Children's Mental Models of Recursive Logo Programs

by
D. Midian Kurland & Roy D. Pea
Center for Children and Technology
Bank Street College of Education

Abstract. Children with a year of Logo programming were asked to think-aloud about the function of some Logo recursive programs, and then to predict by hand-simulation of the programs what the graphics turtle will draw when the program is executed. If discrepancies arose, children were asked to explain them. A prevalent but misguided "looping" mental model of Logo recursion persisted even in the face of contradictions between program effects and the child's predictions.

Introduction. The power and beauty of recursion as a development in the history of programming languages (such as LISP and Logo) and its conceptual importance in mathematics, music, art and cognition generally is widely acknowledged (4). Less attention has been given to the developmental problem of how people learn to use the powers of recursive thought and recursive programming procedures. Our approach to this question is 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 problem solving approaches by children, not only adults (10). Understanding recursive functions in programming involves notational and conceptual problems, the latter including problems with understanding flow of control and data. Expert programmers are guided by a valid mental model of how program code controls computer operations. Novices' faulty models are adapted in response to direct instruction and feedback from their own programming and debugging experiences, in which conflicts between their current model and program behavior is reflected upon.

A widespread belief among computer educators is that young children can "discover" the powerful ideas formally present in programming simply through experimenting within a rich programming environment, as if unconstrained by prior understandings. This belief is largely due to Papert's (7) popular account of Logo, a LISP-like language designed for children to allow them to develop powerful ideas, such as recursion, in "mind sized bites". Many assume children can learn recursion through self-guided explorations of programming concepts in Logo. However, our observations of 8-12 year-olds indicate 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 problem descriptions, Anzai & Uesato (1) have shown how adolescents' understandings of recursive formulations of the factorial function is facilitated by a prior understanding of iteration. 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 iterative structure in a set of problems, 64% were also able to work out recursive solutions to a second problem set. However, only 33% of subjects who did not have prior iteration experience worked out the recursive functions. Anzai & Uesato conclude that understanding recursion is aided by an understanding of iteration, but urge caution when extending this point "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" (p. 102). While Anzai & 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 in implement it in specific programming formalisms. In addition to understanding recursion, the child must understand the logic and terminology governing the language's control structure. 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 (9) and tail recursion in SOLO, a Logo-like language (5), even after extensive instruction. Furthermore, Bonar (2) finds that prior natural language understandings of programming terms misleads novice programmers in their attempts at explaining how a program works. 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 common lament of 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

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 when the programmer explicitly directs the program to execute a specific procedure. However, when the execution of this version of the procedure is finished, control is automatically passed back to the suspended procedure, and execution resumes at the point where it left off. Passing of control is passive here because 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 inserts 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 program unit, and (2) directs flow of control back to the calling procedure so the program carries on.

(3) That there are 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.

Subjects. Seven children (2 girls and 5 boys, 11-12 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 50 hours of classroom programming time under the supervision of experienced classroom tea hers knowledgeable in the Logo, and who by choice followed Papert's "discovery" Logo pedagogy (7). All 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 levels of complexity: (1) procedures using only direct commands to move the turtle; (2) procedures using the iterative REPEAT command; (3) tail recursive procedures; and (4) embedded recursion procedures. This paper focuses on the revealing features of children's performance at levels 3 and 4. Examples of programs at levels 3 and 4 are (:SIDE = 80 for each):

Level 3: tail recursion program

```
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

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

Experimental procedure

Our choice of a method was guided by comprehension studies which utilize "runnable mental models" (3) or simulations of operations of world beliefs in response to specific problem inputs. Children were asked to think aloud about how a Logo procedure would work, then to hand simulate the running of each program line by using a turtle "pen" on paper. Then they were shown the consequences of running the program they had explained, and if their simulation mismatched the turtle's actions, they were asked to explain the discrepancies, and one additional problem at that level was presented.

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.

(1) 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 each command's definition 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 led to 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. Children often did not distinguish the meaning of a command line they were simulating from the meaning of command lines they expected to follow (e.g. lines that if executed would draw a BOX). For example, in program SHAPEC, one child said of the IF statement: "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 said: "this makes it draw a square."

Treating programs as conversation-like. As in understanding conversation, and in problems the non-schooled encounter in formal reasoning (where beliefs about the truth of an argument's premises are focused on rather than the validity of its form: (6), (8)), 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 code line (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 natural language analogy, leading them to believe that when the terms appear the program completely halts. Several children concluded that 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, and drawing occurs.

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 2 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.

(2) 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 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 SHAPEC 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 draws 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 its going to end then."

Again, when the procedure was run and the child saw he was wrong he expressed confusion, but instead of looking for an error of understanding, he 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. This is important for the Logo novice because 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.

Acknowledgements

This work was supported by the Spencer Foundation. We wish to thank 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 study and providing transcripts.

References

(1) Anzai, Y. & Uesato, Y. Learning recursive procedures by middleschool children. Proceedings of the Fourth Annual Conference of the Cognitive Science Society. Ann Arbor, August 1982.

(2) Bonar, J. Natural problem solving strategies and programming language constructs. Proceedings of the Fourth Annual Conference of the Cognitive Science Society. Ann Arbor, August 1982.

(3) Collins, A. & Gentner, D. Constructing runnable mental models. Proceedings of the Fourth Annual Conference of the Cognitive Science Society. Ann Arbor, August 1982.

(4) Hofstadter, D. R. Godel, Escher & Bach: An eternal golden braid. New York: Vintage Books, 1979.

(5) Kahney, H. & Eisenstadt, M. 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, August 1982.

(6) Luria, A. R. Cognitive development. Cambridge, Mass.: Harvard University Press, 1976.

(7) Papert, S. Mindstorms. New York: Basic Books, 1980.

(8) Scribner, S. Modes of thinking and ways of speaking: Culture and logic reconsidered. In Johnson-Laird, P.N. & Wason, P.C. (Eds.), Thinking. Cambridge: Cambridge University Press, 1977.

(9) Soloway, E., Bonar, J. & Ehrlich, K. Cognitive strategies and looping constructs: An empirical study. Comm. ACM, 1983, in press.

(10) Siegler, R. S. Developmental sequences within and between concepts. Monog. of the Society for Research in Child Development, 1981, 46 (Serial No. 189).