ED 319 373 · IR 014 391

AUTHOR	Pea, Roy D.
TITLE	Language-Independent Conceptual "Bugs" in Novice Programming. Technical Report No. 31.
INSTITUTION	Bank Street Coll. of Education, New York, N.Y. Center for Children and Technology.
SPONS AGENCY	National Inst. of Education (ED), Washington, DC.; Spencer Foundation, Chicago, Ill.
PUB DATE	Dec 84
CONTRACT	400-830016
NOTE	14p.
PUB TYPE	Information Analyses (070) -- Viewpoints (120)
EDRS PRICE	MF01/PC01 Plus Postage.
DESCRIPTORS	*Computer Science Education; Elementary Secondary Education; *Error Patterns; Higher Education; *Misconceptions; *Programing

ABSTRACT

Persistent conceptual bugs exist in how novices, from primary school to college age, program and understand programs. These bugs are not specific to a given programming language but appear to be language-independent. The three different classes of bugs are: (1) parallelism, the assumption that different lines in a program can be active at the same time or in parallel; (2) intentionality, the attribution of goal directedness or foresightedness to the program; and (3) egocentrism, the assumption that there is more of the programmer's meaning for what he or she wants to accomplish in the program than is actually present in the code he or she has written. All of these bugs appear to derive from the idea that there is a hidden mind in the programming language that has intelligent, interpretive powers. Each of the three types of bugs is described and exemplified using student errors, and the implications for programming instruction are addressed. (23 references) (Author/MES)

Abstract

This paper argues for the existence of persistent conceptual "bugs" in how novices program and understand programs. These bugs are not specific to a given programming language, but appear to be language-independent. Furthermore, such bugs occur for novices from primary school to college age. Three different classes of bugs--parallelism, intentionality, and egocentrism--are identified, and exemplified through student errors. It is suggested that these classes of conceptual bugs are rooted in a "superbug," the default strategy that there is a hidden mind somewhere in the programming language that has intelligent interpretive powers.

Language-Independent Conceptual "Bugs" in Novice Programming

Roy D. Pea

Technical Report No. 31

December 1984

LANGUAGE-INDEPENDENT CONCEPTUAL "BUGS" IN NOVICE PROGRAMMING*,**

Roy D. Pea

Introduction

It is well known that students have such pervasive conceptual misunderstandings as novice programmers that correct programs early in the learning process come as pleasant surprises. Even after a year or more of programming instruction, students have great difficulty predicting what output a program will have, in what order commands will be executed, or in writing and debugging original programs to solve problems. Furthermore, these problems are not confined to the very young student in elementary school (Kurland & Pea, 1984; Pea, 1983) and junior high (Mawby, 1984), but appear to pervade the programming activities of high school, college (Bonar & Soloway, 1983; Soloway, Ehrlich, Bonar, & Greenspan, 1982), and mature adult students as well. What are the sources of these difficulties?

Many of these conceptual difficulties are confined to specific implementations of particular programming languages, and presumably can be remediated by redesigning the particular features of those implementations. In an exemplary study, Soloway, Bonar, and Ehrlich (1983) have shown how an invented while looping construct not available in Pascal was easier for novices to use in writing programs than the standard Pascal looping constructs. In this paper, however, I plan to consider instead the kinds of fundamental and widespread conceptual misunderstandings or "bugs" (Brown & Burton, 1978) in program understanding that appear, from our own and others' work, to be relatively independent of specific commands or programming languages. These misunderstandings, we will argue, have less to do

*The research discussed in this paper was supported by the Spencer Foundation and the National Institute Education (Contract No. 400-830016). I would like to thank my c[illegible]agues at the Center for Children and Technology for discussing these issues.

**Submitted to Journal of Educational Computing Research, special issue on "Novice Programming," edited by Elliot Soloway.

with the design of programming languages than with the problems people have in learning to give instructions to a computer.

Much of our programming instruction treats learning to program as a new and independent skill having little to do with previous learning: "It is almost as close to a situation of a tabula rasa as we are going to find in an adult" (Anderson, Farrell, & Sauers, 1984, p. 87; see, also, Pea & Kurland, 1984). Furthermore, in the classroom setting, students' errors are commonly considered to be idiosyncratic problems. But something much more interesting psychologically is happening, and we must come to understand it. It is not that students don't know anything that is relevant to programming--they have an intuitive understanding of much of what we say about programming. Depending on their age and developmental level, students have available experiences, and a broad range of concepts and strategies relevant to learning to program (Pea & Kurland, 1983). But one of the most central aspects of their intelligence is misleading when it comes to learning to program. The novice programmer works <u>intuitively</u> and pursues many blind alleys in learning the formal skill of programming. But what does it mean to work "intuitively"?

Specifically, students have a predominant metaphor that guides their behavior when, as novices, they write programming instructions to a computer. This metaphor is <u>conversing with a human</u>. Their strategies for using natural language with other humans leads them astray as they try to deal with programming, because programming is a formal system that interprets each part of a program (instructions to it) in terms of rules that are <u>mechanistic</u>. At least for the programming languages we will be referring to in our examples, there are strict rules for interpreting commands in a rigid sequential order, determined by how flow of control is dealt with in the language. While people are intelligent interpreters of conversations, computer programming languages are not. This fundamental feature of programming systematically violates the canons of human conversation. For example, a programming language cannot infer what a speaker means if she is not absolutely explicit. There are similar problems in the developmental transition from oral to written communication of natural language (Olson, 1977; Tannen, 1983), where the absence of the listener sets new constraints on the explicitness with which meaning must be expressed.

My aim here is to explicate a few of the major obstacles to programming expertise presented by three major classes of students' conceptual bugs in understanding. These errors are bugs in the sense that they are systematic--that is, not random errors or sloppy work--and that they need revision and further instruction for students to make progress in learning to program. I will close by suggesting some implications of these findings for how programming is taught.

Classes of Bugs

Parallelism Bugs

The parallelism bug is revealed in diverse contexts, but its essence is the assumption that different lines in a program can be active or somehow known by the computer at the same time, or in parallel. We can distinguish two different kinds of programs in which the parallelism misunderstanding is common.

One context in which the bug occurs is programs where conditional statements (IF...THEN) occur outside of loops. A common example is one where, early in a program, a conditional statement appears. SIZE will be our variable name in this case. The program says:

```
IF SIZE = 10, THEN PRINT "HELLO
```

Later in the program, a countup loop is encountered, where a variable is incremented by 1 each time until it reaches 10.

```
FOR SIZE = 1 to 10, PRINT "SIZE
NEXT SIZE
```

Now we may ask: What do students think the computer will do? If they understand the control structure of the programming language (in this case, BASIC), they know that the IF statement is first evaluated for its truth. If SIZE is equal to 10, HELLO is printed, and control passes to the next statement. If the VARIABLE is not equal to 10, nothing is printed, and control passes to the next statement. The knowledgeable programmer knows that after the first line of the program--the IF line--is executed, it is inactive, and irrelevant to whatever the rest of the program instructions say because the control cycle never returns to it.

But a recurrent problem for students--in this case, high schoolers in their second year of computer science--to whom we have offered problems of this type is that a very different prediction is offered for what will happen. In one study, 8 out of the 15 students interviewed predicted that during the looping process, when the variable SIZE became equal to 10, HELLO would be printed. When asked to explain why, the student observed that, since variable SIZE was now equal to 10 (i.e., within the loop) and the IF statement was "waiting for" the SIZE to be equal to 10, it could now print HELLO. But in fact, once the IF statement was evaluated and found false, it was never returned to in the program. There is a sense in which these students believe that all the lines in the program are active or alive at once. As one junior high student pronounced: "It looks at the program all at once because it is so fast." The program is thought to have an intelligence under the surface that monitors the action status of every line in the program simultaneously.

Now think about the logic of IF statements in natural conversation (McCawley, 1981). When I say to you, "If you want to go to the store, I'll drive you," there is a duration to my IF statement. It may not be active for a week, or even all day, but your response does not have to be immediate. If in an hour you want to go to the store, I am still likely to drive you there. The idea of an IF statement being evaluated and then taken off the books, as it were, is odd from a natural language perspective. So the student has applied her intuitions about how IF statements function in natural language discourse to the initially mysterious domain of computer language discourse.

A related finding (Soloway, Bonar, Barth, Rubin, & Woolf, 1981; Bonar & Soloway, 1983) involved novice Pascal programmers. A "while demon" bug was revealed when as many as a third of the college students assumed for simple Pascal programs that the actions in the while loop were continuously monitored for the exit condition to become true. For example, one student explained that "every time I [the variable tested in the while condition] is assigned a new value, the machine needs to check that value." The authors note that this interpretation is consistent with English while, as in "while the highway is two lanes, continue north."

The generality of the phenomenon may be observed in a second example of the parallelism bug revealed by students attempting to comprehend programs not involving conditionals--in this case, variable assignment statements which occur in a program after lines referring to that variable. The student thinks (incorrectly) that what will happen later in a program influences what happens earlier. For example, consider the following four-line program:

```
AREA = Height X Width
Input Height
Input Width
PRINT "AREA
```

Many students assume that there is no problem with this program, and predict that it will print out the product of the height and width values the program user has input. But this is not true. When the first statement is executed, that is, the one that defines AREA as height times width, it has not yet received the input values. So it treats height and width as equal to the default value of 0. What is printed is not, as the student assumes, the product of the input values of height and width, but the product of the values of those variables available at the time the first line in the program was executed, that is, 0 X 0 = 0.

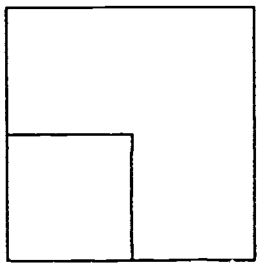
Here, once again, we can see the influence of natural language conversational strategies, where implicit knowledge or expectations of what will come later can guide the interpretation of what occurs early in a conversation (or text). In natural language, apart from procedural instructions such as recipes or building plans, there is no reason not to skip ahead. But in computer programming, the novice student must think to herself: "What conditions regarding inputs are in effect as this line is executed?" In natural language, one does not violate the meaning of a text by reading parts of it out of order, and in fact we even teach scanning ahead for structure as a reading strategy.

Intentionality Bugs

There is another class of important language-independent conceptual bugs that we will call Intentionality Bugs. Intentionality Bugs are those in which the student attributes goal directedness or foresightedness to the program and, in so doing, "goes beyond the information given" in the lines of programming code being executed when the program is run. Students adopt what Dennett (1978) calls an "intentional stance" toward the complex system represented by the programming language, and assume that it has capacities or attributes of a human.

In one example which we have studied in detail (Kurland & Pea, 1984), we ask students to talk out loud as they draw on graph paper what the graphics pen will draw as the following tail-recursive Logo program is executed. As depicted in the figure below, when one types SHAPE 40, the program draws a large square, a medium-sized

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



square inside it, and then stops. More specifically, the program draws a square with a variable side that, when initialized on the first call, is 40 units long. The first line of the program is a conditional counter with the purpose of stopping the drawing after the two squares are drawn. When executed, the next line draws a square the length of the variable SIDE (i.e., 40): REPEAT 4 [FORWARD :SIDE RIGHT 90]. The last line of the recursive program divides the variable SIDE by 2, and 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 stops, because the variable SIDE then equals 10.

When encountering the second line of the program, a conditional that says IF the value of the variable SIDE equals 10 STOP, some students erroneously predict that when the program is run, a box of side 10 will be drawn. When asked why, their comments are revealing. The students have glanced ahead in the program to see what is to them a familiar programming schema or "plan" (Soloway & Ehrlich, 1984)--a command line that results in the drawing of a square: REPEAT 4 [FORWARD (SOME DISTANCE) RIGHTANGLETURN (90 DEGREES)]. They then read the IF statement as a command to draw a square with sides equal to 10, because "it will draw a square," or "because it wants to draw a square." Other students recognize that the variable at the IF statement equals 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 parallelism and intentionality bugs--the program has been given the status of an intentional being which has goals, and knows or sees what will happen elsewhere in itself.

Egocentrism Bugs

Egocentrism bugs are the flip side of intentionality bugs. Whereas intentionality bugs involve comprehending and tracing what a program will do, egocentrism bugs are involved in creating a program to do something. Each bug type presupposes that the computer can do what it has not been told to do in the program.

Egocentrism, an overemphasis on the perspective of self relative to that of others, is a pervasive characteristic of novice thinking, manifested in spatial cognition (Piaget & Inhelder, 1967), communication (Flavell et al., 1968), and other problem domains. It should thus come as no surprise that the task performances of novice programmers are also subject to egocentric biases. Egocentrism bugs are those where students assume that there is more of their meaning for what they want to accomplish in the program than is actually present in the code they have written. Students giving evidence of this bug egocentrically assume that the computer can follow the advice former Mayor of Chicago Richard Daley used to give reporters:

Don't print what I say, print what I mean!

For example, lines of code or variable values are omitted by these students because it is assumed that the computer "knows" or can "fill in," as a human listener can, what the student wishes it to do.

Students do not literally say that the program knows what to do; the errors generated by this bug are almost perceptual in nature--their current conceptions do not guide their attention to these problems as relevant reasons for their programs' not working as planned. When asked to explain what their programs will do, they gloss over the specific commands in a line of Logo code, asserting that a line of graphics code draws a square when, for example, they have included a move command to send the graphic 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 to the computer have been omitted. All they have provided is the skeleton of a program, assuming that in some way the computer can fill in the rest, can say what they "mean."

Bonar and Soloway (1983) provide another clear case of egocentrism, manifested by a college student writing a program in Pascal. The student was writing pseudo-code for the problem: "Write a program which reads in 10 integers and prints the average of those integers." She wrote out:

```
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)
```

When the interviewer asked whether (1a) was the "same kind of statement" as (2a), it became clear "that she thinks the Pascal translator knows far more about these roles than it does":

> "Are they the same kind. Ahhh, ummm, not exactly, because with this [1a] you are adding--you initialize it as zero and you're adding one to it [points to the right side of 1a], which is just a constant kind of thing. [Points to 2a] Sum, initialized to, uhh, Sum to Sum plus Num, ahh--

that's [points to left side of 2a] storing two values in one, two variables [points to Sum and Num on the right side of 2a]. That's [now points to 1a] a counter, that's 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. 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." (p. 5)

Here, again, we see the student believing that the programming language knows more about her intentions than it possibly can.

Soloway et al. (1982) have found among college Pascal programmers a set of errors that we believe also stems from egocentrism bugs. They describe what they call a "mushed variables" bug. After a semester of Pascal, more than one quarter of their novice programmers used the same variable incorrectly for more than one role. For example, in the following program, the variable X is used both to store a value being read in [read (X)] and to hold a running total [X := X+X]:

```
program Student26_Problem2;
   var X, Ave : integer
   begin
   repeat
      Read (X)
      X := X + X
   until X + X [greater-than sign] 100;
   Ave := X div Nx;
   Write (Ave)
   end.
```

They observe that students making these errors may have assumed that the computer would recognize that the same variable played two different roles, and that it could use the different values appropriately.

Conclusions

All the bugs discussed--parallelism, intentionality, and egocentrism--appear to derive from what might be called a superbug. The superbug may be described as the idea that there is a hidden mind somewhere in the programming language that has intelligent, interpretive powers. It knows what has happened or will happen in lines of the program other than the line being executed; it can benevolently go beyond the information given to help the student achieve her goals in writing the program. This "hidden mind superbug" interpretation provides a deep explanation of the various misconceptions that plague the novice programmer.

But there is too facile an interpretation of this argument that must be avoided because it is false. It is not that students literally believe that the computer has a mind, or can think, or can interpret what was not explicitly stated. In our experience, novice programming students are likely to vehemently deny that the computer can think or that it is intelligent. Besides, instructors are very good at highlighting this point at the beginning of courses: Computers are dumb and can do nothing but what you tell them! But students' behaviors when working with programs often contradict their denials; they act as if the programming language is more than mechanistic. Their default strategy for making sense when encountering difficulties of program interpretation or when writing programs is to resort to the powerful metaphor of natural language conversation, to assume a disambiguating mind which can understand. This personal metaphor should be seen as expected rather than bizarre behavior, for the students have no other analog, no other procedural device than "person" to which they can give written 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 this case, mapping conventions for natural language instructions onto programming results in error-ridden performances.

What are the implications of these findings for programming instruction? First, we need to be aware of the pervasiveness of programming misunderstandings that arise from tacit applications of human conversational metaphor to programming. This is powerful transfer, to be sure, but it is misleading and does not work. Second, beyond being aware of these bugs, we have to arrange many more kinds of learning activities for students, and diagnostic activities for teachers, in which the bugs can be made obvious. We believe the persistence of these bugs is in part linked to the infrequency with which they are explicitly confronted by students and teachers alike. Bugs like these could be snared if one used program reading or debugging activities as central components of programming instruction. It was not until we did the tedious work of having students walk through every command in a program, thinking aloud and explaining how the computer would interpret it, that we became aware of the prevalence of these bugs. After that, we saw them everywhere.

Much more research is needed on how best to help students see that computers read programs through a strictly mechanistic and interpretive process, whose rules are fairly simple once understood. We think this can best be achieved by providing clear models that show how the processing of control and data is regulated by the specific programming language under study, and by explicit modelling and instruction in comprehension-monitoring processes for computer programs similar to those that have been effective for written language understanding (Palincsar & Brown, 1984). Currently, our own studies are addressing these problems. Other useful leads will come from artificial-intelligence, knowledge-based programmers' assistants (Waters, 1982) and debugging aides that seek to identify and remediate students' pervasive misconceptions in learning how to program (e.g., Johnson & Soloway, 1984).

Finally, we can be assured of (although not comforted by) the fact that such conceptual difficulties are not specific to the programming domain. There are other formal systems with abstract rules of interpretation--logic, physics, and mathematics--that are also very challenging for students to learn, rife with bugs (e.g., Gentner & Stevens, 1983), but well worth our concerted efforts to help students understand.

References

Anderson, J. R., Farrell, R., & Sauers, R. (1984). Learning to program in LISP. Cognitive Science, 8, 87-129.

Bonar, J., & Soloway, E. (1983). Uncovering principles of novice programming. In SIGPLAN-SIGACT Tenth Annual Symposium on Principles of Programming Languages, Austin, TX.

Brown, J. S., & Burton, R. (1978). Diagnostic models for procedural bugs in basic mathematical skills. Cognitive Science, 2, 155-192.

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

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.

Gertner, D., & Stevens, A. (1983). (Ed.). Mental models. Hillsdale, NJ: Erlbaum.

Johnson, W. L., & Soloway, E. (1984). PROUST: Knowledge-based program understanding (Tech. Rep. No. 285). New Haven, CT: Yale University, Department of Computer Science.

Kurland, D. M., & Pea, R. D. (in press). Children's mental models of recursive Logo programs. Journal of Educational Computing Research, 2.

Mawby, R. (1984, August). Proficiency conditions for the development of thinkings skills through programming. Paper presented at the Harvard University Conference on Thinking, Cambridge, MA.

McCawley, J. D. (1981). Everything that linguists have always wanted to know about logic (but were ashamed to ask). Chicago: University of Chicago Press.

Olson, D. R. (1977). From utterance to text: The bias of language in speech and writing. Harvard Educational Review, 47, 257-281.

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). Logo programming and problem solving (Tech. Rep. No. 12). New York: Bank Street College of Education, Center for Children & Technology.

Pea, R. D., & Kurland, D. M. (1984). On the cognitive effects of learning computer programming. New Ideas in Psychology, 2, 131-168.

Pea, R. D., & Kurland, D. M. (1983). On the cognitive prerequisites of learning computer programming. Project Report to the National Institute of Education. (Also Technical Report No. 18, Bank Street College of Education, Center for Children & Technology)

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

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

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

Soloway, E., Bonar, J., & Ehrlich, K. (1983, November). Cognitive strategies and looping constructs: An empirical study. Communications of the ACM.

Soloway, E., & Ehrlich, K. (in press). Empirical studies of programming knowledge. IEEE Transactions on Software Engineering.

Soloway, E., Ehrlich, K., Bonar, J., & Greenspan, J. (1982). What do novices know about programming? In B. Shneiderman & A. Badre (Eds.), Directions in human-computer interactions. Norwood, NJ: Ablex.

Tannen, D. (1983). (Ed.). Coherence in spoken and written discourse. Norwood, NJ: Ablex.

Waters, R. A. (1982). A knowledge-based program editor. Proceedings of the 7th International Joint Conference on Artificial Intelligence (Vol. II), pp. 920-926.