ED 280 719 SE 047 882

AUTHOR Lee, Okhwa; Lehrer, Richard
TITLE Conjectures Concerning the Origins of Misconceptions in LOGO.
PUB DATE Apr 87
NOTE 34p.; Paper presented at the Annual Meeting of the American Educational Research Association (Washington, DC, April 20-24, 1987).
PUB TYPE Reports - Research/Technical (143) -- Viewpoints (120) -- Speeches/Conference Papers (150)

EDRS PRICE MF01/PC02 Plus Postage.
DESCRIPTORS Computer Oriented Programs; *Computer Science Education; *Concept Formation; Educational Research; *Error Patterns; Higher Education; Logic; *Mathematics Instruction; Microcomputers; *Programing
IDENTIFIERS *LOGO Programing Language; Mathematics Education Research

ABSTRACT

Seven graduate students in a seminar on classroom computing received instruction in LOGO programming. Programming protocols were collected periodically and examined for errors and misconceptions; in-depth interviews were conducted in order to understand specific misconceptions better. As novice students transit from instruction to experience in LOGO, they develop a systematic set of misconceptions concerning the flow of control in programs. These misconceptions result in programming errors including unnecessary repetition of statements, inadequate use of conditional statements, non-existent or inappropriate combination of Boolean operators, failure to initialize variables, and difficulty transferring simple recursive structures developed in the graphics mode to the list processing mode. In addition, students with prior programming experience in BASIC inappropriately attempt to superimpose the iterative FOR...NEXT loop of this language onto recursion in LOGO. The origins of these misconceptions are traced to general properties of cognition and also to specific instructional practices. Four recommendations for instructing novices in LOGO are included. (Author/MNS)

Conjectures Concerning the Origins of Misconceptions in Logo

Okhwa Lee
(Korea Advanced Institute of Science and Technology)

Richard Lehrer
(University of Wisconsin - Madison)

Running head: ORIGINS OF MISCONCEPTIONS

Abstract

As novice students transit from instruction to experience in Logo, they develop a systematic set of misconceptions concerning the flow of control in programs. These misconceptions result in programming errors including unnecessary repetition of statements, inadequate use of conditional statements, non-existent or inappropriate combination of Boolean operators, failure to initialize variables, and difficulty transferring simple recursive structures developed in the graphics mode to the list processing mode. In addition, students with prior programming experience in BASIC inappropriately attempt to superimpose the iterative FOR...NEXT loop of this language onto recursion in Logo. The origins of these misconceptions are traced to general properties of cognition and also to specific instructional practices. We conclude with four recommendations for instructing novices in Logo.

Conjectures concerning the origins of misconceptions in Logo

As computers become more available in school systems, their functions expand from administrative record-keeping to include instruction in classrooms. Programming is one of the most widely practiced instructional activities related to computer "literacy" (Becker, 1982). Consequently, research is needed to delineate the prospects and pitfalls students encounter as they learn a programming language. Accordingly, this research details some of the misconceptions students develop as they learn to program in Logo (Papert, 1980) for the first time.

Misconceptions may be distinguished from simple errors in that the former derive from application of a coherent organization of knowledge about programming, also called a schema (Brewer & Nakamura, 1984, Stevens, Collins & Goldin, 1979). To illustrate, a misconception-based error may result from a misunderstanding about the conditions of application for a looping construct such as recursion. In this instance, a student may fail to include a conditional test when creating a recursive function. The error would be considered schematic in that it results from application of incomplete or incorrect knowledge. Consistent repetition of errors is expected for as long as the learner holds the incorrect conception. In contrast, another error might include a failure to recall the precise syntax for a loop, especially when the loop is embedded within a larger problem. This error is more easily remediated and might not appear when the student is engaged in a problem less demanding of working memory (Anderson, Farrell &

Sauers, 1984). Consequently. this error would not be considered schematic.

A second purpose of this study was to contrast the learning of students with prior programming experience with an imperative language such as BASIC with the learning of students who had no prior programming experience. Although it is widely assumed that prior experience with a programming language will inevitably transfer positively to the learning of a second language, we had reservations which are detailed later. To anticipate, an imperative language such as BASIC often includes no provision for recursion. Hence, students learning Logo-based recursion may incorrectly apply iterative concepts learned during BASIC programming. Thus, prior BASIC learning may interfere with, rather than facilitate, acquisition of Logo in some instances.

Logo was chosen for this study because it is an exemplar of a declarative language (Eisenbach & Sadler, 1985) and because it is often taught in schools. Among the programming concepts in Logo, the flow of control (the order of statement execution) was selected as the focus of this study. Previous research indicates novice programmers experience difficulty implementing an appropriate flow of control, yet knowing how to organize the flow of control constitutes the foundation of structured programming (Copper & Clancy, 1982; Luehrmann & Peckham, 1984; Mayer, 1981; Miller, 1981; Pratt, 1978; Soloway, Ehrlich, Bonar & Greenspan, 1984; Soloway, Bonar & Ehrlich, 1983; Vessey & Weber, 1984). Application of structured programming techniques results in well written programs, and consequently receives particular emphasis during instruction in programming.

Models of errors and misconceptions

Models of errors and misconceptions emerge from programming ideals. Ideal programs, like ideal written compositions, usually reflect a top-down modular structure which accomplishes the intended task efficiently and clearly from the perspective of a potential audience (Bennett & Walling, 1985). Youngs (1974) generated a taxonomy for programming errors which indexes increasingly severe violations of ideal programming practices. He proposed four general types of programming errors: clerical, syntactic, semantic, and logical. Clerical errors are simple typographic errors; a programmer mistypes an expression. Syntactic errors are incorrect expressions in the language which cause the interpreter or compiler to generate error messages (DuBoulay & O'Shea, 1981). In contrast, "semantic errors try to make the computer carry out impossible or contradictory actions, though the expression is syntactically correct." (DuBoulay & O'Shea, 1981, p 151). Lastly, "logical errors concern the mapping from the problem to the program. Here the program does not produce the desired result i.e. it does not do the job it was designed to do. These errors can be due to either poor planning of the algorithm, or incorrect expression of the algorithm in code form." (DuBoulay & O'Shea, 1981, p. 151). Further amplifying Youngs' taxonomy, DuBoulay & O'Shea (1981) suggested another category: stylistic errors, which are similar to Soloway and Ehrlich's (1984) programming discourse errors. Neither of these types of errors (stylistic errors and programming discourse errors) interferes with program execution or produces inaccurate results. Instead, they violate standard programming conventions, thereby making the program inefficient or difficult to comprehend and debug. Considering a program as a composition (Newkirk, 1985), stylistic errors correspond to deficient monitoring of one's intended audience. In summary, clerical and syntactical errors are relatively minor when contrasted to the higher-order errors concerning semantics, logic and style.

Previous research indicates a preponderance of higher-order errors in the protocols of novice programmers (Boies & Gould, 1974; DuBoulay & O'Shea, 1981, Friend, 1975, Youngs, 1974). In these studies, syntactic and clerical errors did not seem to be a great source of difficulty for novices because these errors are not as difficult to debug as are semantic errors. For instance, syntactic and clerical errors are detected easily by the interpreter or compiler of the language but higher-order errors (semantic, logical and stylistic errors) are more subtle. Often, higher-order errors need to be debugged without the benefit of informative error messages. Consequently, in this research we focused on higher-order errors.

Flow of control

Novices usually experience difficulty when learning about the flow of control, which requires knowledge of conditional statements and loopings. Novice programmers often use looping constructs inappropriately (Joni, Soloway, Goldman & Ehrlich, 1983; Soloway, Bonar & Ehrlich, 1983; Soloway, Ehrlich, Bonar & Greenspan, 1984; Waters, 1979; Pratt, 1978; Woodward, Hennell & Hedley, 1979). Inapproriate use of looping constructs may be traced to several sources. For instance, novices typically lack strategic knowledge concerning which statements belong in the loop, what to put outside of the loop, and where to put conditional statements (Soloway, Bonar & Ehrlich, 1983; Vessey & Weber, 1984). In short, novices have declarative knowledge concerning loop construction but must learn how to implement that knowledge (Anderson et al., 1984).

Flow of control in Logo. Controlling the flow of a program entails combinations of conditional statements and loopings, as noted above. Conditional statements available in the Logo language include "IF condition [THEN] [ELSE]" and "TEST IFFALSE IFTRUE", all of which may be amplified by Boolean operations such as "AND", "OR", and "NOT". Two commonly employed looping constructs available in the Logo langauge are the REPEAT command and recursion. The REPEAT command is analogous to a FOR...NEXT loop in BASIC and is used when the number of looping executions is known a priori. In contrast, the recursive looping construct can be used when the number of executions of the loop body is not known beforehand. In recursion, statements in the loop body will be executed until a stop rule becomes true. Therefore, recursive procedures must include a test of conditions. In brief, recursion enables implementation of DO-WHILE (test to see whether a condition is true at the start of the loop) and REPETITION-UNTIL (test condition at the end of the loop) loops (Martin, 1985). Consequently, successful construction of recursive loops in Logo requires knowledge of how to create conditional statements, where to put the conditional statements and when to apply recursive rather than iterative (REPEAT command) procedures.

Using Looping Constructs

Successful use of looping constructs requires knowledge of how to (1) set up loop control variables, (2) initialize variables, (3) update variables, and (4) terminate the loop (Pratt, 1978). We briefly describe each of these in order to elucidate later discussion.

Loop control variables. The loop control variable controls the loop execution; it allows either the loop to continue or to stop. Looping constructs must have loop control variables unless the termination locus of the loop is known before execution of the loop. In the example below (written in Logo), the variable :list1 in the conditional statement, IF :list1 = [] [STOP], acts as a loop control variable. (This procedure counts the number of items in a list and prints this number. It assumes that the variable "total" is initialized to 1 before execution of this procedure.)

```
TO counter :list1 :total
IF :list1 = [] [STOP]
PRINT :total
counter BUTFIRST :list1  SUM :total  1
END
```

Variable initialization. There are two ways of initializing variables in Logo: by using the MAKE command or by using procedure parameters. Initialization of loop control variables should be done before looping. In the previous example, the procedure will return an error message if "total" is not given a value or the wrong answer if "total" is not initialized properly.

Variable updating. Loop control variables should be updated within a loop. As in variable initialization, either the MAKE command or the manipulation of procedure parameters can be used for updating variables. In the example given above, the loop control variable (:list1) was updated with the parameter manipulation (counter BF :list1 SUM :total 1). Logo promotes use of parameter manipulations for updating variables because parameter manipulation of recursive procedures results in a more compact form of updating than the MAKE command (using the MAKE command requires an extra statement line whereas parameter manipulation does not).

Loop termination. Termination of looping can be controlled by a conditional statement of a loop control variable. As noted previously, the contents and placement of a conditional statement are crucial for successful termination of recursive loops.

In this study, we intended to apply the framework described above to examine errors as students attempted to develop appropriate control structures for problems we presented to them.

In sum, we proposed to examine students' implementation of looping constructs, particularly recursion, as they learned Logo for the first time. Consequently, we investigated students' implementations of conditional statements, loop control variables, variable initialization and updating, loop termination, and the overall flow of control.

Method

Subjects. Participants were 7 graduate students participating in a seminar on classroom computing (Fall, 1985). Four students in group B (ASIC) reported previous programming experience with BASIC. The other three students were characterized as group A (MATEUR: no programming experience). Anecdotal observations of 15 students who had participated in a similar seminar the previous semester (Spring, 1985) are also occasionally referred to in the following sections. However, except where explicitly mentioned, the results and discussion deal with the 7 students.

Instructional procedures. Each student received instruction in Logo programming for 1.5 hours each week for eight weeks. Each student had his or her own microcomputer and had unlimited access to these microcomputers during the semester for practice. Instructional topics included turtle graphics, procedures, top-down programming, planning, variables (global vs local), conditional commands, recursion, words and lists, looping constructs (while, do until, repeat loops, stop rules), and logical operations (and, or, not). During the instruction, three programming assignments were given to students. The programming assignments were sufficiently complex to require nontrivial use of recursive procedures, but were simple enough to be solved by novices in a comparatively short period of time. Students also read a wide variety of research papers concerning the use of Logo in schools. The programming problems were as follows:

1. Draw a series of concentric circles with a single variable input for the radius. The procedure should start with the smallest circle and stop within the boundaries of the screen. Each ring should be of a different color.

2. Given two lists of any length, print all possible pairwise combinations of the elements of the lists.

3. Construct a rock-scissors-paper game for one player and a computer. Recall that to win: scissors cuts paper, paper covers rock, and rock smashes scissors. To tie: both the player and the computer pick the same thing. To lose: anything else.

Observational procedures. Students' programming protocols were collected periodically as they wrote programs, and in-depth interviews were conducted informally in order to understand specific misconceptions better. Students' performance was analyzed with respect to: (1) initialization of loop variables, (2) updating loop control variables, (3) loop termination, (4) overall structure of the flow of control, (5) unnecessary repetition of commands and procedures, (6) use of infix (e.g. 2 + 3) and prefix (e. g. SUM 2 3) notation, and (7) use of conditional statements.

Results

Errors and misconceptions common to most subjects regardless of previous programming experience are reported first, followed by those which seemed more prevalent among students with prior programming experience with BASIC (n = 4).

Errors and misconceptions common to both groups

Common errors included (1) misconceptions concerning the use of infix notation in recursive procedures, (2) unnecessary repetition of statements, (3) inappropriate use of conditional statements, (4) inefficient use of logical operations and (5) inappropriate variable initialization. In this section, we also include conjectures concerning the origins of these errors.

<u>Negative transfer resulting from the use of infix notation</u>. Among 15 participants in the first seminar (Spring, 1985), all were able to write simple recursive procedures in the graphics mode but most (n = 13) were not able to transfer spontaneously this knowledge to list processing. For example, students were able to write a procedure such as:

```
TO square.grow :side
IF :side > 100 [STOP]
REPEAT 4 [FD :side RT 90]
square.grow :side + 5
END
```

Given a small numerical input, this procedure will draw a set of embedded squares. Similarly,

```
TO shrink.lst :list1
IF :list1 = [] [STOP]
PR :list1
shrink.lst BUTFIRST :list1
END
```

will reduce an input list to null, printing its elements at each step. We thought students able to create recursive procedures in the graphics mode, such as the one presented in the first example, would have little trouble with similar recursive procedures in the list mode (example 2). Yet, most students failed to transfer the structures from graphics to list environments without considerable help. For instance, many students failed to recognize the role of BUTFIRST in the second example, but had no difficulty understanding the role of addition in the first example.

The origin of this failure to transfer the structure of recursive procedures from the graphics to the list mode appeared to derive from the transition between infix and prefix notations (:SIDE + 5 is an example of an infix notation and SUM :SIDE 5 is the corresponding prefix notation). That is, in the graphics mode, students knew that :side + 5 was an operation on a variable but perhaps the tacit (and automatic) form of this knowledge obscured the role of operations which manipulate parameters in recursion. Hence, the prefix form of the operation required in list processing (e. g. BUTFIRST :list1) was difficult for students to comprehend. We speculate that students incorrectly focused on the surface feature of the form of the operation (infix notation) rather than on the more general notion that recursive procedures act through operations at each call of the procedure. Follow-up instruction, re-formulating the first procedure with the prefix notation, appeared to remove this source of confusion. In the second class (n = 7) all students spontaneously made the transition between the graphics and list modes for simple recursive functions.

Unnecessary repetition of statements. All students unnecessarily repeated commands or procedures. The example listed below demonstrates a prototypical instance of unnecessary repetition (denoted by *); each procedure (print.tie, print.win, print.lose) has the same subprocedure (play.again).

```
TO pro1
print.tie
print.win
print.lose
END
TO print.tie
IF :result = "tie [PR "tie  play.again ]      *
END
TO print.win
IF :result = "win [PR "win  play.again ]      *
END
TO print.lose
IF :result = "lose [PR "lose  play.again ]    *
END
```

The subprocedure, `play.again', can be called once at the same level as the other procedures. The same program without unnecessary repetitions is:

```
TO pro1
print.tie
print.win
print.lose
```

```
play.again
END
```

where the procedures (print.tie, print.win, print.lose) do not contain play.again as a subprocedure.

Another example of unnecessary repetition is listed below. The three statements with * in the example program can be condensed to a single statement; IF :a = :b [PR "tie]

```
TO pro1
IF AND :a = "1  :b = "1  [PR "tie]      *
IF AND :a = "1  :b = "2  [PR "win]
IF AND :a = "2  :b = "1  [PR "lose]
IF AND :a = "2  :b = "2  [PR "tie]      *
IF AND :a = "2  :b = "3  [PR "win]
IF AND :a = "3  :b = "2  [PR "lose]
IF AND :a = "3  :b = "3  [PR "tie]      *
END
```

Like the error involving the use of infix notation, repetition errors appear to result from novices' attending to the surface features of a program -- a means to accomplish an end, without considering the finer nuances of audience (comprehension of the flow of control by someone else or by the author at a later date). This was manifested behaviorally as poor planning. Stated another way, like novice writers of text, our students tended to create small local plans, leaving the details of how these local plans would interface as a problem to be solved later. Therefore, as they needed new procedures, they added these procedures without considering the entire structure of the program. Because students did not receive any error messages from the interpreter, this error in style (unnecessary repetition) resulted.

Inadequate use of conditionals. Two errors are identified: failure to use the ELSE part of the IF THEN ELSE conditional, and failure to recognize when TEST IFTRUE IFFALSE was more appropriate than IF. When using the IF THEN ELSE construct in Logo, the ELSE component can be omitted if it is not needed. Ease of use, however, apparently leads to the development of an impoverished structure for this form of the conditional. That is, students did not use the ELSE component even when it would have been advantageous to do so. The following is an example from a game program which does not use the ELSE component even though it would have been helpful to do so. The procedure consists of 7 IF conditionals whereas the same procedure can be implemented with two IF conditionals which include ELSE. (The procedure below is edited with minor changes for clarity. "r" denotes rock, "p" paper, and "s" scissors.)

```
TO compare :computer :player
IF :computer = :player [OP [it's a tie]]
IF AND :computer = "r  :player = "p  [OP [you win]]
IF AND :computer = "s  :player = "r  [OP [you win]]
IF AND :computer = "s  :player = "p  [OP [sorry, you lose]]
IF AND :computer = "p  :player = "s  [OP [you win]]
IF AND :computer = "p  :player = "r  [OP [sorry, you lose]]
IF AND :computer = "r  :player = "s  [OP [sorry, you lose]]
END
```

The following procedure will test for all conditions listed above with two IF statements. The last statement constitutes the ELSE component. The revised procedure also combines Boolean operators which we describe more fully in the next section.

```
TO compare :computer :player
IF :computer = :player [op [it's a tie]]
IF (OR AND :computer = "r :player = "p
       AND :computer = "s :player = "r
       AND :computer = "p :player = "s)
         [OP [you win]]   [OP [sorry, you lose]]
END
```

Students also failed to recognize when the second form of the conditional, TEST IFTRUE IFFALSE, should be applied. For example, note the comparative clarity of structure resulting from replacement of the IF... with TEST in the last procedure.

```
TO compare :computer :player
IF :computer = :player [OP [it's tie]]
TEST (OR AND :computer = "r :player = "p
         AND :computer = "s :player = "r
         AND :computer = "p :player = "s)
IFT [OP [you win]]
IFF [OP [sorry, you lose]]
END
```

Once again, we believe unexamined tacit knowledge leads to programming errors. In this instance, Logo's ease of use obscures the ELSE component of the IF THEN ELSE construct, allowing students to abstract an incomplete form composed of the first two terms only. On the other hand, students' failure to use the TEST construct may be traced to inadequate provision of enough concrete examples when contrasted with their more widespread exposure to the IF...THEN...ELSE.

Inefficient use of logical operators. Among the three logical operations in Logo, AND, OR and NOT, the AND was used most often in students' protocols. Only two students used logical operators other than AND even though combinations of operators would make programs more efficient and less demanding of working memory. For instance, consider the difference in the example presented above between the 7 IF statements required without combining operators to the two statements which resulted in part from combining logical operators.

Studies in Boolean logical thinking suggest four possible ways to combine truth tables: conjunction, disjunction, conditional, and biconditional. Among these four, conjunction appears to be easiest for novice programmers to code (Mayer, 1983, Gorman, 1982); conjunction is coded as "AND". Consequently, subjects preferred AND and avoided the more complex mappings of logic into code entailed by combining Boolean operators.

Variable initialization. Beginning programmers often forget to initialize variables within programs which often leads to unanticipated action by the program. In Logo, variables may be assigned global values in a general workspace independent of any specific procedure. Hence, when writing programs with variables, students often relied upon this tactic to initialize variables.

However, such a practice decreased the "visibility" of a program and occasionally resulted in errors. For example, students saved programs which relied upon the initial values of variables in the workspace at the time during which the program was saved. Hence, the values of these variables were not evident upon inspection of any of the procedures in which these variables were used. Consequently, initialization of these variables was "invisible" to the reader of any of the procedures. Subsequent execution of these programs often resulted in errors when current values of variables in the workspace did not match the values at the time when the program was saved.

Failure to initialize a variable within a procedure is encouraged by some features of Logo, as noted above. Moreover, Miller (1981) found that in natural language initialization is implicitly done while in a programming language initialization should be specified explicitly. Therefore, initialization requires novice programmers' attention. Although a failure to initialize a variable is usually corrected easily by referring to the error messages generated by the Logo interpreter, it leads to the development of a more subtle stylistic error -- programs with invisible structures.

Errors and misconceptions which Varied by Group

Errors and misconceptions which varied in proportion between groups (group A, n=3, no prior programming experience; group B, n=4, prior programming experience with BASIC) include those concerning (1) use of a "goto" looping structure for recursion, (2) inappropriate use of syntax to initialize variables, (3) inefficient use of syntax to update variables, and (4) inefficient overall structure (unnecessary nesting).

<u>"Goto" recursive looping structure</u>. All four students in group B(ASIC) constructed a recursive loop structure with an iterative, circular form, derived from the conceptual model of the "goto" statement in the BASIC programming language. None of the students in group A(MATEUR) developed this structure. Two levels of errors (shallow and deeper) based on this misconception were observed. An example of the shallow level for the "goto" looping can be seen in the rock-scissors-paper programming assignment. Here, students in group B thought that calling the first procedure in the loop structure listed below would start the whole loop, as it would in BASIC. Two students constructed this procedure in order to ask a player if s/he wished to play another game. The procedure is restated in BASIC in order to demonstrate our conjectures concerning the origins of this structure. The * denotes the "goto" construct.

<u>Logo</u>

```
TO main
intro
init
play
check
IF :answer = "Y [intro]  *
END
```

<u>BASIC</u>

```
10 gosub 100 :rem intro
20 gosub 200 :rem init
30 gosub 300 :rem play
40 gosub 400 :rem check
50 IF $answer = "Y goto 10  *
60 END
```

As an example of the deeper level of errors concerning the "goto" loop structure, students in group B constructed a top-level procedure which contained another procedure which in turn called the top-level procedure. In two protocols obtained from the first programming assignment, the procedure A called B, the procedure B called C, and the procedure C called A in order to "complete" the looping. That is, subjects defined the following procedure.

```
TO main
A
  B
    C [A]
END
```

The iterative quality of this structure may be traced to students' prior programming experience with the FOR...NEXT loop in BASIC. In this instance, because students did not have a FOR...NEXT construct in Logo, they attempted to define one by using the second call of procedure A as a "goto" statement. Students expected the control to return to the first execution of A. Stated another way, the iterative FOR...NEXT loop constitutes the "deep" structure of the misconception which results in a variety of surface manifestations, two of which are presented above.

Initialization of variables. Students in group B preferred the MAKE command to parameter manipulations for variable initialization. No such preference was observed for students in group A. Recall that there are two ways of initializing a variable: manipulation of procedure parameters and the MAKE command. The MAKE command is syntactically similar to the LET command in BASIC, which is used to initialize variables. Consequently, students in group B adopted MAKE as the method of choice for variable initialization. This choice may have contributed to their difficulties with recursion (detailed above) because recursive procedures are implemented most easily with parameter manipulation. Three students in group B consistently used the unnecessary MAKE command when they used recursive procedures. An example follows.

```
TO pro1
MAKE "number 0
counter :number
END
TO counter :number
IF :number = 20 [STOP]
(PR SE [I am counting ] :number)
counter SUM :number  1
END
```

where a more efficient procedure is (assuming the same procedure "counter" is used):

```
TO pro1
counter 0
END
```

<u>Updating variables</u>. Group B students also preferred the MAKE command to parameter manipulation for updating the value of variables. In contrast, students in the first group were more likely to use parameter manipulation. As noted previously, this tendency

among group B students resulted from their familarity with the LET command in BASIC. Although we did not classify this tendency as an error, it further indicates students' superimposition of iterative concepts onto recursive structure, despite instruction which emphasized the difference between the two. Examples of updating variables follow.

<u>Ideal Practice</u>	<u>Group B Practice</u>
TO pro1 :counter	TO pro1 :counter
IF :counter = 10 [STOP]	IF :counter = 10 [STOP]
(statements of loop body)	(statements of loop body)
pro1 :counter + 1	MAKE "counter :counter + 1
END	pro1 :counter
	END

<u>Inefficient overall structure</u>. The overall structure of the flow of control determines the visibility of a program (Greene, 1983). Well-structured programs show the logic of the control structure more easily than programs containing unnecessarily nested subprograms or "spaghetti" code. All the students in group B showed unnecessarily nested procedures where each procedure was better coded as independent of all others. Group A performed better than group B in this matter. For instance, in the example below, students in group B nest the procedure "check" within "play" which in turn is nested within "get.value".

<u>Ideal Practice</u>	<u>Group B Practice</u>
TO top	TO top
intro	intro

```
get.value                get.value
play                          play
check                             check
END                      END
```

Franta and Maly (1976) reported that deeply nested constructs, such as those characteristic of group B programming practices, are detrimental to program readability and maintenance. The unnecessary nesting shown in the example above decreases the comprehensibility and the readability of the program. We speculate this unnecessary nesting results from students continued attempts to superimpose BASIC's iterative structure onto Logo's recursion. In this instance, nested procedures represent a concrete way to insure a transfer of control somewhat analogous to that of BASIC.

Discussion

A well-constructed program has characteristics of good writing. Newkirk (1985) suggests three qualities of well-written programs as parallels to good composition: planning, audience awareness, and revision. Planning refers to top-down analysis and attention to the interface among procedures as these relate to intended goals, audience awareness may be compared to adherence to programming conventions or styles, and revision refers to debugging. Novice programmers in Logo, like novice writers, exhibited misconceptions about the flow of control in programs which indicated incomplete or inaccurate knowledge about all three aspects of the architecture of well-written programs.

In the area of planning, all students included unnecessarily redundant statements or procedures in their programs, reflecting a plan-as-you-go method in preference to the development of prior plans. As students became more knowledgeable about how to solve problems with Logo, plan-as-you-go evolved into partial plans in which one procedure was nested within another, a practice which decreased the readability of these programs. A variety of other errors also contributed to decreasing a program's accessibility to an audience (to either the programmer or an outside reader). For example, students failed to combine Boolean operators resulting in programs which were unnecessarily long and difficult to follow. Obscure programs also developed from students' incomplete specification of the ELSE in the IF THEN ELSE construct. Expanding upon the last quality of good composition, revision, consider the difference between editing and revising. Editing concerns minor changes in text designed to correct minor errors in syntax or in punctuation. In contrast, revision refers to attempts to change the text to make it more complete or coherent. Applying this difference to programming, subjects' debugging usually involved editing and not revising. That is, students often edited procedures for violations of syntax or to change unanticipated products of these procedures. Comparatively little time was spent, however, in editing the interface among procedures for purposes of increasing the visibility of the program. Hence, we conclude that these novice programmers edited but did not often revise their programs.

Other results indicated that prior programming experience may inhibit rather than facilitate learning. In this instance, students with prior programming experience in BASIC attempted to assimilate the construct of recursion to the iterative FOR...NEXT construct of BASIC. Consequently, although the programs constructed by these students were recursive in that they called themselves, their implementation included a circular nesting of procedures which was both unnecessary and deleterious to the visibility of the program. We consider such procedures as an attempt to reconstruct a FOR...NEXT loop with the Logo equivalent (in the minds of these students) of a GOTO statement in BASIC. The influence of this iterative-based prior knowledge also manifested itself in these students' preference for the MAKE command to update the values of variables. Although not an error, this preference clearly stemmed from the LET command in BASIC, a command which is often used in iterative looping implemented in this language.

Taken as a whole, these results fit the model of the acquisition of programming skills developed by Anderson et al. (1984) which describes the difficulties encountered by people as they make the transition between instruction and experience. In the Anderson et al. theory, novice programmers rely upon templates and concrete examples (structural analogy) when they first translate a programming problem into code. Similarly, subjects in the first sample (n = 15) experienced difficulties making the transition from simple recursion in the graphics mode to simple recursion in the list-processing mode. In this instance, use of infix notation in the graphics mode provided a flawed template which inhibited transfer to list-processing applications. Moreover, the attempt by students with prior programming experience in BASIC to graft FOR...NEXT iteration onto recursion resulted from use of an inappropriate template. Anderson et al. also specify a knowledge compilation mechanism which acts to transform procedures developed using structural analogy so as to increase their efficiency. The structure of the goal tree created during a problem-solving episode determines which parts of the episode "belong together" and hence, can be transformed to increase efficiency. From this perspective, incomplete rules may develop due to retrieval failures resulting from heavy demands on working memory. For example, although subjects in this study could use the IF THEN ELSE statement appropriately in conjunction with simple examples provided for instruction, in the context of a more complex problem (rock-scissors-paper) they developed productions which deleted the ELSE portion of the construct. Such deletion may have resulted from working memory failure as students encoded the multiple conditional branches of this game. Although some of the misconceptions and errors developed by students appear inevitable given the general characterisitics of cognition presented in the Anderson et al. model, others appear susceptible to instructional remediation. Consequently, we conclude with conjectures about instructional practices which may prevent some of the misconceptions we observed.

Instructional Implications

Four instructional implications based on the results are suggested: introducing prefix notation with arithmetic operations, recursion with parameter manipulations and stop rules, concrete examples of the TEST IFT IFF construct, and planning before coding.

<u>Prefix notation with arithmetic operations</u>. If arithmetic operations are introduced prior to list processing, use of prefix notation rather than infix notation provides a better template for subsequent instruction in list processing. When students are not introduced to the prefix notation, they are required to master two alien concepts for list processing: prefix notation and list processing operations. Thus, students might preferably be introduced to a recursive function as follows:

```
TO countup :number
IF :number < 0 [STOP]
countup DIFFERENCE :number 1
PRINT :number
END
```

rather than the more usual practice of using the infix notation of :number - 1.

<u>Recursion with parameter manipulations and stop rules.</u> Recursion should be introduced as a procedure wherein the call to itself includes a parameter manipulation. Parameter manipulation helps students understand that each call of the procedure results in a copy of the procedure with distinct values for variables. The definition of the procedure should also include a stopping rule. We suggest the common practices of introducing recursion without

parameter manipulation and without a stopping rule lead ineluctably to iterative-based misunderstandings of recursion. These misunderstandings are especially likely if students have prior programming experience with BASIC. For example, the simple recursive function presented above includes both a stopping rule and a single parameter manipulation. Hence, it serves as a good template for the development of other recursive procedures.

<u>Concrete examples of the TEST IFT IFF construct</u>. All students had difficulties combining logical operations and conditional statements. Use of multi-logical operations should be exemplified with the TEST IFT IFF construct rather than the IF THEN ELSE construct because the TEST IFT IFF conditional represents each component of the IF, THEN, ELSE conditionals as a separate statement. Consequently, programmers can deal with one concept at a time: for instance, they can combine logical operations without the additional burden of coordinating the components of the IF THEN ELSE construct. By reducing working memory demands, novice programmers are better able to focus upon coordination of logical operations.

<u>Planning before coding</u>. Although students readily accepted the idea of planning before coding, such declarative knowledge had to be translated into specific procedures. We found that students had the most difficulty with the "grain" of the plan. For instance, students often planned specific procedures, reflecting "local" plans, while they simultaneously failed to consider relations among procedures. Hence, we speculate that concrete examples of the interplay between global and local plans may help students develop better plans. In

this instance, a rigid adherence to top-down planning is not useful because novices do not have enough knowledge to develop effective plans in a purely top-down fashion. Just as writers use an outline dialectically with ongoing text processing, novice programmers need to view an initial top-down analysis as a temporary framework which will likely need to be revised during the course of programming.

In summary, results of this study corroborate those of previous studies which suggest that novice programmers tend to develop a systematic set of misconceptions as they transit from instruction to experience. Although the origins of these misconceptions may be traced to more general properties of cognition (Anderson et al., 1984), several of the misconceptions observed in this study appear remediable through instruction. Finally, prior programming experience may lead to negative transfer when old templates which have surface but not deep-structure resemblance to constructs in the new langauge are nevertheless used as analogical bridges.

Bibliography

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

Becker, H. (1982). Microcomputers in the classroom -- dreams and realities. Technical report No. 319, January, Center for social organization of schools, The Johns Hopkins University.

Bennett, H. & Walling, D. (1985). Once again, Structured programming: Is it necessary ?., Computers in the schools Double Issue Logo in the schools Haworth Press, Inc. Vol.2(2/3) Summer/Fall

Boies, S.J. & Gould, J.D. (1974). Syntactic error computer programming. Human Factors, 16, 253-257.

Brewer, W.F. & Nakamura, G.V. (1984). The nature and functions of schemas. In R.S. Wyer & T.K. Srull (Eds.), Handbook of Social Cognition. (pp 119-160). Hillsdale, NJ: Erlbaum.

Cooper, D. & Clancy, M. (1982). Oh! Pascal! An introduction to programming. W-W-Norton & Company, New York.

DuBoulay, B. & O'Shea, T. (1981). Teaching novice programming. In M.J. Coombs & J.L. Alty (Eds.) Computing skills and the user interface., (pp 147-200). Academic Press. London.

Eisenbach, S. & Sadler, C. (1985). Declarative languages: An overview Byte, 10 (8), 181-197.

Franta, W. R. & Maly K. (1976). A multilevel flow control statement. Intern. J. Computer Math, Section A Vol. 5 297-307.

Friend, J. (1975). Programs students write. Technical report No.

257, Institute for mathematical studies in the social science, Stanford University.

Gorman Jr., H. (1982). The Lamplight project. Byte, 7 (8), 331-333.

Green, T.R.G. (1983). Learning big and little programming languages. In A. C. Wilkinson (Ed.) Classroom computers and cognitive science, Academic press. pp. 71-93.

Joni, S. & Soloway, E. & Goldman, R. & Ehrlich, K. (1983). Just so stories: How the program got that bug. SIGCUE, Fall, 13-26.

Luehrmann, A. & Peckham, H. (1984). Computer literacy: Survival kit. New York: McGraw-Hill.

Martin, J. (1985). System design from provubly correct constructs. Englewood Cliffs, NJ:Prentice-Hall.

Mayer, R. E. (1981). The psychology of how novices learn computer programming. Computing Surveys, 13 (1) March, 121-141.

Mayer, R. (1983). Thinking, problem solving, cognition. San Francisco: Freeman.

Miller, L.A. (1981). Natural language programming: Styles, strategies, and contrasts. IBM System Journal, 20 (2), 184-215.

Newkirk, T. (1985). Writing and programming: Two modes of composing. Computers, Reading and Language Arts (CRLA), 2 (2), 40-43.

Papert, S. (1980). Mindstorms: Children, Computers, and Powerful Ideas Basic Books, Inc.

Pratt, T. W. (1978). Control computations and the design of loop control structures. IEEE Transactions on software engineering, Vol. Se-5, No.2, 81-89.

Ripley, G. D. & Druseikis, F. C. (1978). A statistical analysis of syntax errors, Computing languages, 3, 227-240.

Soloway, E. & Ehrlich, K. & Bonar, J. & Greenspan, J. (1984). What do novices know about programming? In A. Badre & B. Shneiderman (Eds.), Directions in human-computer interactions. Ablex Publishing Corporation Second printing. New Jersey.

Soloway, E. & Ehrlich, K. (1984). Empirical studies of programming knowledge. IEEE Transactions on software engineering, Vol. Se-10, No.5, September. 595-609.

Soloway, E. & Bonar, J. & Ehrlich, K. (1983). Cognitive strategies and looping constructs: An empirical study. Communications of the ACM, 26 (11), 854-860.

Stevens, A. & Collins, A. & Goldin, S. E. (1979). Misconceptions in student's understanding. International Journal of Man-Machine Studies, 11, 145-156.

Vessey, I. & Weber, R. (1984). Conditional statements and program coding: an experimental evaluation. International Journal of Man-Machine Studies, 21, 161-190.

Waters, R. (1979). A method for analyzing loop programs. IEEE Transactions on software., Vol. Se-5 (3). 237-247.

Woodward, M. R. & Hennell, M. A. & Hedley, D. (1979). A measure of control flow complexity in program text. IEEE Transactions on software engineering, Vol. Se-5 (1). 45-50.

Youngs, E.A. (1974). Human errors in programming. International Journal of Man-Machine Studies, 6, 361-376.