Children's mental models of recursive logo programs

D. Midian Kurland, Roy D. Pea

J. EDUCATIONAL COMPUTING RESEARCH, Vol. 1(2), 1985

CHILDREN'S MENTAL MODELS OF RECURSIVE LOGO PROGRAMS*

D. MIDIAN KURLAND
ROY D. PEA
Center for Children and Technology
Bank Street College of Education, New York

ABSTRACT

Children who had a year of Logo programming experience were asked to think-aloud about what brief Logo recursive programs will do, and then to predict with a hand-simulation of the programs what the Logo graphics turtle will draw when the program is executed. If discrepancies arose in this last phase, children were asked to explain them. A prevalent but misguided "looping" interpretation of Logo recursion was identified, and this robust mental model persisted even in the face of contradiction between what the program did when executed and the child's predictions for what it would do.

The power and elegance of recursion as a development in the history of programming languages (such as LISP, the lingua franca of artificial intelligence, and Logo) and its conceptual importance in mathematics, music, art and cognition generally is widely acknowleged [1]. Far less attention has been given, however, to the fundamental *developmental* problem of how people learn to use the powers of recursive thought and recursive programming procedures. Our approach to this research question has been influenced by several findings basic to a developmental cognitive science, specifically, the role of mental models in guiding learning and problem solving, and the widespread use of systematic, rule-guided approaches to problem solution by children, not only adults [2]. Understanding recursive functions in programming involves (programming language) notational and conceptual problems, the latter including problems with understanding flow of control. In programming, the novice is guided by a mental model of how program code controls the computer's operations. For novices, this model is adapted

* This work was supported by a grant from the Spencer Foundation.

over time as the result of both direct instruction and feedback from their own programming and debugging experiences, in which conflicts between their current theory and the behavior of the program is reflected upon.

A widespread belief among computer educators of precollege age populations is that young children can "discover" many of the powerful ideas formally present in programming simply through experimenting within a rich programming environment, as if unconstrained by prior understandings. This belief has been largely due to Papert's popularization of Logo [3], a LISP-like language designed for use by children to allow them to develop powerful ideas, such as recursion, in "mind sized bites." Papert and others assume that children can learn recursion through self-guided explorations of programming concepts in the Logo language. However, our observations of eight- to twelve-year-olds who have had a year of experience programming in Logo indicates that most avoid all but simple iterative programs, which do not require the deep understanding of control structure prerequisite for an understanding of recursion.

In a study examining children's ability to develop recursive descriptions of problems, Anzai and Uesato have shown how adolescents' understandings of recursive formulations of the factorial function is facilitated by a prior understanding of iteration [4]. They demonstrate that for mathematics, recursion can be learned via a discovery process by most children, particularly if they have first experimented with iterative functions. Of their subjects who correctly identified the iterative structure in a set of problems, 64 percent were also able to work out recursive solutions to a second set of problems. However, only 33 percent of the subjects who did not have prior experience with iteration were able to work out the recursive functions. Anzai and Uesato conclude that understanding recursion is aided by an understanding of iteration, but that "we should be cautious when we try to extend the consideration to more complex domains such as computer programming . . . [since] a complex task necessarily involves many different cognitive subprocesses, and it is not always easy to extract from them only the part played by recursion." [4, p. 102]

While Anzai and Uesato focus on the insight necessary to generate a recursive description of a math function, in programming one must acquire that insight *and* be able to implement it in specific programming formalisms [4]. In addition to an understanding of recursion, the child requires an understanding of the logic and terminology governing the control structure of the language. Adult novices have trouble with both. Learning to program they have great difficulties in thinking through flow of control concepts such as Pascal's *while* loop construction [5], and tail recursion in SOLO, a Logo-like language [6], even following extensive instruction. Furthermore, Bonar has found that prior natural language understandings of programming terms misleads novice programmers in their attempts at explaining how a program works [7]. Prior meaning is brought to the task of constructing meaning from lines of programming code. We expect children will also be guided in their interpretation of programming language

constructs by their natural language meanings, and by faulty mental models of flow of control structure. Indeed, a widespread experience among programming instructors is that novices have great trouble acquiring the concept of recursion and the ability to use recursive formalisms in their programs.

HOW RECURSION WORKS IN LOGO: A USER'S PERSPECTIVE

In this study, children worked with recursive programs composed of procedures written in Logo. When a Logo program is run, if a procedure references itself, execution of that procedure is temporarily suspended, and control is passed to a *copy* of the named procedure. Passing of control is *active* in the sense that the programmer is explicitly directing the program to execute a specific procedure. However, when the execution of this instantiation of the procedure is finished, control automatically is passed back to the suspended procedure, and execution resumes at the point where it left off. Passing of control in this case is *passive* since the programmer did not need to specify where control should be passed in the program.

To understand how recursive procedures work in Logo one must know:

1. The rule that execution in Logo programs proceeds line by line. However, when a procedure calls another procedure *or* itself, this acts to insert all lines of the named procedure into the executing program at the point where the call occurred. Control then proceeds through each of these new lines before carrying on with the remaining lines of the program. Thus control is *passed forward* to the called procedure, and then is *passed back* to the calling procedure.
2. That when a procedure is executed, if there are no further calls to other procedures or to itself, execution proceeds line by line to the end of the procedure. The last command of all procedures is the END command. END signifies that execution of the current procedure has been completed *and* that control is now passed back to the procedure from which the current one was called. END thus 1) signals the completion of the execution of one logical unit in the program, and 2) directs the flow of control back to the calling procedure so the program can carry on.
3. That there are several exceptions to the line by line execution rule. An important one for recursion is the STOP command. STOP causes the execution of the current procedure to be halted and control to be passed back to the procedure from which the currently executing one was called. Functionally, then, STOP means to branch immediately to the nearest END statement.

How well novice programmers' mental models of the workings of recursive procedures took into account these three central points was our research focus.

CENTRAL POINTS

Subjects

Seven children (two girls and five boys, eleven- to twelve-years-old) in their second year of Logo programming participated in the study. The children were highly motivated to learn Logo programming, and had averaged over fifty hours of classroom programming time under the supervision of experienced classroom teachers knowledgeable in the Logo language, who followed the "discovery" logo pedagogy set out by Papert [3]. All seven children had received instruction in iteration and recursion, and had demonstrated in their classroom programming that they could use iteration and recursion in some contexts.

Materials

Short Logo programs were constructed of procedures which reflected four distinct levels of complexity: 1) procedures involving only direct commands to move the turtle; 2) procedures using the iterative REPEAT command; 3) tail recursive procedures; and 4) embedded recursion procedures. This article focuses on the revealing features of children's performance at levels 3 and 4. Examples of programs at levels 3 and 4 are:

Level 3: tail recursion program (:SIDE = 80)

```
TO SHAPEB :SIDE
   IF :SIDE = 20 STOP
   REPEAT 4 [FORWARD :SIDE RIGHT 90]
   RIGHT 90 FORWARD :SIDE LEFT 90
   SHAPEB :SIDE/2
END
```

Level 4: embedded recursion program (:SIDE = 80)

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

Experimental Procedures

Our choice of a method was guided by comprehension studies which utilize "runnable mental models" or simulations of operations of world beliefs in response to specific program inputs [8]. Children were asked to give a verbal

account of how a Logo procedure would work, then to hand simulate the running of the program line by line by using a graphic turtle "pen" on paper. Then they were shown the consequences of running the program they had just explained, and if their simulation mismatched the turtle's actions during drawing, they were asked to explain the discrepancies, and one additional problem at that level was presented, with the same procedure.

RESULTS

All seven children made accurate predictions for programs at the first two complexity levels with only minor difficulties. They expressed no problems with the recursive call of the tail recursive programs of level 3; however, two children treated the IF statement as an *action* command to the turtle, and another assumed that since *she* did not understand the IF statement the computer would ignore it. No child made accurate predictions for either embedded recursion program at level 4. The children's problems with explaining embedded recursion may be traced to two related sources. The first involves general bugs in their mental model for how lines of programming code dictate the computer's operations when the program is executed, while the second concerns the particular control structure of embedded recursive procedures.

General Bugs in Program Interpretation

Decontextualized interpretation of commands – Children carried out "surface readings" of programs during their simulations. They attempted to understand each line of programming code individually, ignoring the context provided by previous program lines. They stated the definition of each command, rather than treating program lines as parts of a functional structure in which the purpose of particular lines is context-sensitive and sequence-dependent. This caused particular trouble during their simulations in keeping track of the current value of the variable SIDE, and in determining the actual order in which lines of code would be executed. Understanding recursion is impossible without this knowledge about sequential execution. The child must learn to ask: "How does the line I'm reading relate to what has already happened and affect the lines to follow?" The two bugs which follow concern an opposite tendency, an overrich search for meaning in other program lines.

Assignment of intentionality to program code – The children often did not differentiate the meaning of a command line they were simulating from the meaning of lines of commands they *expected* to follow (e.g., lines that if executed would draw a BOX). For example, in the program SHAPEC, one child came to the IF statement and said: "If :SIDE equals 100 stop. O.K., I think this will make a box that has a hundred side." Another child at the same point in that procedure, simply said: "this makes it draw a square."

Treating programs as conversation-like – As in understanding conversation, and in the problems non-schooled people encounter in formal reasoning (where beliefs about the truth of an argument's premises are focused on rather than the validity of its form [9, 10], children appropriate for problem solving any knowledge they believe will help them understand. In the case of Logo program comprehension, this empirical strategy has the consequence of "going beyond the information given" to comprehend the meaning of lines of code, such as deriving implications from one line of code (e.g., an IF statement) about the meaning of another line. For example, one child interpreted the recursive statement in SHAPEC as having the intention of drawing a square, predicting that the turtle would immediately draw a square before proceeding to the next command.

Overgeneralization of natural language semantics – Children interpreted the Logo commands END and STOP by analogy to their natural language meanings, which led them to believe that when they appear the program comes to a complete halt. Several children thus concluded that procedure SHAPEC would not draw at all, since when :SIDE reaches the value of 10, the program "stops, it doesn't draw anything." In fact, STOP and END each passively return control back to the most recently active procedure.

Overextension of mathematical operators – Children expressed confusion about the functions of numbers as inputs, and in arithmetic functions such as dividing the variable value, or addition of a constant to it, during successive procedure calls.For example, one child explained SHAPEC this way:

> . . . if SIDE equals 10 then stop. See, instead of going all forward 80, you just go forward 10. Then you're gonna stop. Then you're gonna go. Then (line 3) I guess what you're gonna do is keep on repeating that two times, so it'd be forward about 20 instead of forward 10, forward 20 (line 4), and you're gonna repeat 4, so it'd be forward 80 because it says repeat 4 forward side . . .

Numbers were also often pointed to as the mysterious source of discrepancies between the child's predictions and the results of program execution.

Mental model of embedded recursion as looping – The children were fundamentally misled by thinking of recursion as looping. While this mental model is adequate for *active* tail recursion, it will not do for embedded recursion, which requires an understanding of both *active* and *passive* flow of control. *The most pervasive problem for all children was this tendency to view all forms of recursion as iteration.* For example, one child explained the recursive call in program SHAPEB in the following manner:

> [the child explained what the first four lines did, then said]: "line 5 tells it to *go back up* to SHAPE, tells it to *go back up* and do the process called SHAPEB, this is the process [points to lines 2-4]. It *loops back up*, and it divides SIDE by 2 so then SIDE becomes 40 . . . [carries on explaining correctly that the procedure will draw two squares]"

In this example, the child clearly views tail recursion as a form of looping, rather than as a command to suspend execution of the currently executing procedure and pass control over to a new version of SHAPEB. However, in this case his wrong model leads to the right prediction, so he is not compelled to probe deeper into what the procedure is actually doing. This same child explained that SHAPEC:

> ". . . checks to see if SIDE 80 equals 10. If it does, end the program. Next, line 3 [the recursive call] tells it to *go back to the beginning* except to divide SIDE by 2 which ends up with 40. Then it goes down there (line 2) checks to see if SIDE is 10 . . . [then] *back to the beginning* . . . [continues to loop back until SIDE equals 10 then] checks to see if it equals 10, it does, stops. OK, a little extra writing there (points to lines 4 and 5). [Draws a dot in the paper to indicate his prediction of what the procedure will do and comments] and that is about as far as it goes because it never gets past this SHAPE (line 3). *It is in a loop* which means it cannot get past 'cause every time it gets down there (line 3), *it loops back up.*"

This time the child's explanation and prediction were incorrect since the SHAPEC program makes the turtle draw a series of three squares in a line, each twice as big as the previous one. The child expressed complete bewilderment when the procedure was executed, and could offer no explanation to account for the discrepancies. On the second program of this type, which makes the turtle draw three squares of different sizes inside one another, the child worked down to the recursive call and then said:

> "um. Wait a minute. I don't understand this. Well anyway, from past experience, like just now, I guess *it's not going to listen to that command* (points to the recursive call) and *it's going to go past it*, and it's going to [draw a square] and I guess *it's going to end then.*"

Again, when the procedure was run and the child saw he was wrong he expressed confusion, but instead of trying to explain what might cause the procedure to behave as it did he instead asked:

> "Is this the same language we used last year? Because last year if you said SHAPE, if you named the program in the middle of the program, *it would go to that program.* We did that plenty of times, but it's not doing that here. I don't know why."

The child blamed the language for not conforming to his expectations, but in doing so he indicated that at *some* level he knew the correct meaning of a recursive call: "It would go to that program." However, though he seemed to know the rule, when he worked through a program, his simpler, and in many cases successful, looping model prevailed.

DISCUSSION AND CONCLUSIONS

We believe these findings are important because they reveal that the children's conceptual bugs in thinking about the functioning of recursive computer programs are *systematic* in nature, and the result of weaker theories that do not correspond to procedural computation in Logo.

These findings also imply that, just as in the case of previous work with adults, programming constructs often do not allow mapping between meanings of natural language terms and programming language uses of those terms. Neither STOP or END stop or end, but pass control back. The reason that this is important for the Logo novice is that when their mental model of recursion as looping fails, they have no way of inferring from the syntax of recursion in Logo how flow of control does work. So they keep their inadequate looping theory, based on their successful experience with it for tail recursion, or blame discrepancies between their predictions and the program's outcomes on mysterious entities such as numbers, or the "demon" inside the language itself. An important issue of a development theory of programming then is: How do inadequate mental models get transformed to better ones?

For a developmental psychology of programming, we require an account of the various factors that contribute to learning central computational concepts. So far efforts to help novices learn programming languages through utilizing programming tutors or assistants have bypassed what we consider to be some of the key factors contributing to novice's difficulties working with computational formalisms. Beyond mistaken mental models about recursion, we have found these to involve atomistic thinking about how programs work, assigning intentionality and negotiability of meaning as in the case of human conversations to lines of programming code, and application of natural language semantics to programming commands. In studies underway, it appears that none of these sources of confusion will be intractable to instruction, although their pervasiveness in the *absence* of instruction, contrary to Papert's idealistic individual "Piagetian learning," suggests that self-guided discovery needs to be mediated within an instructional context.

ACKNOWLEDGMENTS

We wish to thank the participants of a workshop at MIT's Division for Studies and Research in Education, from Geneva and Cambridge, for provocative discussions of these issues. Sally MacKain provided invaluable assistance in running the studies, and providing transcripts.

REFERENCES

1. D. R. Hofstadter, *Godel, Escher and Bach: An Eternal Golden Braid*, Vintage Books, New York, 1979.

2. R. S. Siegler, Developmental Sequences Within and Between Concepts, *Monographs of the Society for Research in Child Development, 46*, (Serial No. 189), 1981.
3. S. Papert, *Mindstorms*, Basic Books, New York, 1980.
4. Y. Anzai and Y. Uesato, *Learning Recursive Procedures by Middleschool Children*, Proceedings of the Fourth Annual Conference of the Cognitive Science Society, Ann Arbor, Michigan, August 1982.
5. E. Soloway, J. Bonar, and K. Ehrlich, Cognitive Strategies and Looping Constructs: An Empirical Study, (Technical Report No. 242), Yale University Press, New Haven, Connecticut, 1982.
6. H. Kahney and M. Eisenstadt, *Programmers' Mental Models of Their Programming Tasks: The Interaction of Real-World Knowledge and Programming Knowledge*, Proceedings of the Fourth Annual Conference of the Cognitive Science Society, Ann Arbor, Michigan, August 1982.
7. J. Bonar, *Natural Problem Solving Strategies and Programming Language Constructs*, Proceedings of the Fourth Annual Conference of the Cognitive Sciences Society, Ann Arbor, Mighican, August 1982.
8. A. Collins and D. Gentner, *Constructing Runnable Mental Models*, Proceedings of the Fourth Annual Conference of the Cognitive Science Society, Ann Arbor, Michigan, August 1982.
9. A. R. Luria, *Cognitive Development*, Harvard University Press, Cambridge, Massachusetts, 1976.
10. S. Scribner, Modes of Thinking and Ways of Speaking: Culture and Logic Reconsidered, in *Thinking*, P. N. Johnson-Laird and P. C. Wason (eds.), Cambridge University Press, Cambridge, 1977.

Direct reprint requests to:

Dr. Roy D. Pea
Center for Children and Technology
Bank Street College of Education
610 West 112th Street
New York, NY 10025