
Toward a Culture of Creativity: A Personal Perspective on Logo's Early Years, Legacy, and Ongoing Potential

Wallace Feurzeig, *feurzeig@bbn.com*

Principal Scientist, Learning Systems, BBNT, Cambridge, Massachusetts, USA

Abstract

Logo was born in 1966, forty-one years ago this summer. It was designed very specifically to be a powerfully expressive yet readily accessible language for construction, exploration, and investigation of ideas and processes in math, science, language, and music—to give children a lively learning environment. This was a time of great expectations in the United States. Following the first moon landing, American children everywhere were counting backwards from 10 to zero, closing with a triumphant “Blastoff!” Everything seemed possible. Schools, sometimes among the most change-resistant of institutions, were often open to project-based inquiry, in marked contrast with older, lock-step instruction.

Computers are now nearly ubiquitous in schools throughout the U.S. They are used extensively for word processing and information retrieval. Instructional applications abound, often enhanced by visually rich graphics and animation. Because of the dramatic rate of development and application of computers, one might have predicted that the new learning experiences made possible by programming ideas and activities would be well-established throughout schools by now. But sadly, facilities for student design and invention are severely limited or absent. The use of high-level programming languages in schools, particularly during children's formative years, has almost vanished. Their powerful potential as expressive tools for knowledge construction has yet to be realized.

Education should aim to help children succeed in tasks that they have to work at, that take time, and that require a significant investment of thinking. Such learning experiences also teach children how to make a serious commitment to a task or subject area. Throughout its history, the Logo movement has played a significant role in fostering these goals, and it continues to do so, both within schools and outside. Logo may have gone underground in U.S. schools. But the influence of the ideas and philosophy underlying Logo remains powerful and pervasive, in the world and even in my country. I envisage a process of cultural change that will eventually produce a critical mass of young people who are comfortable and competent with the richness and variety of learning experiences made possible by constructive tools like Logo. These men and women will become, in turn, the creators and teachers of the next wave of learning technology.

Introduction

Thanks to Ivan, Erich, and Jenny for inviting me to come and giving me the opportunity to revisit my many friends here. I am delighted to be here. I look forward to pleasant interactions with you during the week.

Let me begin with the suggestion that we dedicate this conference to Seymour Papert and that we join in expressing our hope for Seymour's full recovery and continued inspiration

In the Beginning

The first document on Logo was published in August 1967, exactly forty years ago, as a technical memorandum, titled "The Logo System: Preliminary Manual." The newborn language was created during a few dramatic months from the summer through the fall of 1966.

I'll begin with some personal history leading up to my role in that momentous event. From 1948 through 1950 I was an undergraduate in Mathematics at the University of Chicago, working part-time in biological computation on electric calculators at the Argonne National Laboratory. The Laboratory was established at the University by the great physicist Enrico Fermi as a follow-on of the Manhattan Project, to continue scientific research in nuclear physics. I had just read the groundbreaking paper by John von Neumann on the concepts underlying the design and construction of electronic digital computers. I was stunned by the realization that numbers, when directed to a processing engine, could come alive and do things—the fundamental equivalence of data and programs. It was an awe-inspiring, truly beautiful idea. I had an epiphany—a transcendent experience with an emotional impact comparable to being struck by the discovery of Mozart and Shakespeare. I could think of nothing else for days afterwards. That paper made me fall in love with computers.

Then a wonderful thing happened. I heard about a recently formed mathematics group at the laboratory, dedicated to designing, building, and programming an electronic digital computer for scientific computation. At this time there were very few computers in the world. The Eniac, the first electronic computer, was built in 1945. The AVIDAC, the eleventh electronic digital computer, counting from the Eniac, was built in 1952. I joined the AVIDAC group in 1951 as a programmer. I programmed AVIDAC and other first-generation machines on problems in several areas of science and math: simulation of neutron processes, reactor design, polynomial approximation, Monte-Carlo methods, and numerical analysis.

This was before the invention of magnetic core memory. Machines at that time had cathode ray tube (Williams Tube) memories with capacities of 1024 (40-bit) words! So storage efficiency was very important. One notable space-saving development was the invention of subroutines by a programmer colleague, Jean Hall. Another colleague, George Evans, developed the first magnetic tape memory. This was before the first software languages—there were no compilers and no assembly languages. Programs were input in octal at the computer console. However, a programmer had full control of the computer. (This was before the era of batch processing, where programs on punched cards were written off-line and submitted to a bulk job-management system that delivered the result of the current run to the programmer hours or days later! Such lack of direct contact and control was anathema to one who enjoyed immediate symbiotic interaction with the machine.)

In 1953 I left Argonne to join a research institute of the University, where my charge was to build the first on-campus computer group and to direct work in the areas of simulation programming, operations research computing, and compiler implementation methods. (A major project of our institute was the design and implementation of the first large-scale strategic simulation program for the Strategic Air Command. The purpose was to develop strategies to counter a bomber and ICBM strike by the Soviet Union in the event that they initiated an attack on the United States.) I installed the first computer on the campus in the mathematics building, Eckhart Hall, in the two-

story room where Millikan in 1909 did the classic oil-drop experiment to measure the electric charge of the electron.

While there, I became involved in the design and specification of high-level programming languages for representing computational algorithms. I was a member of the international team that developed the initial version of the ALGOL programming language, ALGOL58, originally known as the International Algorithmic Language. ALGOL became the de facto standard for representation of algorithms for the next 30 years. My main contribution, with colleague Ned Irons, was in showing how to implement recursion in the language, something that was initially considered infeasible by many of our fellow team members.

In 1961 I left Chicago for Boston with the key members of my group, to join a new company, (Massachusetts) Computer Associates, focused on developing sophisticated software for syntax-directed compiling. However, our programs had to be run on a remote site. I missed having a computer directly available to me for developing and testing my programs on line. Then I learned about the exciting creative work in the new areas of interactive computing and artificial intelligence newly underway at the Cambridge research firm Bolt Beranek and Newman (now BBN Technologies.) BBN had become a unique center of innovative research and applications in the emerging field of computer science.

I joined BBN in 1962 as a member of the Artificial Intelligence Department (one of the earliest AI organizations) to work on its newly acquired DEC PDP-1 Mark1 computer. My new colleagues were actively engaged in pioneering AI research in computer pattern recognition, natural language understanding, automated theorem proving, LISP language development, and robot problem solving. Distinguished MIT consultants Marvin Minsky and John McCarthy collaborated on some of these efforts. Other BBN groups were doing original work in cognitive science, instructional research, and human-computer communication. Some of the first work on knowledge representation, question-answering systems, interactive graphics, and computer-aided instruction was also underway. J.C.R. Licklider was the spiritual and scientific leader of much of this work, championing the cause of on-line interaction during an era when almost all computation was being done via batch processing. Licklider, Ed Fredkin, and other BBN computer scientists had worked with DEC engineers to help specify the architecture and instruction set of the PDP-1 computer, the machine that was most widely used in early AI work. The PDP-1 was a “*thin-skinned*” computer, designed to facilitate on-line, real-time work. The possibility of developing interactive systems within that kind of computational environment was one of the magnets that attracted me to BBN.

My initial effort at BBN was on expanding the intellectual capabilities of current teaching systems. At that time, virtually all work in computer-aided instruction (CAI) simply imported programmed instruction onto computers, an approach that made the student a passive responder to a built-in computerized instructor. The student’s task was to feed back the information given to him by the system. (Monkey sees, monkey does.) I felt that this was insulting, not only to the intelligence of the student but also to the intelligence of the computer.

Why shouldn’t the student be able to ask as well as answer questions, to direct the interaction as well as be directed by it? And why shouldn’t the computer be able to make adaptively informed responses to the student’s inputs? This led to my building the first *shared-initiative* CAI system, originally dubbed the Socratic System, and the associated *Mentor* language, to enable the development of interactive problem-solving dialogues in which the student and the computer participate on a more equal footing. The initial applications were in decision-making areas such as medical diagnosis. The system provided powerful new capabilities, such as history-sensitive and context-sensitive responses, to support “intelligent” instructional dialogues.

For example, in the differential diagnosis/clinical medical application, a simulated patient presented complaints. The student was free at any time to get a medical history from the patient, to do a physical exam, and to order laboratory tests. As he obtained results, he could make tentative diagnoses. The system could challenge these diagnoses, using its information on what the student should have learned from his inquiry thus far, what potentially relevant information

he had failed to request, and whether he had considered other diseases with similar symptoms. A given student input could produce different responses at different stages of the instructional dialogue, depending on the changing state of the student's knowledge and inferences.

Somewhat later my interest in learner-controlled instruction took a different turn. Around that time, there were two revolutionary advances in computer science and technology; the invention of computer time-sharing and the development of the first high-level "conversational" programming language. The idea of sharing a computer's cycles among several autonomous users working simultaneously on-line had stirred the imagination of BBN and MIT programmers and spurred them to develop software for sharing computer cycles among multiple users. BBN had demonstrated the first successful operation of computer time-sharing in 1964. The initial system supported five simultaneous users on the PDP-1, all sharing a single CRT screen for output. Seeing dynamic displays from several distinct programs, simultaneously and autonomously, ("out of time and tune") was a breathtaking experience.

In 1965, I founded the BBN Educational Technology Department to develop computer methods that would build upon the learning and teaching opportunities these two breakthroughs offered.

Computer time-sharing made use of remote distributed computer stations economically feasible; it opened up the possibilities of widespread computer use in schools. The other new development—interactive programming languages—dramatically increased the power of the tools we could offer students.

Student Programming Languages as Educational Tools: Stringcomp

In 1964 BBN programmers implemented the TELCOMP programming language. It was modeled after JOSS, the first "conversational" (i.e., interactive) computer language, which had been developed in 1962-63 by Cliff Shaw of the Rand Corporation. TELCOMP was a FORTRAN-derived language for numerical computation. BBN made it available as a time-sharing service for engineers.

I saw dramatic new possibilities for such interactive ("thin-skinned") programming languages in education. My interest shifted from earlier work on tutorial systems to the development of interactive programming languages for children, specifically designed for learning. Our initial focus was on making mathematics more accessible and compelling to beginning students. Shortly after TELCOMP was introduced, I extended it by incorporating a capability for symbolic computing operations, to make it useful as an environment for teaching mathematics and language, not just numerical computation. The extended programming language was called *Stringcomp*. It was one of the first languages that supported "non-numerical" (symbolic) computing. Up to then, computers were thought of solely as number processing devices. It's hard to believe now, but the development of "non-numeric computing" was viewed as an AI problem.

In 1965-66, with U.S. Office of Education support, we explored the use of Stringcomp in eight elementary and middle-school mathematics classrooms in the Boston area. After students were introduced to Stringcomp, they worked on writing Stringcomp programs to address standard problems in arithmetic, algebra, and trigonometry and to investigate different approaches to their solutions. The project strongly confirmed our expectation that the use of interactive programming with a high-level interpretive language would be highly motivating to students.

My collaborators in this research were BBN scientists Richard Grant, Daniel Bobrow, and Cynthia Solomon. We were joined by Seymour Papert, who had recently arrived at MIT from Jean Piaget's Institute in Geneva. Danny Bobrow was the go-between, who introduced Seymour and me. He said, "You guys ought to know each other." We agreed, and Seymour then joined our group as a consultant while continuing as a research associate at MIT.

Our positive experience with Stringcomp in the schools suggested the idea of creating a programming language expressly designed for children. Most existing languages were designed for doing computation rather than mathematics. They lacked capabilities for non-numeric

symbolic manipulation. Even their numerical facilities were typically inadequate. For example, they did not include arbitrary-precision integers (big numbers are very interesting to children as well as mathematicians). Existing computer languages were ill-suited for educational applications in other respects as well. Their programs lacked modularity and semantic transparency. They made extensive use of type declarations, which can stand in the way of children's need for expressing their ideas without delay or distraction. They had serious deficiencies in control structure, e.g., lack of support for modularity and recursion. Many, such as the recently developed BASIC language, lacked procedural constructs. Most had no facilities for dynamic definition and execution. Few had well-developed and articulate debugging, diagnostic, and editing facilities, essential for educational applications. These considerations led to the development of Logo.

A Programming Language Expressly Designed for Children: Logo

The need for a new language designed for, and dedicated to, education was evident. The basic requirements for the language were:

- Third-graders should be able, with very little preparation, to use it for simple tasks
- Its structure should embody mathematically important concepts with minimal interference from programming conventions
- It should permit the expression of mathematically rich, non-numerical algorithms, as well as numerical ones

Remarkably, the best model for the new language (*Logo*) turned out to be *Lisp*, the lingua franca of artificial intelligence, which was often regarded by non-users as one of the most difficult languages. Although the syntax of Logo is more accessible than that of Lisp, Logo is essentially a dialect of Lisp. While it inherits most of Lisp's powerful computational capabilities, it expresses programs in language that young students find a great deal easier to use.

Logo was developed at BBN by Dan Bobrow, Richard Grant, Cynthia Solomon, and me, in collaboration with Seymour. We believed that it would point the way to transformational changes in education—not only through its powerful potential for learning real mathematics—though that was our primary initial focus—but also in language, music, and science... Our long-term hope was that it was going to really revolutionise education. Our intent was that work with Logo would help kids learn to think more strategically, not only in mathematics but in other subject areas as well. that it would help them become actively involved in constructing knowledge, not merely taking it in from outside. Our view of children's learning and programming was very different from the more traditional, instructionist ideas of our friends working in schools with CAI or BASIC programming.

The initial language design came about through extensive discussions among Seymour, Danny, and me, beginning at a meeting at my home one evening in 1966. Seymour developed the initial functional specifications. Danny, who was an experienced Lisp programmer, did the first implementation, in Lisp on a Scientific Data Systems SDS 940 computer. For some reason I don't remember, that implementation was tentatively called Ghost. Subsequently, Dick and I made substantial modifications to the design and implementation, with help from Cynthia and BBN engineers Frank Frazier and Paul Wexelblat.

We had some differences of course. For example, Seymour was opposed to the convention that multiplication and division should have precedence over addition and subtraction. Indeed, why should it? Except that to overturn it would have greatly increased the difficulty of the language's acceptance by the mathematics education community.

I named the new language Logo from the Greek λογος, "the word or form which expresses a thought; also the thought itself." Everyone concurred.

The first version of Logo was piloted with fifth- and sixth-grade mathematics students in the Hanscom Field School in Lincoln, Massachusetts in 1967, with support from the Office of Naval

Research, as part of a project on new computer advances for instruction. These trials replicated the success of our earlier classroom work with Stringcomp.

In 1967-68, our group designed a new and greatly expanded version of Logo, which was implemented by BBN software engineer Charles R. (Bob) Morgan on the DEC PDP-1 machine. BBN scientist Michael Levin, one of the original implementers of Lisp, contributed to the design. From September 1968 through November 1969, the National Science Foundation supported the first intensive program of experimental teaching of Logo-based mathematics in elementary and secondary classrooms.

At that time, NSF scientists, together with most people, believed that computers, and especially computer programming, had little relevance for early education. Moreover, they were used to supporting universities, not non-U organizations such as research firms like BBN. In 1968, when I spoke to Milton Rose, head of the NSF Office of Computer Affairs, about providing support for our initial Logo teaching, he was very resistant. He said "Why should I support you—you're not a university." I replied "Because this is very important for education, and no one else can do it!" Which was true. So he relented and provided us the first of several NSF grants for supporting Logo research, development, and teaching through the 1970s and early 1980s.

The seventh-grade teaching materials were designed and taught by Seymour and Cynthia at the Muzzey Junior High School in Lexington, Massachusetts. I designed the second-grade teaching materials with Marjorie Bloom, a schoolteacher who had joined our project as a consultant. We connected two or three computer terminals at each school to the PDP-1 at BBN in Cambridge over telephone lines. I remember Paul Wexelblat stringing wires from the school telephones to our computer in Cambridge. These early teaching experiments demonstrated that Logo offered an effective conceptual foundation for the teaching of mathematics that is intellectually, psychologically, and pedagogically empowering to students.

In November 1969, we published the report of the first fifteen months of the Logo Project, "Programming Languages as a Conceptual Framework for Teaching Mathematics," authored by the entire team. Another major product was a detailed report written by Dick Grant, George and Joan Lukas, and me, documenting a Logo curriculum in four areas, including the evolution of different number representations and the corresponding algorithms of their arithmetic, and problem-solving strategies for maze traversal and sequence extrapolation.

Our intent was not to teach programming as a subject in its own right, but to exploit programming to teach mathematical thinking. A stronger claim would have been to teach generic (i.e., domain-independent) thinking skills. But thinking has to be about *something*, it does not exist in vacuo. There were attempts in the 1960s to develop artificial intelligence programs with general problem-solving capabilities. This quest proved to be a modern day philosopher's stone. To be effective, AI applications require extensive domain knowledge as well as powerful inferencing methods. The same holds here. Programming has been exploited to teach in areas other than mathematics, but the early claims that it can teach generic thinking abstracted from content are not well-founded.

Because of the fundamental connections between the foundations of mathematics and the concepts underlying computation and programming, mathematics is a natural context for Logo presentations. We have developed Logo-based teaching sequences for primary and secondary mathematical topics including number systems, functions, algebra, logic, and problem-solving strategy. Other Logo courses in geometry, algebra, and computer science have been developed by several of you here as well as many others. Hal Abelson and Andy diSessa designed a beautiful treatment of Logo-based mathematics at undergraduate levels covering topics such as random motion, branching processes, space-filling designs, vector operations, topology of curves, spherical geometry, and general relativity. The richness of non-numerical topics open to programming has been exploited for developing curriculum sequences in the arts, sciences, and language study. For example, Horacio Reggini developed extensive Logo-based materials in visual arts areas. Paul Goldenberg and I developed a foundational course on language and linguistics that utilizes Logo's highly-developed capabilities for language manipulation.

Because cognitive skills are domain-dependent, one should not expect that the problem-solving competence acquired in one domain will transfer to others. Thus, it is not surprising that Logo programming skills do not directly transfer to problem solving in other domains. Logo programming can be expected to have stronger effects on the development of *meta-cognitive* skills. These are components of learning that may be domain-independent, and that function as executive processes to monitor the progress and products of cognition. They include skills such as self-monitoring, comprehension, time management, and reflectivity. There is evidence that the use of Logo to teach programming facilitates the transfer of such skills in children.

I was interested in showing that young children—seven-year-olds—could learn to use Logo to write elementary mathematical procedures requiring logical thinking. Some Piagetians at this time seriously doubted whether children that young could exhibit formal-stage thinking. We sought to show that young children could indeed demonstrate formal thinking skills through the use of an appropriate interactive programming language. I investigated the feasibility of using Logo with that goal to primary school children in 1969 at the Emerson School, a traditional public school in Newton, Massachusetts. The students were a group of eight second- and third-graders of average mathematical ability.

After introducing them to the elements of *Logo*, I turned them loose on tasks like writing a procedure for Countdown, the down-counting preceding a space launch. (After the first moon shot, kids everywhere in the US were joyously mouthing this Countdown sequence as a kind of mantra!) The students' task was to write a Logo program for counting integers down from 10 to 0 in steps of 1, followed by "Blastoff!"

The work of one of the seven-year-olds, Steven, illustrates the process of development. Steven, like all second-graders in the group, was familiar with the countdown invocation. He wanted to write a program in Logo to accomplish this. His program, called Countdown, had a variable starting point. For example, to start counting down from 10, one would simply type Countdown 10, with the following result:

10 9 8 7 6 5 4 3 2 1 0 Blastoff!

He designed the program along the following lines. (The English paraphrase corresponds, line by line, to the actual *Logo* instructions.)

To Countdown an input number
 Print the number.
 Is the number equal to 0?
 If it is, type "Blastoff!" and then stop.
 If not, subtract 1 from the number, and call the result "newnumber".
 Then call the Countdown procedure again, using newnumber as the input.

Steven's program, written in Logo, followed this paraphrase very closely. (Note that the procedure calls itself—it employs recursion, however trivially.)

Steven tried his procedure. He was pleased that it worked. He was then asked if he could modify Countdown so that it counted down by 3 each time, to produce

10 7 4 1 Blastoff!

He said "That's easy!" and he edited his program, changing subtract 1 to subtract 3 in the fourth line of the procedure.

He tried the revised program, by typing Countdown 10, with the following result:

10 7 4 1 -2 -5 -8 -11 -14 -17 . . .

The program printed line after line of outputs for several several pages. It had to be stopped manually. Steven looked delighted! When I asked if his program worked, he said "No." "Then why do you look so happy?" He replied, "I had heard about minus numbers before, *but up till now I didn't know they really existed!*"

Steven saw that his stopping rule—testing the current number to see if it is 0, had failed to stop the program. He found the bug in his program: he should also have allowed for the current number becoming negative. He changed the stopping rule to: Test the number. Is it less than or equal to 0? And now his program worked.

Steven's work shows two characteristic aspects of programming activity—the clear operational distinction between the definition and the execution of a program, and the crucial mediating role served by the process of debugging.

Children already knew and liked problems of this sort. They thought that they understood these problems perfectly because, with a little prodding, they could give a loose verbal description of their procedures. But before working with Logo, they had not made these descriptions precise and general, partly for lack of formal habits of thought, and partly for lack of a suitably expressive language.

The process of transforming loose verbal descriptions into precise formal ones becomes possible using Logo and, in this context, enjoyable to children. The value of using Logo becomes apparent when children attempt to make the computer perform their procedures. A partial, or incorrect, solution is a useful object; it can be extended or fixed, and then incorporated into a large structure. Using procedures as building blocks for other procedures is standard and natural in Logo. The use of functionally separable and nameable procedures composed of functionally separable and nameable parts makes the development of constructive formal methods meaningful and teachable.

In 1970, BBN software engineers Bob Morgan and Walter Weiner implemented subsequent versions of Logo on the DEC PDP-10 computer, a system widely used in universities and educational research centers. From 1971 to 1974, BBN made DEC 10 Logo available to over 100 universities and research centers who had requested it for their own research and teaching. Subsequently, with inputs from BBN scientists Paul Goldenberg and George Lukas, we developed the portable Logo system, with the goal of making it relatively easy to port Logo across a wide range of computers.

In 1970, Seymour founded the Logo Laboratory at MIT. It had become more difficult for a research firm like BBN to obtain funding for educational research. It was a great deal easier for a university. Also, there were disagreements about what steps to take to advance Logo. Seymour sought to create a radically new educational system. He abhorred current school systems, though he continued reluctantly to work in schools. My BBN colleagues were interested in enhancing public education by continuing to provide Logo-based instructional materials and associated teacher training. We were well aware of “the homeostatic propensity of schools to resist change”, as aptly put by Bob Davis, the distinguished mathematics education researcher. In principle, school people often agreed about the need for major reform. In practice, however, they echoed Mark Twain's quip: “I'm all for reform—it's just change I don't like.”

We were somewhat naive perhaps in believing that students' work with Logo programming during their formative years would foster dramatic changes in the teaching of mathematics. We were encouraged that many US schools during the 1960s and 1970s were open to the introduction of programming ideas and activities. However, we didn't believe that a social and cultural revolution leading to a major structural overhaul of schools was feasible in the foreseeable future. Through working with teachers and kids in current schools, we sought to develop the ideas and materials that would pave the way for future revolutionary change.

The advent of microcomputers, such as the DEC PDP-11, catapulted Logo into becoming one of the most widely used educational languages during the 1970s and 1980s. Logo was employed in mathematics, computer science, and computational linguistics courses from elementary through undergraduate levels.

In 1971, visiting consultant Mike Paterson introduced the Logo-controlled robot turtle, a “floor turtle” physically connected to the computer via hardwire lines. As the turtle moved, it dragged the connecting wire along with it. Using Logo turtle commands, a student could command the

turtle to move forward or back a specified distance, turn to the right or left a specified angle, sound its horn, use its pen to draw, and sense whether contact sensors on its antennas encountered an obstacle. "Screen turtles" were introduced at the MIT Logo Lab around 1972.

In 1972, BBN engineer Paul Wexelblat designed and built the first wireless floor turtle, "Irving." Irving was a remote-controlled RF turtle about one foot in diameter. It was capable of moving freely under Logo commands via a radio transceiver attached to a teletype terminal connected to a remote computer. We were interested in seeing whether elementary and junior high school students could control Irving to do simple AI tasks such as avoiding obstacles while circumnavigating an area. However, Irving initially lacked touch sensors, so it was not able to detect obstacles. Before we settled on bumpers as the appropriate device for touch sensors, we considered the use of antennas. (If we had done that, we might have called Irving a beetle instead of a turtle!)

During this phase of Irving's development, we wondered what kind of AI task, if any, was feasible for a robot without sensory feedback. On the face of it this seemed impossible—intelligent behavior requires feedback from the environment. However, we conceived a kind of limiting case, the "path reversal" task. The first part of the task was to move Irving from its initial room location to an adjoining room where it was no longer visible. This typically required a sequence of about twenty move and turn commands. After Irving was out of view, the challenge was to bring it back "home," to its starting point in the original room, only through the use of Logo commands, without peeking into the adjoining room, for example.

The student had a complete record of the sequence of commands she had used to control Irving, since each command was printed on the teletype. Students were typically taken aback, because they could no longer see the turtle when it was in the adjoining room. How could they control what they couldn't see? The task was compelling and, for all but the most sophisticated children, seemed non-trivial. Even bright students were initially perplexed. The notion that there is an algorithm for accomplishing it was not obvious. They knew that Forward and Back are inverse operations, as are RightTurn and LeftTurn, and PenUp and PenDown. But they didn't know how to use this for reversing Irving's path.

However, once students were asked how they could make Irving undo its last move so as to get where it had been "the time before last," most had a rapid flash of understanding and knew how to complete the entire path reversal. They saw that the way to return the turtle home is to undo the sequence of actions that the turtle took to reach its final location by performing the sequence of opposite actions in the reverse order.

The path reversal algorithm, an instance of the algorithm for inverting a functional chain, is a mathematical idea of considerable power with a wide range of application. It is the basis for matrix inversion and the solution of linear systems. It has important applications in thermodynamics and system design. It ought to be given a memorable name, such as "the theorem of return." The use of the turtle made it accessible to beginning students.

The path reversal activity directly paved the way for developing a procedure for solving linear equations in algebra. We introduced fifth-grade students to Logo in Cambridge the following year to show that the use of Logo could overcome traditional difficulties in learning algebra. Students worked with tasks like making and solving "secret codes," represented as Logo procedures. Functions and algorithms became familiar objects created to serve purposes that were interesting to them. Students were delighted to find that the algorithm for solving a linear equation—doing the inverse of the operations that generated the equation, but in the reverse order—is the same algorithm that was effective for path reversal.

Function Machines

In the late 1980s, BBN colleagues Sterling Wight, John Richards, Paul Horwitz, Ricky Carter, and I developed Function Machines, a Logo-based visual programming environment expressly designed to support the learning and teaching of mathematics by making the operation of

algorithms transparent. The Function Machines language employs two-dimensional visual representations, graphic icons, in contrast with the symbolic textual expressions used for representing mathematical structures in standard programming languages. The visual representations in Function Machines significantly aid the understanding of function, iteration, recursion, and other key computational concepts. Students are better able to understand an algorithm by seeing its inner operation as it runs, as well as seeing its external behavior, the outputs generated by its operation. Function Machines supports both kinds of visualizations.

Computer modeling and visualization are as valuable for students as they are for researchers, and for very much the same reasons. Modeling bridges the gulf between the world of observation and experiment. Visualization enables students to observe and study complex processes as they are run. It can provide insight into the inner workings of a process—not just *what* happens, but *why*.

In Function Machines the central metaphor is that a function (representing a procedure, algorithm, or mathematical model) is a “machine” (displayed as a rectangular icon with inputs and outputs). A machine’s data and control outputs can be passed as inputs to other machines through explicitly drawn connecting paths. The system provides, as primitive constructs, machines corresponding to the standard mathematical, graphics, list-processing, logic, and I/O operations found in standard Logo languages. These are used as building blocks to construct more complex machines in a modular fashion.

Any collection of connected machines can be encapsulated under a single icon as a higher-order “composite” machine. A composite machine may be used as a component of a more complex machine, or of itself. Proceeding in this way, programs of arbitrary complexity can be constructed. Execution is essentially parallel—many machines can run concurrently. By explicitly showing the passage of data into and out of machines, and by illuminating the data and control paths as machines are run, the program visually animates the computational process. Thus, the program semantics become a great deal more transparent.

Function Machines is one among several innovative extensions of the Logo that came into the world in 1966. There are now over 130 implementations. Some of the later implementations such as GeomLogo, Boxer, StarLogo, NetLogo, Imagine, Elica, and Scratch offer wonderful enhancements to the original versions: capabilities such as object-based animation, 3D graphics, high-fidelity simulation, and visual user interfaces. As Hal Abelson said, Logo is “a philosophy of education and a continually evolving family of programming languages that aid in its realization.”

The Present

Since the 1990s, educational conditions have worsened considerably in US schools. The mania for testing and accountability, supported by federal mandates like the “No Child Left Behind” legislation, has become the driving force in public schools. Testing has become an end in itself, relegating learning secondary and subordinate. Accountability is the watchword of administrators, though the question of what teachers and students should be accountable *for* and *why*, is not addressed. On top of that, there is the dominance of the “tyranny of the clock”—the daily lesson plan, the ruthless component of the scope-and-sequence curriculum, which dictates that all students work on the same lesson at the same time. In nearly all public schools, these developments have spelled the end of the early openness to Logo programming, and have doomed project-based inquiry.

Mathematics education in the US is in trouble. Many students don’t have the foggiest idea of what mathematics is. It simply doesn’t make sense to them. I’m sure you all have examples of kids’ confusions and misconceptions. Here is a nice one, a nonsense problem reported by Harvard math educator Kay Merseth:

There are 125 sheep and 5 dogs in a flock. How old is the shepherd?

In work reported at the 1986 American Education Research Association meeting, researchers reported that three out of four schoolchildren produced a numerical answer. Here is a typical one:

$125 + 5 = 130$...*this is too big*... $125 - 5 = 120$... *is still too big*... $125/5 = 25$.

That works! I think the shepherd is 25 years old.

Though it doesn't do honor to his school math experience, what is notable about this response is that the child is trying to make sense of this nonsense problem.

It is very troubling that many citizens cannot use mathematics and science in their everyday lives, even in situations that have significant personal and social import for them. Democracy requires citizens who can evaluate the risks and benefits associated with nuclear power, who can compare the costs of a year in prison with those of a year in drug rehabilitation or in day care. We are all-too-successful in turning out Americans who cannot employ mathematical ways of thinking to any purpose that matters in their lives. The concern is not that our citizens are unable to perform arithmetic calculations or solve problems in geometry. It is that they often lack the knowledge and skill required for reasoning about the world in which we live, for making sense out of the problems we face, and for making more informed and intelligent decisions about them.

Many adults cannot evaluate the information offered in the daily papers, let alone its implications. Are the numbers in a newspaper article on the costs of health insurance credible? How can we understand the national debt in human terms? How shall we estimate the economic and human cost of a growing AIDS epidemic? It is essential to address this problem during students' formative years by fostering their development of mathematical thinking in situations that call for the use of quantitative reasoning and estimation. The two major components of such reasoning are almost totally ignored in the present curriculum despite being very close to the heart of mathematical practice and art: How does one do approximate calculation and estimation? How does one go about solving open-ended mathematical problems?

A number of *dichotomies* (some true and some false) help to inform our discussion of mathematics education and the role of Logo. First, let's consider:

A False Dichotomy: Tradition versus Reform

There are two often impassioned views of mathematics education, those held by the so-called traditionalist and reformist camps. The former hold that school math should focus on acquiring knowledge of basic number operations and calculation skills. The latter hold that it should focus on the development of critical thinking and problem-solving skills. *This is a false dichotomy!* Children need to be able to do both kinds of things. Of course they should have computational competence. But they should also acquire competence in mathematical ways of thinking and their application to things that matter in their lives. We need to help children develop basic reasoning skills while they are developing basic number and computation skills. These goals are not inherently opposed. We need to go *forward* to basics—not *back*—by moving toward a more comprehensive and powerful set of basic mathematical skills, all of which can be fostered by Logo programming activities.

A True Dichotomy: Mathematics versus School Math

A *real dichotomy* underlies the math wars issue—*mathematics versus school math*.

If all that school math is designed to do is to help kids acquire the ability to do sums, long divisions, and square root calculations, it might be argued that, with the introduction of calculators and computers, school math is no longer necessary. (This is analogous to Jonathan Swift's brilliant 1729 essay "A Modest Proposal: For Preventing the Children of Poor People in Ireland from Being a Burden to Their Parents or Country ..." Swift suggests that the Irish might ease their economic troubles and relieve their population problems by cooking and eating the *poor* children.) In the same satiric fashion, though somewhat more benignly, we suggest that school math should

be removed from the curriculum. Now that we have computers to do calculation, we no longer need children in school math classes. Children could be sent home early to do much more pleasant things (and perhaps more intellectually beneficial ones *as well*.)

I'm being facetious. Of course kids should learn to do calculation, but they should learn it in a distinctly different way, and Logo can make an enormous difference in the ease, enjoyableness, and effectiveness of their learning. At present, six or seven years of school are dedicated to teaching a superficial understanding of numbers and arithmetic operations on them. At the end of this protracted period, a large percentage of students fail to achieve even modest competence. What a terrible and unnecessary waste of time—all those years focused on calculation and the kids can't emulate a mechanical calculator!

Part of the problem is that the standard arithmetic algorithms are taught as cookbook recipes, disconnected both from the world of real mathematics and from the world of kids. Boring, repetitive drills stamp out any flicker of curiosity and generate an indelible perception of mathematics as the realm of lengthy, ritualistic calculations. Our present method of teaching the subject conveys to our students the unmistakable (and lasting!) impression that mathematics is both difficult and deadly dull. Rarely in the course of thirteen years of pre-college education are students given to understand that mathematics can be *fun*.

Of course, you know that it *can*. Each of us has very specific ideas about how Logo can be used to motivate children's development of mathematical ways of thinking while transforming their dislike of school math into fondness for the real thing. Each of us has favorite areas for Logo-based interventions. Let me briefly share two of mine. One is the early introduction of combinatorics. This is the seminal area of mathematics that treats arrangement and ordering problems, the study of different ways of "*jumbling*" things. It's not the process of enumerating a given set of objects—what children learn as counting, but a natural and powerful extension of that—inductive enumeration—appropriately called *counting without counting*. Students are introduced to mathematically rich counting problems such as matching, merging, and sorting that involve different ways of representing objects and operations. Work with these problems gives concrete meaning to powerful ideas such as equivalence, uniqueness, and completeness.

Another favorite area, an elementary introduction to transfinite mathematics, builds on kids' love of big numbers and on their fascination with the idea of infinity, starting from the realization that there is no biggest integer. This branch of mathematics extends the concept of "counting" in a distinctly different way, leading to the development of powerful ideas such as cardinality, denumerable sets, countability, and countable infinity. These topics are a great deal more interesting to kids than the standard school fare, and the basic ideas and proofs are accessible fairly early.

Could topics like these be part of a *new* school math introduced in the context of programming? Of course, as we know, a compelling curriculum is not enough. None of this is possible without teachers who know the underlying mathematics, who are able to learn from their students, and who are comfortable at times about relinquishing control and ceding it to their students. Finally, it requires a trusting school culture and a political and social environment that support the challenge and risks of project-based learning, a willingness to trade the predictability of lock-step, scope-and-sequence, lesson-plan structures for the development of kids who are provably mathematically literate (yes, accountably so!) *and much much happier* in their math classes.

There are several other false dichotomies in education: *structure versus freedom, knowledge versus creativity, instruction versus construction*. These are often viewed as diametrically opposed, adversarial positions. It's as though if you are on one side you can't possibly concede anything to the other. It reminds me of my friend Oliver Selfridge's characterization of much of the thinking in current Artificial Intelligence research—that things must either be true or false—as *binary heresy*.

This either-or kind of thinking can lead otherwise intelligent people to say stupid things that express false dichotomies. For example, a few years ago a highly respected Boston economist, assessing the effects of computer technology in education, wrote "Computers have proved to be

ineffective in education. How do we know that? Because we've had computers in schools for several years now, and schools are still terrible!" An obvious response: books must *really* be ineffective, because we've had books in schools for much longer, and still, in the terse words of Brian Harvey, "Schools suck!"

Another False Dichotomy: Structure versus Freedom

Students need disciplined guidance along the way to acquiring new knowledge and skills. They also need opportunities to actively explore and experiment with the new concepts and materials on their own, to test and try to extend their understanding by designing and constructing artifacts. Without structure, there is no freedom. And without freedom, there is no foundation for creative development and intellectual growth.

The music of Johann Sebastian Bach exemplifies the powerful synergy between structure and freedom. Bach's Art of Fugue, for example, is one of the most transcendent and emotionally charged works in all music, a model of creative freedom. Yet, it is so mathematically structured that its overall organization and the interrelations among its component elements can be precisely described as a sequence of formal mathematical algorithms. The beautiful drawings and sculptures of M.C. Escher, the most mathematical graphic artist of our time, show the same incredibly imaginative integration of structure and freedom.

An educational philosophy that often extols freedom while abhorring structure is the progressive school movement, exemplified by A.S. Neill's Summerhill School in England, which is noted for its philosophy that children learn best with freedom from "coercion." All lessons are optional, and pupils are free to choose what to do with their time. The school was founded with the belief that "the function of a child is to live his own life—not the life that his anxious parents think he should live, not a life according to the purpose of an educator who thinks he knows best." For some troubled kids, the unstructured school environment can provide a nurturing, and perhaps corrective, experience. However, for kids who don't already come with their own learning agenda, it can be an ineffective intellectual experience.

Another example of freedom without structure, this one from Logo: One of the middle-school students who was introduced to Logo in one of our early teaching experiments, came to me and said "Now I know how to program. *Tell me what to program.*" Students need to be motivated, to have their own purposes and drives. They need to learn how to make their vague ideas clear, precise, and expressible, to structure their thinking. They also need to acquire a body of content knowledge that they can draw upon to concretize their mental constructs. This goes against the instinctive tendency of some of our colleagues, for whom the *very idea* of an explicit teaching agenda or curriculum is anathema.

Work with Logo can also exhibit *structure without freedom*. I have seen curriculum materials designed to teach Logo programming that are so tightly prescribed and circumscribed, that they bring to mind the "do it by the numbers dot-to-dot" paper drawing exercises for young children. In one case, in a Logo teaching sequence in a New York City school, students were told which commands to enter in a given procedure, line-by-line, from To all the way to End. They were then instructed to run "*their*" procedure, after which they were told, "Look what you've discovered!"

Another Dichotomy: Instruction versus Construction

This is a well-intended distinction but, ultimately, a deceptive one. We need *both*. Knowledge instruction and knowledge construction are synergetic learning components, intimately joined in the learning process. They work hand in hand, mind to mind. Instruction is often a precursor to construction. We learn from the mistakes we make in constructing knowledge. We try, as Thelonus Monk so profoundly put it, to make "the *right* mistakes." That's what debugging is all about.

The distinction between instruction and construction serves to highlight the important contrast Seymour draws between *instructionism* and *constructionism*. Instructionism is the extreme case of instruction without construction, i.e., without enabling students to make the knowledge their own, to “own” it and move forward from there, on their own. Unfortunately, instructionism all too often characterizes *schooling* as oppose to *learning*—yet another dichotomy!

Constructionism in the sense of making knowledge construction the primary goal of instruction is what most of us in the Logo community strive to achieve in our teaching. There is, however, an extreme brand of constructionism that rejects instruction altogether and that extols learning without teaching—“teacher-proof learning.” It expresses a sentiment that seems attractive: for example, we admire autodidacts—intelligent persons who are “self-taught”. But, alas, the notion that we can dispense with teaching as a key component of learning is wishful thinking. It simply doesn’t compute! Learning requires shared interactions between children and teachers. Sometimes, during these interactions, the children teach and the teachers learn. And programming can provide a powerful mediating role.

A Russian scholar friend tells the following joke. Bush, Blair, and Putin are given an audience with God and invited to ask what the deity can do for their countries. Bush says, “When will the world realize that the US knows what’s right and what’s good for everyone?” God responds, “It will happen, Mr. Bush, but not in your time. Blair says, “When will the countries of the world acknowledge the great contributions Britain has made to democratic institutions and ideals?” God responds, “It will happen, Mr. Blair, but not in your time.” Then Putin says. “God, when will Russia become a normal country like other countries?” This time, God responds, “It will happen, Mr. Putin, but not in My time.”

When will constructionism take over educational practice? Not in our time, and certainly not in schools as we know them. So let’s be realistic. This *is* our time. De-schooling will not occur during our lives! Schools are going to be around for the foreseeable future. Should we abandon working with schools? If not, what should we do to foster constructionism? Let’s discuss some possibilities—after considering one more false dichotomy.

The Last False Dichotomy: Revolution versus Reform

Why is this a *false* dichotomy—surely, revolution is not reform? Aren’t they polarities? For a long time, the Logo movement has been (somewhat simplistically) characterized as consisting of two warring camps—the reformers and the revolutionaries. Those in the BBN Logo group were labeled reformers, because we believed that Logo would ultimately make significant inroads toward school reform. The MIT Logo group, led by Seymour, called for a fundamental restructuring of education, a political and social (though non-violent) revolution.

I share the revolutionary perspective—that *is* the goal we should work toward. But I have somewhat different views about what should be done to advance that goal in our time. Like many of you, I believe in continuing to work with Logo within the current school world. Large-scale revolutionary school reform may not be possible during our lifetime, but there are school classrooms where the Logo philosophy and culture can flourish today.

There are some things we can do to promote and facilitate a groundswell of support for revolutionary educational change. Fundamentally, we need to work toward creating a thirst for intellectual curiosity and critical thinking in society. We want to build a critical mass of citizens who see the need to develop places of learning that do away with the false and dysfunctional dichotomies and create, in their stead, environments that integrate structure and freedom, knowledge and creativity, instruction and construction.

This is an awesome challenge. It will require an extensive “propaganda” campaign via television and the Internet, actively supported by leading national figures—celebrities with great popular appeal and influence, such as movie and television stars and sports heroes. And it requires skilled media personnel, working with us, to design exemplary entertainment programs that demonstrate how mathematics, the sciences, the arts, and other subjects can be presented in

new and compelling ways through programming and new cognitive games technology. It's very ambitious indeed, but there's hope that this thrust can make significant progress in our time.

Another initiative is a cognitive apprenticeship approach to mathematics and science education. This would mirror the pre-college model in which young music students spend significant time at a conservatory—perhaps several hours on Saturdays over a period of years—not only learning to play an instrument but also participating in a comprehensive music education program—performing in choral and instrumental ensembles, musicology, music history, and composition—on the path to becoming complete musicians. We can draw upon a potentially enormous resource of retired mathematicians, scientists, engineers and teachers, who love their subjects, who can work effectively with kids in shared endeavors, and who have the time and interest to participate, given the opportunity.

This is an interesting time in the United States. Increasing numbers of people are fed up with high-level incompetence. Most people have begun to see the intellectual, as well as the moral, failings of our leadership. Many wonder at how the population could have been so ill-informed as to have (apparently) voted for an anti-intellectual government. Our concern is not just about mathematics and science education. It's about intelligent citizenship. It's about creating a new generation of young people who are thoughtful about their lives and their social, cultural, and political decisions, and who understand the sense and purpose of the ideas and goals we all hold dear.