J. EDUCATIONAL COMPUTING RESEARCH, Vol. 2(1), 1986

LANGUAGE-INDEPENDENT CONCEPTUAL "BUGS" IN NOVICE PROGRAMMING*

ROY D. PEA

ABSTRACT

This article 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.*

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 [1-5] and junior high [6], but appear to pervade the programming activities of high school, college [7, 8], 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 ameliorated by redesigning the particular features of those implementations, or by means of automatic error finders. In an exemplary study, Soloway, Bonar, and Ehrlich 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

* The research discussed in this article was supported by the Spencer Foundation and the National Institute of Education (Contract No. 400-830016).

constructs [9]. In this article, however, I plan to consider instead the kinds of fundamental and widespread conceptual misunderstandings or "bugs" [10] 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 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" [11, 12]. 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 [13]. But one of the most central aspects of their intelligence is misleading when it comes to learning to program. The novice programmer works *intuitively* 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 analogy that guides their behavior when, as novices, they write programming instructions to a computer. This analogy is *conversing with a human*. Their pragmatic strategies for using natural language with other humans lead 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 *mechanistic*. 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 human conversational maxims, such as the cooperative principles outlined by Grice [14], and developed in theories of natural language pragmatics (e.g., Cole [15]; Searle [16]). For example, a programming language cannot infer what a speaker means if she is not absolutely explicit, whereas a listener in a human-human conversation can query the speaker for clarification. There are similar problems in the developmental transition from oral to written communication of natural language [17, 18], 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. This division is offered as a first attempt at defining a taxonomy for guiding discussion of these problems. 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. The data on which these arguments are based consisted of several years of Logo programming studies, with eight- to twelve-year-olds, and fourteen- to seventeen-year-olds, and observations of high school students learning Basic programming. Details follow where appropriate. 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. Though there may be others, 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 one each time until it reaches ten.

```
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 ten, HELLO is printed, and control passes to the next statement. If the *variable* is not equal to ten, 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, eight out of the fifteen students interviewed predicted that during the looping process, when the variable SIZE became equal to ten, HELLO *would* be printed. When asked to explain why, the student observed that, since variable SIZE was *now* equal to ten (i.e., within the loop) and the IF statement was "waiting for" the SIZE to be equal to ten, 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 [19]. When I say to you, "If you want to go to the store, I'll drive you," there is more than an instantaneous 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 temporal period will vary according to context in ways currently little understood.) 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 the duration of IF statements in natural language discourse to the initially mysterious domain of computer language discourse. It is possible that a different notation for if . . . then (e.g., condition-action pairs) could attenuate this interpretive problem.

A related finding involved notive Pascal programmers [7, 20]. 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 (which would essentially be true were it written in Prolog, in which the interpreter does do inference!), 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 zero. 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 often no reason not to skip ahead. But in computer programming, the novice student must think: "What conditions regarding inputs are in effect as this line is executed?" In natural language, one rarely violates the meaning of a text by reading parts of it out of order, since line-by-line comprehension is not essential. In fact, we even teach scanning ahead for structure as a reading strategy. Nonetheless, natural language out-of-order reading does often disrupt text comprehension, as research on story grammars reveals.

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 calls an "intentional stance" toward the complex system represented by the programming language, and assume that it has capacities or attributes of a human [21].

In one example which we have studied in detail [12], 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 forty 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 two, 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 forty and twenty 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" [22]—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 if the program is *commanding* the computer to draw a square with sides equal to ten, 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 forty, but then say that the program sees the box statement line ahead which it wants to draw, but has to *stop* at ten!

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 children's thinking, manifested in spatial cognition [23], communication [24], and other problem domains. Under the strenuous cognitive demands of a new task environment, it may also surface as a characteristic of the performances of novice programmers who are adolescents and adults. 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.[1]

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. A common problem of this kind is the omission of punctuation or control characters, and the nonprovision of values for variables. Lest these omissions be thought of only as careless work, one can probe the students to test our current hypothesis, which attributes more significance to these omissions than oversight. When asked to explain what programs they have written will do, they gloss over the specific commands in a line of Logo code just written, 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 provide another clear case of egocentrism, manifested by a college student writing a program in Pascal [7]. The student was writing pseudo-code for the problem: "Write a program which reads in ten 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":

[1] Several counterexamples and, perhaps, part of a growing trend, are Teitelbaum's DWIM (Do What I Mean) systems added to the Interlisp programming environment, which corrects spelling errors by using syntactic context, and commercially available syntax-correcting compilers. Such painless error revisions are the subject of feverous debates among programmers.

> 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 [points 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 [7, 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.* have found among college Pascal programmers a set of errors that we believe also stems from egocentrism bugs [8]. 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 analogy of natural language conversation, to assume a disambiguating mind which can understand. It is not clear at the current time whether this strategy is consciously pursued by students, or whether it is a tacit overgeneralization of conversational principles to computer programming "discourse." The central point is that this personal analogy 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 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 [25]. But, in this case, mapping conventions for natural language instructions onto programming results in error-ridden performances.

A rival explanation for the aforementioned classes of bugs is that the novice programmer does *not* impute interpretive intelligence to the machine. It is not that the programmer assumes that a distinction needs to be expressed and that the computer can make that distinction. Instead, he or she simply does not understand that there are ambiguities to be resolved in the code that has been written. From this perspective, the common developmental problem of coming to distinguish alternatives which are initially fused or collapsed in thought is viewed as the source of the kinds of errors we have discussed. While this possibility should be considered for some error-ridden programs, there are types of errors, such as the parallelism bugs, which are unlikely to result from such conceptual fusion. And it is difficult to see on this rival interpretation why we find that novice programmers often utilize intentional terms to describe the process by which the computer executes the commands presented by the program.

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 the 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.

There are additional complexities to be faced from a pedagogical perspective. From the programmer's viewpoint, it is not true that *every* operation to be carried out has to be made explicit. There are many things which programming languages automatically carry out, without, as it were, specific instructions to do so (e.g., physical address management; stack storage allocation; Pascal compiler disambiguation of the meaning of the semicolon from context). So the lesson the novice programmer needs to learn is that some meanings do not need to be explicitly expressed in the code he or she writes, while others do. Since the boundaries of required explicitness are conventions that vary across programming languages, the learner must realize the necessity of identifying in exactly what ways the language he or she is learning "invisibly" specifies the meaning of code written.

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. These explanations can be supported by explicit think-aloud examples of how the facile programmer thinks about and makes decisions with respect to program creation and understanding, and through instruction in comprehension-monitoring processes for computer programs similar to those that have been effective for written language understanding [26]. Other useful leads will come from artificial-intelligence, knowledge-based programmers' assistants [27], and debugging aides that seek to identify and remediate students' pervasive misconceptions in learning how to program [28].

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 [29], but well worth our concerted efforts to help students understand.

ACKNOWLEDGMENT

I would like to thank my colleagues at the Center for Children and Technology for discussing these issues, and I hereby express my appreciation for the constructive comments of several anonymous reviewers.

REFERENCES

1. D. M. Kurland and R. D. Pea, Children's Mental Models of Recursive Logo Programs, *Journal of Educational Computing Research*, *2*, in press.
2. U. Leron, Some Problems in Children's Logo Learning, in *Proceedings of the Seventh International Conference for the Psychology of Mathematics Education*, Weizmann Institute, Jerusalem, 1983.
3. J. D. Milojkovic, "Children Learning Computer Programming: Cognitive and Motivational Consequences," doctoral dissertation, Department of Psychology, Stanford University, 1983.
4. R. Nachmias, D. Mioduser, and D. Chen, *Acquisition of Basic Computer Programming Concepts by Children*, (Technical Report Number 14), The Computers in Education Research Lab, School of Education, Tel Aviv University, Tel Aviv, 1985.
5. R. D. Pea, *Logo Programming and Problem Solving*, (Technical Report Number 12), Bank Street College of Education, Center for Children and Technology, New York, 1983.
6. R. Mawby, Proficiency Conditions for the Development of Thinking Skills Through Programming, paper presented at the Harvard University Conference on Thinking, Cambridge, Massachusetts, 1984.
7. J. Bonar and E. Soloway, Uncovering Principles of Novice Programming, in *SIGPLAN-SIGACT*, tenth annual symposium on Principles of Programming Languages, Austin, Texas, 1983.
8. E. Soloway, K. Ehrlich, J. Bonar, and J. Greenspan, What Do Novices Know about Programming? in *Directions in Human-Computer Interactions*, B. Shneiderman and A. Badre (eds.), Ablex, Norwood, New Jersey, 1982.
9. E. Soloway, J. Bonar, and K. Ehrlich, Cognitive Strategies and Looping Constructs: An Empirical Study, *Communications of the ACM*, November 1983.
10. J. S. Brown and R. Burton, Diagnostic Models for Procedural Bugs in Basic Mathematical Skills, *Cognitive Science*, *2*, pp. 155-192, 1978.
11. J. R. Anderson, R. Farrell, and R. Sauers, Learning to Program in LISP, *Cognitive Science*, *8*, pp. 87-129, 1984.
12. R. D. Pea and D. M. Kurland, On the Cognitive Effects of Learning Computer Programming, *New Ideas in Psychology*, *2*, pp. 131-168, 1984.
13. R. D. Pea and D. M. Kurland, *On the Cognitive Prerequisites of Learning Computer Programming*, project report to the National Institute of Education. (Also Technical Report Number 18, Bank Street College of Education, Center for Children and Technology), New York, 1983.
14. H. P. Grice, Logic and Conversation, in *Syntax and Semantics 3: Speech Acts*, P. Cole and J. Morgan (eds.), Academic Press, New York, 1973.

15. P. Cole, *Radical Pragmatics*, Academic Press, New York, 1981.
16. J. R. Searle, *Intentionality*, Cambridge University Press, Cambridge, 1983.
17. D. R. Olson, From Utterance to Text: The Bias of Language in Speech and Writing, *Harvard Educational Review*, *47*, pp. 257-281, 1977.
18. D. Tannen (ed.), *Coherence in Spoken and Written Discourse*, Ablex, Norwood, New Jersey, 1983.
19. J. D. McCawley, *Everything that Linguists Have Always Wanted to Know about Logic (But Were Ashamed to Ask)*, University of Chicago Press, Chicago, 1981.
20. E. Soloway, J. Bonar, J. Barth, E. Rubin, and B. Woolf, Programming and Cognition: Why Your Students Write Those Crazy Programs, *Proceedings of the National Educational Computing Conference*, pp. 206-219, 1981.
21. D. Dennett, *Brainstorms*, Bradford Books, Montgomery, Vermont, 1978.
22. E. Soloway and K. Ehrlich, Empirical Studies of Programming Knowledge, *IEEE Transactions on Software Engineering*, in press.
23. J. Piaget and B. Inhelder, *The Child's Conception of Space*, Norton, New York, 1967.
24. J. H. Flavell, P. T. Botkin, C. L. Fry, J. W. Wright, and P. E. Jarvis, *The Development of Role-taking and Communication Skills in Children*, Wiley, New York, 1968.
25. D. E. Rumelhart and D. A. Norman, Analogical Processes in Learning, in *Cognitive Skills and Their Acquisition*, J. R. Anderson (ed.), Erlbaum, Hillsdale, New Jersey, 1981.
26. A. S. Palincsar and A. L. Brown, Reciprocal Teaching of Comprehension-fostering and Comprehension-monitoring Activities, *Cognition and Instruction*, *1*, pp. 117-175, 1984.
27. R. A. Waters, A Knowledge-based Program Editor, *Proceedings of the 7th International Joint Conference on Artificial Intelligence* Vol. II, pp. 920-926, 1982.
28. W. L. Johnson and E. Soloway, *PROUST: Knowledge-based Program Understanding*, (Technical Report Number 285), Department of Computer Science, Yale University, New Haven, Connecticut, 1984.
29. D. Gentner and A. Stevens (eds.), *Mental Models*, Erlbaum, Hillsdale, New Jersey, 1983.

Direct reprint requests to:

Dr. Roy D. Pea
Bank Street ~~School~~ College
610 W. 112th Street
New York, NY 10025