present an excellent overview of research into how novices learn programming.

2. A Mismatch Between a Natural Language Plan and a Program

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 a 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". To see why this is so, consider a problem analogous to Problem 1 but in a programming language domain:

> 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 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. Do novice programmers easily acquire such a plan? Apparently, no.

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

S: How do I get [the while loop] 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. ...

I: So how will the while statement behave?

S: 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 subjects "logical guess" is that the while behaves like a demon and not as a specific testing step among other steps. This is consistant with English phrases like "while you are on the highway, watch for the Northfield sign". Soloway et al [1981a] report that 34% of an introductory programming course had the "while demon" misconception.

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) <crossed out>
      end
   ...
```

and a transcript of the subject discussing this program:

S: 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 allows the "get a value, process that value" plan. In a separate study Soloway, et al [1981b] show that a new Pascal looping construct supporting this plan significantly improved novice and intermediate performance with Problem 2.

3. 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. 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, 1981c]. In the future, we hope to extend the tutor to understand a stylized form of these natural language plans.

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.

Acknowledgements - My deepest thanks to Elliot Soloway for his support and guidance. I would also like to thank John Clement for his critical comments.

4. References

Bonar, Jeffrey, Kate Ehrlich, Elliot Soloway, and Eric Rubin, (1982) "Collecting and Analyzing On-Line Protocols from Novice Programmers", in Behavioral Research Methods and Instrumentation, May 1982.

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

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

Shneiderman, Ben and Richard 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, Elliot, Jeffrey Bonar, Beverly Woolf, Paul Barth, Eric Rubin, and Kate Ehrlich (1981a) "Cognition and Programming: Why Your Students Write Those Crazy Programs", appeared in proceedings of the National Educational Computing Conference.

Soloway, Elliot, Jeffrey Bonar, and Kate Ehrlich (1981b) "Cognitive Factors in Looping Constructs". Computer and Information Science Technical Report 81-10, University of Massachusetts, Amherst, May.

Soloway, Elliot, Beverly Woolf, Eric Rubin, and Paul Barth (1981c) "Meno-II: An Intelligent Tutoring System for Novice Programmers", Proceedings of International Joint Conference in Artificial Intelligence, Vancouver, British Columbia.

Soloway, Elliot, Kate Ehrlich, Jeffrey Bonar, Judith Greenspan, (1982) "What Do Novices Know About Programming?", To appear in Directions in Human-Computer Interactions, edited by B. Shneiderman and A. Badre, Ablex Publishing Company.

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