LOGO AS A MEDIUM FOR EXPLORING ATONAL THEORY

J. KENT WILLIAMS

INTRODUCTION

Logo, a widely used educational computer programming language, can serve as a medium for learning and exploring basic concepts and techniques of atonal theory and analysis. The discussion begins with a consideration of the unique characteristics of computational media as educational environments and then continues with a brief overview of the Logo language, the educational philosophy on which it is based, the attributes that distinguish it from other high-level programming languages, and possibilities for its use. Finally, I present an approach for using Logo in college-level courses that introduce atonal theory and analysis.

COMPUTER ENVIRONMENTS AS INSTRUCTIONAL MEDIA

Advocates of electronic technology have often contended that such media are merely neutral transmitters of the information that is provided to them as input. Such answers are often made in response to critics who claim that a given medium has a corrupting effect on those who use it. While it is true that instructional media can and do transmit any program that is couched in a suitable format regardless of its educational, aesthetic, or moral value, it does not follow that they are entirely neutral transmitters of that information. For, as Patricia Greenfield has noted, each medium is like a filter with its own technical and formal characteristics, and as a result, "transforms information while communicating it, emphasizing particular aspects of events and ideas, deemphasizing others. As a consequence, each medium presents certain types of information easily and well, other types with difficulty or relatively poorly."[1] Greenfield asserts that printed media excel at conveying a person's inner thoughts, audio media "make dialogue and figurative language salient," while television and films have particular strengths in presenting actions, both successive and simultaneous, in three-dimensional space. Computer technology "has a great strength in allowing

users to interact with complex systems having multiple, interacting, dynamic variables."[2]

Other theorists have focused on the microcomputer as a medium for representing and manipulating systems of symbols, and for creating products (e.g., drawings, poems, pieces of music) within those systems. Karen Sheingold notes that computers offer unique possibilities for dynamic interaction, programmability, and cognitive support.[3] Jeanne Bamberger sees the computer as a medium that "reflects back to us images of our commonsense ways of making *things* and making *sense*,"[4] one that, in Sheingold's words, "can help us discover and reflect on what we already know intuitively . . . and see familiar actions and objects in new ways."[5]

One of the foremost advocates of computer-aided problem solving and discovery learning has been Seymour Papert of MIT. Papert's book *Mindstorms: Children, Computers, and Powerful Ideas*[6] is the definitive statement of the so-called Logo philosophy of educational computing. Beginning from the Piagetian premise that learners are builders of their own knowledge structures, and that, when provided with the proper materials, they can learn without being formally taught, Papert goes on to describe how students can use computers to solve intellectual problems, explore various domains of human knowledge, and in the process, confront some of the truly powerful ideas in science, mathematics, and the art of cognitive model building.

Fundamental to Papert's vision is the idea that students should learn to control computers, rather than be controlled by them.[7] While the current generation of personal computers can be controlled in a number of different ways, the highest and most expressive degree of control is still achieved through the medium of a programming language. It follows, therefore, that those who wish to realize the computer's maximum potential as a tool for intellectual discovery should learn to write programs.

Support for Papert's position can be found in recent articles by Roger C. Shank and Robert Farrell of the Yale Artificial Intelligence Project,[8] and by Jeffrey Bonar of the Learning Research and Development Center, University of Pittsburgh. Bonar begins his article with the bold assertion that

> Within ten to twenty years all educated people in jobs with any amount of professional responsibility . . . will be programmers. That is, as part of their job they will regularly adapt and extend computational tools to meet the particular demands of their jobs.[9]

Bonar calls such people "casual programmers," as opposed to professional programmers, and clarifies this distinction as follows:

> Just as the typical literate person does not write novels, the typical computer user will not write major programs. Nonetheless, the typical user will be able to write the equivalent in programs of the typical literate person's 1 to 20 paragraph documents. These programs may not be elegant or efficient (in the narrow sense of running fast on a particular machine), but they will be an important extension of the user's intentions on the machine.[10]

CHARACTERISTICS OF LOGO

To make casual programming as accessible and rewarding as possible, Seymour Papert and others around MIT, circa 1968, developed Logo.[11] They modeled it on Lisp, a language that has particular strengths in the areas of symbol manipulation and list-processing. Most of the early work with Logo was done at MIT in the Artificial Intelligence Laboratory and the Division for Study and Research in Education.[12] The first microcomputer implementations appeared in the early 1980s and, since then, Logo has acquired a large and devoted following among teachers and learners at all levels.

Logo is most closely associated with turtle graphics, a computational medium in which users (usually children) learn basic concepts of computer programming and plane geometry by typing commands which cause a triangle-shaped cursor to move about on the display screen, leave a trail, and draw simple figures.

The acceptance of Logo at the secondary and college levels has been hampered, ironically, by its widespread success at the elementary level. The notion prevails that Logo is little more than a graphics environment for children to play in until they grow up and are ready to learn a "real language." As David Thornburg has noted,[13] this attitude is regrettable because Logo is very much a real programming language. In fact, when educational and cognitive factors are given high priority, it can reasonably be argued that Logo is among the most "powerful" of the popular high-level languages. Among its features are several that make it eminently suited as a computational medium for exploring atonal theory:

1. It is an interpreted language. This means that Logo programs can be run as soon as they are written. The

user need not go through the tedious process of compiling, linking, debugging, and executing what is endured by developers of commercial software who use compiled languages to obtain greater speed and efficiency. Logo's brief turnaround time enables a high degree of interaction between the learner and the computer which, in turn, allows for more efficient learning. Of course there is a tradeoff. Logo programs run more slowly, and they are not, as a rule, stand-alone applications. As a result, they must be run in the Logo environment.

2. Logo uses only a few data types: words (character strings), numbers (which are a special type of word), and lists.[14] Lists, which are the most appropriate data type for representing the various sets used in atonal theory, can be nested to any degree of complexity. Figure 1a shows how a pc set would be represented as a "flat" list, a list all of whose elements are either words or (in this case) numbers. Figure 1b is a nested list which contains the pentachordal subset-types of [0 1 3 4 6 7] along with their frequency of occurrence. Careful attention to the balanced pairs of square brackets will reveal that this list contains three primary elements, each of which is itself a two-element list whose first element is a five-element list (the subset-type) and whose second element is a number (the frequency of occurrence).

3. In its unextended state, Logo consists of a finite number of "primitive" procedures that can be classified as either operations or commands. Operations are procedures that compute and output a value; commands are procedures that produce an effect, usually visible (such as displaying or printing a value) or audible (such as playing music). Primitives can be invoked, either singly or in combination, from the so-called "immediate mode." A compound invocation that would compute and display the residue mod12 of the integer 20, is shown in Figure 2. In this example, REMAINDER is the operation that outputs its value to the command PRINT that displays the value on the screen.

4. Programming occurs when the Logo language is extended by combining primitives to form modules of code that are termed (user-defined) procedures. These procedures are stored in the workspace, a reserved area of the computer's memory and are invoked by their user-coined names. Users are given considerable flexibility in naming procedures and variables. The act of calling a procedure into service, of directing it to perform its task, is termed invocation. User-defined procedures may be invoked by the user from the immediate mode, by other user-defined procedures, or by themselves (see number 6 below).

5. Logo procedures can accept numbers, words, or lists of any degree of complexity as primary data (see Figure 1b), and they can pass such data freely among themselves.

6. Logo procedures can invoke clones of themselves. This programming technique, which is termed recursion, enables the definition of procedures that are both general in their applicability to a wide range of problems and elegant in their simplicity.

7. A text editor is provided to define and edit procedures. Extensive and informative error messages are also supported.

8. In most programming languages a clear distinction is drawn between the program code and the data to be processed. Logo allows this distinction to be finessed by representing its procedures as lists. This enables the definition of procedures that can, in turn, define or modify other procedures.

9. The availability of music-generating primitives in certain versions of Logo makes it possible to hear the results of some musical operations.[15]

Figure 1a. A pc set represented as a "flat" list.

```
[0 1 3 4 6 7]
```

Figure 1b. A nested list of the 5-element subset types of [0 1 3 4 6 7] and their frequency of occurrence.

```
[[[0 1 3 4 6] 3] [[0 1 3 4 7] 2][[0 1 3 6 7] 2]]
```

Figure 2. Compound invocation of two Logo primitives.

```
?PRINT (REMAINDER 20 12)
8
```

Given a computational environment with the characteristics listed above, how might it be utilized as a medium for exploring atonal theory and analysis? The range of possibilities can be subsumed under three headings used for an early collection of articles on educational computing.[16]

The computer as tutor

In this mode, an instructor would devise a series of "request procedures"[17] each of which focused on some basic analytical task (e.g., computing a certain type of interval, or transposing a pitch class). Students would use these procedures to learn how to perform the various tasks. Fully-developed procedures would judge the student's responses at all stages of the problem-solving process, and, where appropriate, provide diagnostic error messages in the manner of traditional computer-assisted instruction (CAI).

The computer as tool

In this mode, Logo procedures would be used as software tools to lighten the computational burden that is inevitably associated with atonal analysis. Such procedures could be predefined and fully operational in which case the student could be given the Logo code listings and then

instructed to define and debug the procedures. The extent to which tool procedures should be "user-friendly" is of major concern here. On the one hand, procedures that included "bullet-proofing" and error-checking routines would be easier for novices to use. On the other hand, they would be longer and more involved, and hence, more difficult to key in and understand.

The computer as tutee

This mode of usage is most consonant with the Logo philosophy as espoused by Papert and others. The student is seen as the tutor who devises instructions for the computer (the tutee) to execute. These instructions must, of course, ultimately be couched in a language that both the student and the computer understand, which in this case is assumed to be Logo. Implicit in this approach is the assumption that students' understanding of a problem is enhanced when they are required to describe in precise terms a possible solution to that problem. In stereotypical programming, these terms are assumed to be the lowest-level primitives of the language. Since Logo is extensible, however, students may be shielded from its primitives and, instead, be provided with "scaffolds," or "black boxes," previously-defined procedures that can be invoked either from the immediate mode or from higher-level, user-defined procedures. Thus, for example, rather than defining basic tool procedures (such as those that compute intervals or transpose sets) in terms of Logo's primitives, students many be provided with working versions of such procedures. After using these procedures enough to understand the tasks they perform and the data they require, students may then ascend a rung on the conceptual ladder by defining higher-order procedures in *these* terms.

The extent to which students should be required to learn the vocabulary and syntax of a programming language and, by extension, basic concepts of computer science, and to generate their programs "from scratch" are major issues in implementing this type of computational environment. In this regard, it should be noted that the Logo language was designed (and has proven) to be relatively easy to learn and use, at least in comparison with other high-level languages, and that while there is a price to be paid for doing so, the reward is greater expressive power and enhanced understanding, both of general techniques of problem solving and, usually, of the subject as well. Ultimately, the issue of whether to write programs or use ready-made applications can only be resolved by individual instructors who must take into account all the factors relevant to their teaching situation.

The approach to using Logo I will describe might best be termed pragmatic in the sense that I have incorporated certain aspects of orthodox Logo philosophy but have modified others. More specifically, I generally subscribe to the idea that programming expertise can develop more or less concurrently with one's understanding of a subject domain. Ideally, both should proceed gradually and at the student's own pace from the simple and perceptible to the more complex and abstract. This implies a preference for the "bottom-up" as opposed to the "top-down" approach to program development generally advocated by proponents of "structured programming."

The second principle is that emphasis should be placed on developing Logo procedures, and on understanding how they work, rather than on merely using them. In particular, the process of defining, testing, and debugging should be viewed as an opportunity to refine one's conception of a problem and understanding of Logo syntax, rather than as a reminder of one's intellectual limitations and lack of familiarity with the language.

Finally, students should be encouraged to recognize underlying similarities among seemingly different problems and to acquire a repertoire of techniques for solving such problems. Educational research has shown that transfer is a difficult objective to achieve in any teaching/learning environment. Papert's claim that Logo programming enhances transfer has been subjected to considerable testing that has yielded mixed results to date.[18]

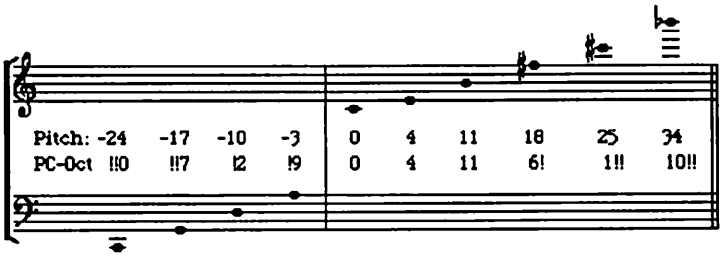
My differences with Papert are those of most teachers who have used Logo in a highly structured setting. The luxury of a free and open computing environment in which students may pursue practically any topic of their imagination and are motivated primarily by the excitement of their own discoveries is simply not feasible in my current teaching assignment. To address the need for structure and accountability, I have devised a combination text/workbook that has been used during the atonal segment of an undergraduate course in twentieth-century theory and analysis.[19] The activities I will describe are drawn from this book.

To understand the ensuing examples, it is necessary to explain a few aspects of pitch symbology. Pitches in the chromatic system (the default mode of pitch representation) are symbolized either by positive and negative integers, or by pitch-class octave symbols (see Figure 3). The default location of the reference pitch (pitch zero) is middle C, but this may be changed by invoking the KEY procedure with a pitch integer as its input. For example, the invocation KEY 3 raises the reference pitch three semitones to D#/Eb; KEY (-2) lowers it two semitones to A#/Bb.

The duration of notes and rests is also indicated with integers whose correspondence to absolute time is relative. Once a basic tempo has been established, the durational value of notes is directly proportional to the

numerical value of their duration integers (i.e., larger numbers indicate longer durations and smaller numbers indicate shorter durations).

Figure 3. Pitch representation in the chromatic system of Music Logo.



EXPLORING ATONAL THEORY

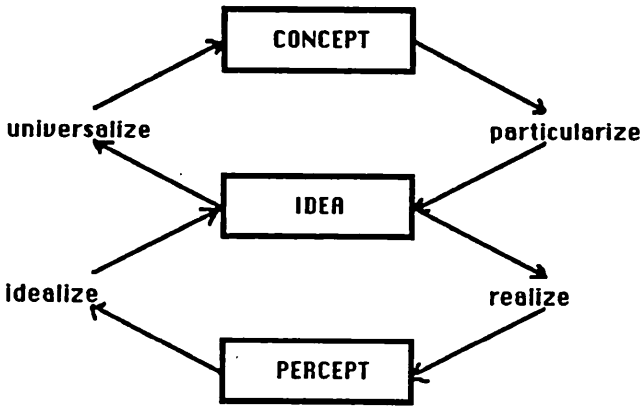
Several years ago, the English theorist, Philip Barford, wrote a perceptive account of the process of developing musical cognition.[20] I take his main point to be that the process must proceed slowly and be continually reinforced by abundant experiences with musical sound. In his discussion, Barford posits three levels of understanding: percept, idea, and concept (see Figure 4). For our purposes, a percept can be defined as a sound that can be heard, an idea is a sound that can be imagined, and a concept is the unifying principle behind a group of related ideas.

As Figure 4 illustrates, Barford also posits mental operations by which one level is transformed into the next. In going from sensory perception to mental abstraction (from bottom to top in the diagram) a percept is *idealized* to form an idea or tonal image that, in combination with other related ideas, may eventually be *universalized* to form a concept. Moving in the other direction (from top to bottom), a concept is first *particularized* as an idea that may then be realized as a perceptible musical sound.

My reason for digressing here is to present a working vocabulary for the subsequent discussion and to underscore both Barford's and Papert's concerns regarding the necessity to make haste slowly when attempting to develop theoretical concepts.

To illustrate how the task of learning Logo might be correlated with that of learning the fundamentals of atonal theory, the proposed instructional sequence will be broken down into several large units, with the implicit understanding that further subdivision is both possible and desirable. The discussion of each unit will be preceded by a statement of the objective(s) for each of the two domains.

Figure 4. Schematic diagram of Barford's model of concept development.



Unit One

Atonal Theory

Gain familiarity and competence with integer representation of pitches and durations.

Logo

Learn the syntax for invoking primitive procedures from the immediate mode.

Understand and be able to use music-generating primitives. This means knowing a procedure's name, the number of inputs that it requires, the data type of each input, and the effect that is produced upon a successful invocation.

Students can begin by invoking various music-generating primitives from the immediate mode to play short melodic lines. Figure 5 contains verbal definitions of these primitives along with typical invocations.

In addition to playing notated melodic fragments, other activities are possible. It is relatively easy for an instructor to define "request procedures" that play melodic fragments, request the student to type the corresponding pitch numbers or pc-oct symbols, and then play the student's version along with the original fragment. Students can also be given Logo code with instructions to notate the corresponding melody on the proper staff. Having done so, they may then invoke the code to compare their

notation with the heard melody. A third option is the identification of a familiar melody or the sightsinging of an unfamiliar one directly from its Logo code. (The reader is invited to identify the melodic segments that would be generated by the invocations of SING, SAMEDUR, and PLAY in Figure 5.) The main concern at this point is to provide varied opportunities for idealizing both musical sound and traditional notation as logo code, and for realizing Logo invocations in sound and score notation.

Figure 5. Invocations of music-generating primitives.

NOTE :P :D
Generates a note of pitch :P and duration :D
Example: ?NOTE "0! 8

REST :D
Generates a rest of duration :D
Example: ?REST 8

SING :PLIST
Generates a series of notes by matching the pitches of :PLIST with a predefined duration.
Example: ?SING [3 !11 !4 !7]

SAMEDUR : PLIST :D
Matches each pitch of :PLIST with duration :D
Example: ?SAMEDUR [10 9 0! 11] 12

PLAY :PLIST :DLIST
Matches the pitches of :PLIST with the corresponding durations of :DLIST
Example: ?PLAY [5! 9! 5! 10 1! 0! 9] [1 7 1 7 1 8 4]

Unit Two

Atonal Theory

Develop greater competence with integer representation of pitches and durations.

Use reference pitches other than middle C.

Logo

Learn to use the text editor for defining "tuneblock" procedures.
Learn basic concepts of modular programming
naming of procedures
syntax of tuneblocks procedures
hierarchical relationships between procedures.

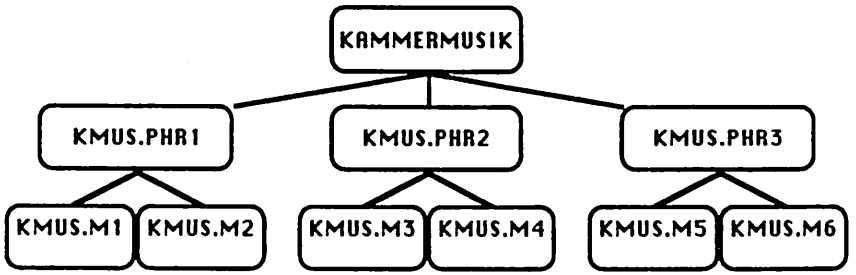
Since the possibilities for playing melodies in the immediate mode are quite limited, students should be introduced to the text editor as soon as possible. Once the requisite sub-tasks (invoking and quitting the editor, various editing keystrokes) are mastered, attention may be focused on musically rewarding activities. Several sets of "tuneblock" procedures should be defined with each set comprising the actual tuneblocks, i.e., those procedures that invoke the music-generating primitives to play a brief segment of music, along with superprocedures (procedures that invoke the tuneblocks or other procedures of a higher order).

An instructor can build upon the student's conception of hierarchical motive and phrase structure in music and use that understanding as a model. It is appropriate, therefore, at this stage to use excerpts that have clearly defined motive and phrase structure and to not be overly concerned about the purity of the atonal idiom.

Figure 6 contains a schematic diagram for a set of procedures that play the opening clarinet solo to Hindemith's *Kammermusik No. 2* along with a listing of the Logo code for the first two measure-level procedures. Once defined, any of the procedures in the diagram may be invoked singly or in combination. The most efficient way to play the entire melody is to invoke KAMMERMUSIK, the "toplevel" procedure that invokes its three phrase-level subprocedures each of which invokes its two measure-level subprocedures. The tuneblocks could be also scrambled by invoking the measure-level procedures in any desired sequence.

Additional experience can be gained by defining polyphonic tuneblocks, those that play multi-voiced passages. Figure 7 contains the definitions for a set of procedures that plays the first half of Dallapiccola's "Linee" from the *Quaderno musicale di Annalibera.*

Figure 6. Tree diagram of a set of tuneblock procedures.



(A duration integer of 1 represents a quarter note.)

```
TO KMUS.M1
   PLAY [4! 4! 4!] [.5 .25 .25]
   SAMEDUR [11! 8! 3! 0!] .25
   PLAY [10 10 0!] [.5 .25 .25]
   SAMEDUR [8 3] .5
 END
```

```
TO KMUS.M2
   SAMEDUR [4 6 4 6] .25
   SAMEDUR [7 2! 7! 6!] .25
   PLAY [5! 5!6!] [.5 .25 .25]
   NOTE "8! 1
 END
```

Figure 7. Set of Logo procedures to play A section of Dallapicolla's "Linee"

(A duration integer of 1 represents an eighth note.)

```
To LINEE.A.TREBLE
  REPEAT 4 [SAMEDUR [9 10] 1]
  REPEAT 4 [SAMEDUR [2! 5] 1]
  REPEAT 4 [SAMEDUR [7 1!] 1]
  REPEAT 3 [SAMEDUR [0! 4] 1]
  PLAY [0! 4] [1 4.5]
  REST 1.5
END

To LINEE.A.BASS
  REST 8
  SAMEDUR [8 7 3 0!] 4
  NOTE 10 8
  REST 5
END
```

```
To LINEE.A
  LINEE.A.TREBLE
  BEGINVOICE 0
  LINEE.A.BASS
END
```

Unit Three

Atonal Theory

Introduction to the concept of pitch class.
Learn algorithms for basic pitch and pc operations
computing interval
Transposition (Tn)
Inversion (I)
Retrograde and Retrograde-Inversion (R and RI)
Comparison of sets for Tn, I, and R relatedness
Compound operations (TnI, TnR, TnRI, etc.)
Computing prime-by-inversion matrices

Logo

Further experience with the "bottom-up" approach to procedure development.
Introduction to
input variables
basic list processing primitives
tail recursion.

Once students are familiar with the system of pitch and pitch class representation, the music-generating primitives, and use of the Logo text editor, they can begin to develop procedures that perform basic set-theoretical operations. These would include computation of the various types of intervals and transposition and/or inversion of sets of pitches or pitch classes.

Rahn's formula for an ordered pitch interval and its realization as a Logo procedure named OPI are shown in Figure 8a.[21] The Music Logo primitive DECODE is invoked as a subprocedure from within OPI to insure that both of the pc-oct symbols are converted to pitch numbers before subtraction occurs. Once such a procedure has been debugged and is operational, it can be invoked from the immediate mode whenever the user

needs to have its task performed. A typical invocation is shown below the definition.

OPI can serve as a model for other interval-computing procedures. UOPI, a procedure that computes the unordered interval between two pitches is shown in Figure 8b. The only new feature here is the invocation of ABSVAL, a user-defined subprocedure that returns the absolute value of the number given as its input.

As students begin to work with pitch classes, they will realize the need for a MOD12 procedure such as that shown in Figure 8c. MOD12 provides the opportunity to introduce REMAINDER, a Logo primitive that returns the remainder of the integer division of its first input by its second input. MOD12 can also serve as an introduction to recursion, a programming technique that will prove increasingly useful at later stages. The availability of MOD12 enables the definition of OPCI (Figure 8c), a procedure that computes the ordered interval between two pitch classes.

The only remaining interval type (in Rahn's scheme) is the unordered pc interval, or interval class, the formula and procedures for which are shown in Figure 8d. Here the added requirement is for LESSER.OF, a subprocedure that compares two numbers (here, complementary ordered pc intervals) and outputs the lesser of these two values.

Figure 8. Procedures to compute intervals (after Rahn)

a. ordered pitch interval: ip<x,y> = y - x

```
TO OPI :X :Y
  OUTPUT ( DECODE :Y ) - ( DECODE :X )
END

?OPI 0 4
RESULT: 4
```

b. unordered pitch interval: ip(x,y) = | y - x |

```
TO UOPI :X :Y
  OUTPUT ABSVAL ( OPI :X :Y )
END

TO ABSVAL :N
  IF :N < 0 OUTPUT - :N
  OUTPUT :N
END
```

c. ordered pc interval: I <a, b> = (b - a) mod 12

```
TO OPCI :A :B
  OUTPUT MOD 12 ( :B - :A )
END

TO MOD12 :N
  IF :N > -1 OUTPUT REMAINDER :N 12
  OUTPUT MOD12 ( :N + 12 )
END
```

d. unordered pc interval:
i(a,b) = smaller of i<a,b> and i<b,a>

```
TO UOPCI :A :B
  OUTPUT LESSER.OF ( OPCI :A :B ) ( OPCI :B :A)
END

TO LESSER.OF :X :Y
  IF :X < :Y OUTPUT :X
  OUTPUT :Y
END
```

Figure 9 lists another set of basic tools, those that transpose and/or invert a single pitch or pitch class.

Figure 9. Procedures for transposition and/or inversion of pitches and pitch classes.

Pitch transposition

```
TO P.T :N :P
  OUTPUT ( :P + :N )
END
```

Pitch class transposition

```
TO PC.T :N :PC
  OUTPUT MOD12 ( :PC + :N )
END
```

Pitch inversion

```
TO P.I :P
  OUTPUT ( - :P )
END
```

Pitch class inversion

```
TO PC.I :N :PC
  OUTPUT MOD12 ( - :PC )
END
```

Pitch inversion followed by transposition

```
TO P.TI :N :P
  OUTPUT P.T :N (P.I :P)
END
```

Pitch class inversion followed by transposition

```
TO PC.TI :N :PC
  OUTPUT PC.T :N (PC.I :PC)
END
```

With these tools at hand, it now becomes possible to define a class of superprocedures each of which operates on a list of elements. At this point one enters the realm of list processing, the heart of the Logo language.

Most of Logo's list processing primitives fall into either of two categories: Figure 10a shows invocations and results of four procedures that reference specific elements of a list. Figure 10b shows another set of four procedures that construct new lists.

Figure 10. Invocations of Logo list-processing primitives.

a. Procedures which reference elements of a list

```
?FIRST [0 1 4 6]          ?LAST [0 1 4 6]
Result: 0                 Result: 6

?BUTFIRST [0 1 4 6]       ?BUTLAST [0 1 4 6]
Result: 1 4 6             Result: 0 1 4
```

b. Procedures which construct new lists

```
?SENTENCE 2 [4 7 10]      ?LIST [2 3] [4 7 10]
Result: [2 4 7 10]        Result: [[2 3] [4 7 10]]

?FPUT 3 [4 5 9]           ?LPUT 3 [4 5 9]
Result: [3 4 5 9]         Result: [4 5 9 3]
```

Nearly all user-defined procedures that invoke list processing primitives to operate upon lists utilize recursion. Recursion can be defined as a technique of solving a problem by repeatedly reducing it to simpler versions of the same problem until the solution becomes trivial. To illustrate, let us consider a typical operation in atonal theory, that of transposing a set of pitch classes. As Figure 11 shows, the problem of transposing the set [0 1 4] can be reduced to the problem of transposing pc 0 and concatenating the result with the transposition of the set [1 4]. This problem can, in turn, be

reduced to transposing pc 1 and concatenating the result with the transposition of the one-element set [4], which, in turn, can be reduced to the problem of transposing pc 4 and concatenating the result with the transposition of the empty set. The strategy, then, is to invoke a list referencing primitive to obtain an element of the input list (typically the first element), perform the requisite operation upon that element, and then invoke a list-constructing primitive to add the result to a new output list.

In the procedure shown in Figure 11, the list constructing primitive SE (abbreviation for SENTENCE) is invoked to add the newly-transposed pc to a new list. SE takes a default number of two inputs. If both inputs are literally present, it combines them to form a "flat" list. Here, however, the second input to SE is not a literal list but an invocation of a clone of PCS.T, the procedure that will output the next transposed pc. That procedure, will, in turn, invoke a clone of itself, which will invoke a clone of itself, etc. When the original input list is finally exhausted, the stop condition will have been met and the current invocation will output the empty list as directed. Each prior invocation of PCS.T will then output its transposed pc, placing it at the head of the new list. Since the procedures will output their values in reverse order of their invocation, the list of transposed pcs will be constructed from end to beginning.

Understanding a tail-recursive procedure that outputs a list is probably the most difficult task for the student of Logo at the intermediate level. The Logo community recognizes this fact and provides metaphors, games, and other teaching aids to help students overcome this conceptual hurdle.[22] As with any other task, some students get the hang of recursion before others. My experience has been that with the help of the TRACE function nearly all students understand the process within a few days.

The culmination of this unit should be a series of projects in composition and/or analysis in which students use their new software tools to do something that is both musically interesting and analytically substantial. An appropriate project would be the modeling of a brief musical work, that is, the definition of a set of Logo procedures that describe the work in terms of a few elements and operations upon those elements and which, when invoked, generate and play the work.

Generality, the extent to which a computer program can be utilized for various instances of a given class of problems, is a major concern of computer programmers. The ideal objective is to isolate a process (the task to be performed) from the data (the information to be processed) by defining the task in terms sufficiently general to enable the processing of any valid set of data.

Figure 11. Recursive invocations in the transposition of a pc set.

(OUTPUT is abbreviated as OP in this and succeeding examples.)
(BUTFIRST is abbreviated as BF in this and succeeding examples.)

```
PCS.T 6 [0 1 4]
  IF (EMPTY? :SET) OP []
  OP FPUT (PC.T 6 (FIRST :SET)) (PCS.T 6 (BF :SET))

    PCS.T 6 [1 4]
      IF (EMPTY? :SET) OP []
      OP FPUT (PC.T 6 (FIRST :SET)) (PCS.T 6 (BF :SET))

        PCS.T 6 [4]
          IF (EMPTY? :SET) OP []
          OP FPUT (PC.T 6 (FIRST :SET)) (PCS.T 6 (BF :SET))

            PCS.T 6 []
              IF (EMPTY? :SET) OP []

              outputs []

          outputs [10]

      outputs [7 10]

outputs [6 7 10]
```

Music students often have difficulty distinguishing between the basic pitch materials of a work and the process(es) to which those materials are subjected. They can begin to appreciate this distinction by redefining their tuneblock procedures more generally.

The extent to which the pitch materials of a work determine its essential nature is a major concern of music theorists and the effect of varying these materials can be dramatic in the case of a twelve-tone work. Consider, for example, Dallapiccola's "Fregi" (No. 6 from *Quaderno musical di Annalibera*), a work nearly all of whose pitches can be derived by various operations upon the 12-ps set that unifies the entire opus. The task of modeling this piece in sets of Logo procedures might be broken down into three stages.

The first stage would be the definition of a set of procedures that play the entire melodic line when given the first twelve pitches as input. Since all of the realization of the basic 12-pc set are related by the pitch (as apposed to pc) operations, this is a relatively easy task. The student uses OPI and OPI.PAT, the ordered pitch interval and interval-pattern computing procedures, to discover the operation and the interval of transposition by which each of the succeeding melodic phrases can be derived from the first phrase[23] then utilizes that knowledge in the definition of the tuneblock procedures. Figure 12 lists a set of Logo statements that define this process. Note that the durational and pitch patterns are assigned to variable names in the FREGI.SETUP procedure and then invoked by those names in the tuneblock procedures.

Figure 12. Set of procedures to generate melody of "Fregi"

```
TO FREGI.SETUP
  KEY (-2)
  MAKE "RPAT1 [1.5 2 2 1.5 1.5 3 1 1 1 1.33 1.33 1.33]
  MAKE "RPAT [1.33 1.33 1.33 .75 .75 .75 .75 1.33 1.33 1.33 .66 3.33]
  MAKE 'FREGI.PSET [0! 1 !5 !8 !10 !4 !3 !7 !!9 !2 !11 6]
END

TO FREGI.MELODY
  FREGI.SETUP
  FREGI.A1
  FREGI.A2
  FREGI.B1
  FREGI.B2
END

TO FREGI.A1
  PLAY :FREGI.PSET :RPAT1
END

TO FREGI.A2
  PLAY (PS.TR 7 :FREGI.PSET) :RPAT2
END

TO FREGI.B11
  PLAY (PS.TI:FREGI.PSET) :RPAT1
END
```

```
TO FREGI.B2
  PLAY (PS.TRI (-6) :FREGI.PSET) :RPAT2
END
```

The second stage might involve the definition of a procedure that would take two inputs: an ordered set of twelve pcs, and a list of octave numbers. The student could invoke the procedure with Dallapiccola's pcs and octave numbers to generate the melody of "Fregi," but could also invoke the procedure with other pcs or octave numbers to generate pieces that retained "Fregi's" processes, but varied its pitch or pc content.

The third and final task would be to define a set of procedures that would generate the notes of the dyadic accompaniment of "Fregi." This could lead to the development of a new variant of the PS.TI procedure, the one that inverts a list and transposes a list of pitches.

Unit Four

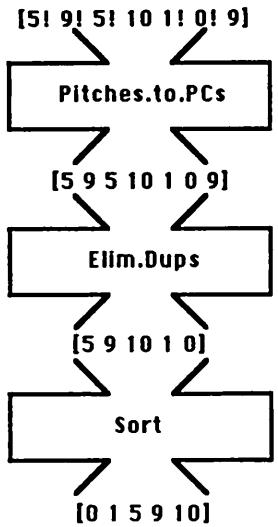
Atonal Theory
- Unordered pc sets
- Normal form
- Set types.
- Recognize and validate T and TnI relations
- Compute and compare ic vectors.

Logo
- Further development of analysis tool procedures
- Use of "black box" procedures where appropriate.
- Development and use of programs that solve complex problems.
- Use of discovery learning to refine theoretical concepts.

The step into the realm of unordered pc sets is, as David Beach has noted,[24] one of the giant steps in the study of atonal theory. For although the notion of unordered pc sets is more general and conceptually powerful than that of ordered sets, it is also considerably more abstract. In most cases, it is more difficult to particularlize and realize one's way from an unordered pc set or set-type back to real music because there are so many possible realizations. Although a Logo-based approach offers no panacea for this dilemma, it can offer general strategies for coping with it.

First, the instructor and student should attempt to embody all stages of the idealizing and generalizing processes (see Figure 4) in Logo procedures so the route from sensory perception to cognitive abstraction may be traced. This should include the definition of a set of procedures that convert an ordered list of pitches into an unordered pc set. A "plumbing diagram" for such a program is shown in Figure 13.[25]

Figure 13. "Plumbing" diagram of a program to convert an ordered list of pitches or pc/oct symbols to an unordered pc set.



Second, whenever possible, procedures that embody the particularizing processes (see Figure 4) should also be developed so that students might see and understand how abstract entities such as unordered pc sets can be transformed into the pitches and durations of real music. The feasibility of doing this, would ultimately depend, of course, on the nature of the musical specimen under analysis and the degree of abstraction involved.

Underlying the above considerations is the even more fundamental need to use the computer and Logo in ways that enhance understanding of the analytical process. It may very well be, at this stage, that the development of programming skills will begin to lag behind the students' growing competence in atonal theory. As a result, the instructor may need to provide more help in defining procedures, or simply give the students segments of Logo code with the understanding that such procedures are to be treated,

for the present, as "black boxes." Whatever policy is adopted, understanding is more likely to evolve if programs are developed gradually with adequate time devoted to comprehending at least the task and data requirements (if not the internal workings) of each. Merely getting the right answer or computing every possible characteristic of an object under analysis should never be stressed at the introductory level.

In cases where comparisons and judgements must be made, it is often better to develop a network of procedures that compute the necessary data but leave the actual decision up to the student. When the problem has been solved repeatedly, and the criteria for making the final decision are well understood, the judgment process can then be encoded and superimposed upon the original program.

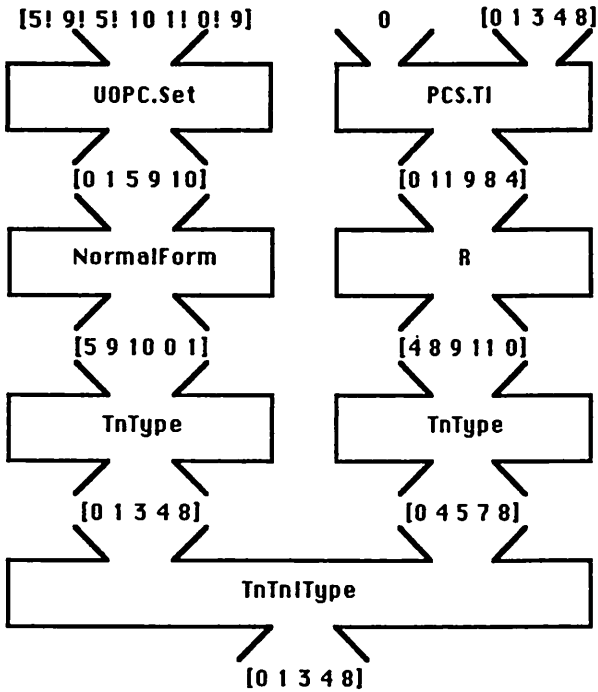
One instance of this approach would be the development of a network of procedures that determine the set-type of a list of pitches. Figure 14 illustrates how such a network can be built by assembling smaller modules. The module named UOPC.Set at the top of the left column would embody the entire process of deriving an unordered pc set from a list of pitches; that is, all of Figure 13 would be nested within this module. The final stage, a procedure named Tn. TnIType, which accepts the Tn-type and the TnIType as inputs and determines which is the set type, would be added to the network only after students had grown familiar with the functions of the other procedures and had successfully chosen the set-type several times themselves.

Unit Five

Atonal Theory
- Advanced analysis projects
- Generative composition of pieces using variables
- Theoretical investigation
  - Set union, intersection, complementation
  - Compute and compile a table of set-types
  - Inclusion relations

Logo
- Develop additional tool procedures
- Develop capacity for independent and creative theoretical and analytical work

Figure 14. "Plumbing" diagram of program to determine the set type of a list of pitches.



As students become more comfortable with the computer, with Logo, and with the basic concepts and operations of atonal theory and analysis, they can pursue theoretical, as opposed to compositional, or analytical work. Having become gradually aware of the universe of pc sets, they can embark on expeditions into that unfamiliar realm. Although this type of activity is the most abstract, it can also be the most intellectually stimulating.

One large-scale project appropriate for this stage is the construction of a table of set-types. Instead of being given such a table, or told to consult one in a standard text, class members could develop one of their own. In the process of doing so, they would not only learn a great deal about pc sets and their relationships, but would also experience some of the excitement of doing ostensibly original research. An approach to such a project is outlined in Figure 15. Obviously, the work would have to be divided up among the various class members and done over a considerable period of time.

Inclusion relations, the microworld of subsets and supersets, is another interesting and pertinent topic. In fact, a model for the concept of inclusion is already present in the nested structure of Logo's subprocedures and superprocedures.

Figure 15. Outline of a class project in compiling a table of set-types.

1. Have each class member devise a strategy for determining the twelve trichordal set-types.

2. Compare the various strategies and select the most efficient. (As an alternative, students could be given a list of "raw sets" and instructed to determine the set-type of each and then tabulate the results.)

3. Utilize the same method to determine the set-types for 4 and 5-element sets.

4. Study the concept of complementation at both the literal and abstract levels.

5. Use complementation to determine the set-types of 9, 8, and 7-element sets. This would be done by finding the set-type of the literal complement of each 3, 4, and 5-element set in the table.

6. Review the concept of unordered pc interval (interval class). Computer the ic vectors of the set-types in the table. Compare the vectors of set-types of the same cardinality and of complementary set-types. Use the ic vector as a means of determining the degrees of symmetry of each type.

7. Apply knowledge gained from the above activities to determine the set-types of hexachords and to investigate their properties.

The first task might be the development of a procedure that accepts a set of cardinality C and outputs the subsets having C - 1 elements. Doing so involves processing the original set C times and excluding a different pc each time. This indicates a need for a new list-referencing "primitive," one that returns all of the elements of the set given as its second input save the one specified as its first input. Figure 16a shows some typical invocations

for a procedure named AB (for ALLBUT) that enables the definition and invocation of SUBSETS (see Figure 16b), a procedure that computes the C - 1 subsets of an unordered pc set.

Figure 16. Tool procedures for computing subsets.

a. Invocations of ALLBUT

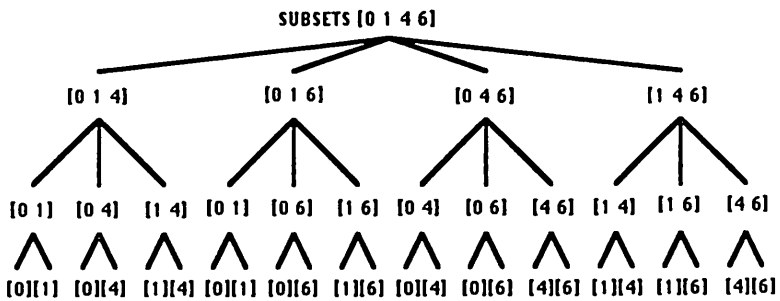
```
?AB 0 [0 1 4 ]          ?AB 1 [0 1 4]          ?AB 4 [0 1 4]
[1 4]                   [0 4]                  [0 1]
```

b. Invocation of SUBSETS

```
?SUBSETS [0 1 4]
[[0 1][0 4][1 4]]
```

The next task, that of obtaining the smaller subsets, is particularly interesting from a problem-solving point of view. It would seem the problem could be solved recursively; that is, if one can compute the C - 1 subsets of a set, then the C - 2 subsets would be subsets of those, the C - 3 subsets would be subsets of those, etc. It should be possible, then, to define a recursive procedure that computes something like the family tree of a set. A tree diagram of the output from such a procedure is shown in Figure 17. Careful inspection of the subsets listed in the lower nodes reveals a flaw in this strategy. For although each subset can indeed be derived from its parent set, there is an increasing number of duplicates at successively lower levels.

Figure 17. Tree diagram of output from recursive invocations of SUBSETS.



The presence of this bug compels one to rethink the problem. Upon closer examination, it becomes clear that we are really interested at any given level in finding the *set* of subsets of cardinality X, where X is less than C and greater than zero. Therefore, in those cases where one is computing subsets from a list of subsets (e.g., in computing the 2-element subsets of a tetrachord from its 3-element subsets), it will be necessary to check for duplicates before adding a newly-computed subset to the list.

SUMMARY AND CONCLUSIONS

The underlying theme for these activities is that computers and programming languages can be valuable tools for learning the fundamentals of atonal theory, but that they can and should be used in ways that foster understanding of the musical data, analytical processes, and theoretical concepts of that domain. My experience has led me to believe that this objective can be achieved through the medium of an educational programming language, one that provides appropriate data types, encourages modular and hierarchical thinking, and is relatively easy to learn and use.

The broadest definition I know for the verb "to program" is "to communicate with a computer in a language that both the user and the computer understand." Under this definition, it is possible to program computers on a number of different levels. Logo offers the novice or casual programmer the possibility of building a multi-leveled model of an intellectual problem and of communicating with the computer at any of those levels. This feature makes it possible for teachers to design computational environments in which students can address the computer at their own level of expertise.

It would appear that, as computer usage permeates elementary and secondary education, more and more students will be entering music schools having learned the fundamentals of "computer literacy" and programming. It should be possible, therefore, to build upon that foundation and use it as a platform for nurturing the development of music-theoretical knowledge and thought processes. My experience with Logo leads me to believe that it deserves serious consideration as the most appropriate language for this task. I encourage instructors of atonal theory to explore the language and its literature, and to develop professional contacts with the Logo community.

NOTES

[1]Patricia Greenfield, "Educational Technologies, Education, and Cognitive Development," in *Applications of Cognitive Psychology*, ed. Dale E. Berger, Kathy Pedzek, and William P. Banks (Hillsdale, NJ: Lawrence Erlbaum Associates, 1987): 18.

[2]Greenfield, "Educational Technologies," p. 19.

[3]"The Microcomputer as a Symbolic Medium," in *Mirrors of Minds*, ed. Roy D. Pea and Karen Sheingold, (Norwood, NJ: Ablex, 1987): 202-203.

[4]*The Computer as Sandcastle*, (Technical Report No. 20, New York: Bank Street College of Education, 1983):1.

[5]Sheingold, "The Microcomputer as a Symbolic Medium," p. 204.

[6]New York, Basic Books, 1980.

[7]For a critique of Papert's *Mindstorms* see John Davy, "Mindstorms in the Lamplight," in *The Computer in Education: A Critical Perspective.*, Douglas Sloan, ed., (New York: Teachers College Press, 1984).

[8]"Creativity in Education: A Standard for Computer-Based Teaching," *Machine-Mediated Learning* 2 (1988): 175-194.

[9]Jeffrey Bonar, "Everyone Will Be a Programmer," *Technology and Learning* 1 (May/June 1987): 1.

[10]Ibid.

[11]Details of the development process are related in Robert W. Lawler, "Learning Environments: Now, Then, and Someday," in *Artificial Intelligence in Education, Volume One: Learning Environments and Tutoring Systems*, ed. Robert W. Lawler and Masoud Yazdani, (Norwood, NJ: Ablex, 1987): 10-12. See also Joel E. Bass, "The Root's Logo's Educational Theory: An Analysis," *Computers in the Schools* 2 (1985): 107-116.

[12]For a summary of this work, see the *Logo 86 Bibliography* (Cambridge, MA: Massachusetts Institute of Technology, 1986), Section 4.

[13]See David D. Thornburg. "Logo: A Computer Language at the Crossroads," *A+* 4/3 (March 1986): 78-84.

[14]More sophisticated versions of Logo also support objects, arrays, and (in the case of the Macintosh) data types such as cursors, menu bars, and regions of the display screen.

[15]*Object Logo for the Apple Macintosh* (Paradigm Software Inc., P.O. Box 2995, Cambridge, MA) provides low-level primitives for music synthesis and MIDI support. *LogoMusic* (for IBM-PC and Macintosh systems with MIDI keyboards) has been developed by Jeanne Bamberger of MIT and is used extensively in music courses and workshops there.

[16]Robert Taylor, ed., *The Computer in the School: Tutor, Tool, Tutee.* (New York: Teachers College Press, 1980).

[17]Logo request procedures are programs that pause at certain points during their execution to accept input from the user.

[18]See Gavriel Salomon and D. N. Perkins, "Transfer of cognitive skills from programming: When and how?." *Journal of Educational Computing Research* 3/2 (1987): 149-169; Gavriel Salomon, "AI in Reverse: Computer Tools That Become Cognitive." Paper presented at the Annual Meeting of the American Educational Research Association, April 5-9, 1988. ERIC Report ED 295610; Roy D. Pea and D. Midian Kurland. "On the Cognitive Effects of Learning Computer Programming," *New Ideas in Psychology* 2/2 (1984): 137-168; and various authors, "The Cognitive Effects of Computer Learning Environments," *Journal of Educational Computing Research* 2/4 and 3/1 (1986).

[19]*Exploring Atonal Theory with Music Logo*, Greensboro, NC, 1987. By the author.

[20]See "Preface to the Study of Music Theory," *Music Review* 33 (1972): 24-28.

[21]See John Rahn, *Basic Atonal Theory* (New York and London: Longman Inc., 1980): 20-31.

[22]Patrick C. Less and Margaret A. Mitchell, "Demystifying Logo Recursion: A Storage Process Model of Embedded Recursion," *Computers in the Schools* 2/2-3 (Summer/Fall 1985): 197-208; Max R. Martin, "Recursion—A Powerful But Often Difficult Idea," Ibid., pp. 209-217; and Brian Harvey, *Computer Science Logo Style*, Vol. 1 Intermediate Programming (Cambridge, MA: The MIT Press, 1985), Chapter 5.

[23]Before this can be done, the student must discover the relationship between the interval patterns of ordered sets and the serial operations by which such sets can be derived from one another. To be more specific, it would seem, intuitively, that two sets which exhibit R-related interval patterns could be mapped onto each other

discovery of this counter-intuitive relationship, students should then attempt to explain it.

[24]See David Beach, "Pitch Structure and the Analytic Process in Atonal Music: An Interpretation of the Theory of Sets," *Music Theory Spectrum* 1 (1979): 8.

[25]These "plumbing" diagrams are modeled upon others found in Brian Harvey's *Computer Science Logo Style.*

REFERENCES

Bamberger, Jeanne. *The Computer as Sandcastle.* New York: Bank Street College of Education, 1960. Technical Report No. 20.

Barford, Philip. "Preface to the Study of Music Theory." *Music Review* 33 (1972): 22-33.

Bass, Joel E. "The Roots of Logo's Educational Theory: An Analysis." *Computers in the Schools* 2/2-3 (Summer/Fall 1985): 107-116.

Beach, David. "Pitch Structure and the Analytic Process in Atonal Music: An Interpretation of the Theory of Sets." *Music Theory Spectrum* 1 (1979): 7-22.

Beckwith, Sterling. "Talking Music With a Machine." *College Music Symposium* 15 (Spring 1975): 94-99.

Bonar, Jeffrey. "Everyone Will Be a Programmer." *Technology and Learning* 1 (May/June 1987): 1-3; 5.

Burke, Michael P. and Roland, Genise L. *Logo and Models of Computation.* Menlo Park, CA: Addison Wesley, 1987.

diSessa, Andrea. "Artificial Worlds and Real Experience." *Artificial Intelligence and Education, Volume One: Learning Environments and Tutoring Systems.* Edited by Robert W. Lawler and Masoud Yazdani. Norwood, NJ: Ablex, 1987.

Friendly, Michael. *Advanced Logo: A Language for Learning.* Hillsdale, NJ: Lawrence Erlbaum Associates, 1985.

Goldenberg, E. Paul and Feurzeig, Wallace. *Exploring Language with Logo.* Cambridge, MA: MIT Press, 1987.

Greenberg, Gary. "Becoming a Knowledge Engineer: A Musician's Approach." *Academic Computing* 1/1 (1987): 42-44; 70.

Greenfield, Patricia M. "Electronic Technologies, Education, and Cognitive Development." *Applications of Cognitive Psychology: Problem Solving, Education, and Computing*. Edited by Dale Berger, Kathy Pedzek, and William P. Banks, Hillsdale, NJ: Lawrence Erlbaum Associates, 1987.

Harvey, Brian. *Computer Science Logo Style*. 3 vols. Cambridge, MA: MIT Press, 1985, 1986, 1987.

Lawler, Robert. "Learning Environments: Now, Then, and Someday." *Artificial Intelligence in Education , Volume One: Learning Environments and Tutoring Systems*, pp. 1-25. Edited by Robert W. Lawler and Masoud Yazdani. Norwood NJ: Ablex, 1987.

Less, Patrick C., and Mitchell, Margaret A. "Demystifying Logo Recursion: A Storage Process Model of Embedded Recursion." *Computers in the Schools* 2/2-3 (Summer/Fall 1985): 197-208.

Lough, Tom, ed. *LOGO 86 Bibliography*. Cambridge, MA: MIT, 1986.

Martin, Max R. "Recursion—A Powerful But Often Difficult Idea." *Computers in the Schools* 2/2-3 (Summer/Fall 1985): 209-217.

Masterson, Fred A. "Evaluating Logo: A Case Study in Requirements for Student Programming Languages." *Computers in the Schools* 2/2-3 (Summer/Fall 1985): 179-196.

McArthur, David. "Developing Computer Tools to Support Performing and Learning Complex Cognitive Skills." *Applications of Cognitive Psychology: Problem Solvng, Education, and Computing*. Edited by Dale Berger, Kathy Pedzek, and William P. Banks, Hillsdale, NJ: Lawrence Erlbaum Associates, 1987.

Papert, Seymour. *Mindstorms: Children, Computers, and Powerful Ideas*. New York: Basic Books, 1980.

Pea, Roy D., and Kurland, D. Midian. "On the Cognitive Effects of Learning Computer Programming." *New Ideas in Psychology* 2/2 (1984): 137-168.

Pea, Roy D. "Integrating Human and Computer Intelligence." *Mirrors of Minds: Patterns of Experience in Educational Computing*, pp. 128-146. Edited by Roy

Proceedings of the MIT Logo Conferences. Cambridge, MA: MIT, 1984-1986.

Rahn, John. *Basic Atonal Theory.* New York and London: Longman Inc., 1980.

Sheingold, Karen. "The Microcomputer as a Symbolic Medium." *Mirrors of Minds: Patterns of Experience in Educational Computing.* Edited by Roy D. Pea and Karen Sheingold, Norwood, NJ: Ablex, 1987, pp. 198-208.

Salomon, Gavriel. "AI in Reverse: Computer Tools That Become Cognitive." Paper presented an the Annual Meeting of the American Educational Research Association. April 5-9, 1988. ERIC Report ED 295 610.

Salomon, Gavriel and Perkins, D. N. "Transfer of Cognitive Skills From Programming: When and How?" *Journal of Educational Computing Research* 3/2 (1987): 149-169.

Schank, Roger C., and Farrell, Robert. "Creativity in Education: A Standard for Computer-Based Teaching." *Machine-Mediated Learning* 2 (1988): 175-194.

Thornburg, David. "Logo: A Computer Language at the Crossroads." A + 4/3 (March 1986): 78-84.

Williams, J. Kent. *Exploring Atonal Theory With Music Logo.* Greensboro, NC: By the author.

_____. "Talking Music Theory With a Computer and a Human Being." *Journal of Computer-Based Instruction* 16/2 (Spring 1989): 71-79.

Various authors. "The Cognitive Effects of Computer Learning Environments." *Journal of Educational Computing Research* 2/4 and 3/1 (1986).