History of Logo

CYNTHIA SOLOMON, Cynthia Solomon Consulting, USA
BRIAN HARVEY, University of California, Berkeley, USA
KEN KAHN, University of Oxford, UK
HENRY LIEBERMAN, MIT Computer Science and Artificial Intelligence Lab (CSAIL), USA
MARK L. MILLER, Learningtech.org, USA
MARGARET MINSKY, New York University-Shanghai, China
ARTEMIS PAPERT, Independent artist, Canada
BRIAN SILVERMAN, Playful Invention Co., Canada

Shepherd: Tomas Petricek, University of Kent, UK

Logo is more than a programming language. It is a learning environment where children explore mathematical ideas and create projects of their own design. Logo, the first programming language explicitly designed for children, was invented by Seymour Papert, Wallace Feurzeig, Daniel Bobrow, and Cynthia Solomon in 1966 at Bolt, Beranek and Newman, Inc. (BBN).

Logo's design drew upon two theoretical frameworks: Jean Piaget's constructivism and Marvin Minsky's artificial intelligence research at MIT. One of Logo's foundational ideas was that children should have a powerful programming environment. Early Lisp served as a model with its symbolic computation, recursive functions, operations on linked lists, and dynamic scoping of variables.

Logo became a symbol for change in elementary mathematics education and in the nature of school itself. The search for harnessing the computer's potential to provide new ways of teaching and learning became a central focus and guiding principle in Logo language development. It encompassed a widening scope that included natural language, music, graphics, animation, story telling, turtle geometry, robots, and other physical devices.

CCS Concepts: • **Software and its engineering → General programming languages**; • **Social and professional topics** → *History of programming languages.*

Additional Key Words and Phrases: Logo, Lisp, Constructionism, Constructivism, turtle geometry, Education–Interactive learning environments, History of computing–History of programming languages, Computing education– Computational thinking, Computing education programs—Computer science education, Informal education, Computing literacy, K-12 education (ages 5-18), Software notations and tools–General programming languages, Imperative languages, Functional languages, Language features–Control structures, Data types and

Authors' addresses: Cynthia Solomon, Cynthia Solomon Consulting, 14E Kenneson Road, Somerville, Massachusetts, 02145, USA, cynthia.solomon@gmail.com; Brian Harvey, University of California, Berkeley, 784 Soda Hall #1776, Berkeley, CA, 94720-1776, USA, bh@berkeley.edu; Ken Kahn, University of Oxford, 15 Norham Gardens, Oxford, OX2 6PY, UK, toontalk@gmail.com; Henry Lieberman, MIT Computer Science and Artificial Intelligence Lab (CSAIL) , 32 Vassar St. G475, Cambridge, MA, 02139, USA, lieber@media.mit.edu; Mark L. Miller, Learningtech.org, 751 Laurel St. #411, San Carlos, CA, 94070, USA, mlmiller@learningtech.org; Margaret Minsky, New York University-Shanghai, 1555 Century Ave., Pudong, Shanghai, 200122, China, margaret.minsky@gmail.com; Artemis Papert, Independent artist, Montreal, Quebec, Canada, artemis@turtleart.org; Brian Silverman, Playful Invention Co., Montreal, Quebec, Canada, brians@playfulinvention.com.

2475-1421/2020/6-ART79
https://doi.org/10.1145/3386329

structures, Recursion, Procedures, functions and subroutines, Theory of computation–Models of computation—Computability–Lambda calculus, Recursive functions, Interactive computation.

ACM Reference Format:
Cynthia Solomon, Brian Harvey, Ken Kahn, Henry Lieberman, Mark L. Miller, Margaret Minsky, Artemis Papert, and Brian Silverman. 2020. History of Logo. *Proc. ACM Program. Lang.* 4, HOPL, Article 79 (June 2020), 66 pages. https://doi.org/10.1145/3386329

Contents Summary

Contents

1 INTRODUCTION

> "I believe with Dewey, Montessori and Piaget that children learn by doing and by thinking about what they do. And so the fundamental ingredients of educational innovation must be better things to do and better ways to think about oneself doing these things."
> . . .
> "We can give children unprecedented power to invent and carry out exciting projects by providing them with access to computers, with a suitably clear and intelligible programming language and with peripheral devices capable of producing on-line real-time action."
> . . .
> "…in its embodiment as the physical computer, computation opens a vast universe of things to do. But the real magic comes when this is combined with the conceptual power of theoretical ideas associated with computation." [Papert 1972a]

Logo was to be a language for learning, for exploration, and especially for mathematics.

In 1966 computers were few, large, and sprawling; programmers interacted with them either by submitting card decks to a feeder or by using terminals situated outside of the computer area. But computers' power and promise as a tool for understanding intelligence was in the air. This vision permeated the two labs in which Logo was initially crafted: Bolt, Beranek and Newman, Inc. (BBN) and the MIT Artificial Intelligence Lab.

1.1 The Logo Environment

From its inception Logo was more than just a programming language. It was also a computer environment made up of people, things, ideas. And it was a computer culture: a way of thinking about computers and about learning and about talking about what you were doing.

Functionally, the Logo environment is comprised of the following:

(1) a computer
(2) a programming language
(3) a collection of computer peripherals, usually including a robot called a "turtle" (and a graphical simulation of it, called a "display turtle")

(4) a collection of projects
(5) a meta-language: a consistent way of talking about the language, the projects, etc.
(6) a rich learning culture
(7) a collection of "bridge activities" such as juggling and puzzle-solving

All of these components are interdependent and the special virtues of the environment follow from their synergy. For example, one would expect very limited educational benefits to arise from merely teaching programming, even Logo programming, in a completely abstracted context, or from using turtles as toys without the insight one derives from the underlying computer culture.

The design of the environment as a whole was strongly influenced by several ideas that are central to Logo work with young children: procedurization, anthropomorphization (Papert later talked about body syntonics), and debugging.

A timeline of notable Logo events is in Appendix A.

1.2 Logo Programming: An Example

Here is an example of some Logo code. The earliest programs focused on playing with words and sentences. The turtle, which became iconic for Logo, appeared later.

`TO` introduces a procedure definition. `OUTPUT` sends a value back to the caller of the procedure. `PICK` chooses an item at random from a list.

```
TO NOUN
OUTPUT PICK [BIRDS DOGS WORMS DONKEYS GEESE CATS [GUINEA PIGS]]
END

TO VERB
OUTPUT PICK [HATE TRIP BITE LOVE]
END

TO ADJECTIVE
OUTPUT PICK [RED PECULIAR JUMPING FAT FUZZY [FUZZY WUZZY]]
END

TO SENGEN
PRINT (SENTENCE ADJECTIVE NOUN VERB ADJECTIVE NOUN)
SENGEN
END
```

When `SENGEN` is invoked,[1] this code produces sentences such as

```
RED GUINEA PIGS TRIP FUZZY WUZZY DONKEYS
PECULIAR BIRDS HATE JUMPING DOGS
FAT WORMS HATE PECULIAR WORMS
FAT GEESE BITE JUMPING CATS
```

You might ask: does `SENGEN` make up sentences the way we do as adults or the way we did when we first learned to talk or write? You might also ask: what relationship does `SENGEN` have to understanding grammar? The first question is open to research and speculation. The second question might be an easier one to answer. Often when we discussed this project with children they did not relate the programming process to grammar. Later as they used their programs, the

[1] One way to try this is to copy and paste this program into an online Logo such as https://www.calormen.com/jslogo/ and then type `SENGEN`.

children frequently exclaimed "So this is why they call them nouns and verbs!" The children also began to appreciate formal systems. Studying grammar by generating sentences that obey certain rules requires the programmer to become aware of rules as well as exceptions.

Since this program seems to make sensible sentences without knowing very much about grammar, children often develop an appreciation for cleverness. For example, SENGEN doesn't know that some words are singular and some are plural, or that singular nouns should be matched with singular verbs. It does not know about verb tenses or pronominal relations. Its apparent intelligence comes from the programmer's choice of words and categories. SENGEN builds sentences from vocabulary lists of nouns, verbs, adjectives, connectives, and so on. It then assembles its selections according to some rules of grammar.

In 1968-69 a class of seventh graders (12 year olds) tackled SENGEN. The major bug they encountered was in not paying attention to whether a word or phrase was a noun, a verb or an adjective. It was relatively straightforward for the children to make procedures to output lists of words from which another procedure could select a word but they were completely taken aback when their sentences looked like this:

```
GEESE FAT WORMS GEESE JUMPING
DOGS JUMPING CATS FUZZY FAT
DOGS PECULIAR FUZZY DOGS FAT
GEESE HATE DONKEYS DOGS BIRDS
FUZZY WUZZY HATE FUZZY FUZZY BIRDS
FUZZY WUZZY PIGS DONKEYS FUZZY BIRDS
```

when they expected this:

```
FUZZY GUINEA PIGS LOVE JUMPING DOGS
RED WORMS TRIP RED DONKEYS
FAT GUINEA PIGS BITE FAT CATS
RED GUINEA PIGS LOVE FUZZY GUINEA PIGS
FUZZY GUINEA PIGS HATE RED WORMS
PECULIAR GUINEA PIGS HATE FAT WORMS
```

That's when they seemed to explode with surprise and realize that they do need to separate verbs from nouns. No one had to tell them what the bugs were. They knew enough about sentence construction to see the bugs, and now they were learning explicitly some rules of grammar that they already knew implicitly. They were, after all, seasoned creators of understandable sentences.

A second example makes algebraic sentences instead of English sentences and makes a quiz out of the result.

```
TO GETNUM
OUTPUT 1 + RANDOM 10
END

TO GEN
MAKE "NUM1 GETNUM
MAKE "NUM2 GETNUM
MAKE "TRIAL GETNUM
MAKE "ANSWER :NUM1 * :TRIAL + :NUM2
PRINT (SENTENCE :NUM1 [* BOX +] :NUM2 [=] :ANSWER)
GETBOX
END
```

```
TO GETBOX
QUESTION [WHAT IS BOX?]
IF ANSWER = :TRIAL [PRINT [GREAT! YOU GOT IT!] STOP]
GETBOX
END
```

Trying it out:

```
GEN
5 * BOX + 10 = 35
WHAT IS BOX?
5
GREAT! YOU GOT IT!

GEN
9 * BOX + 8 = 89
WHAT IS BOX?
5
WHAT IS BOX?
19
WHAT IS BOX?
9
GREAT! YOU GOT IT!
```

2 EARLY INFLUENCES ON LOGO

2.1 People

2.1.1 Wallace (Wally) Feurzeig. Feurzeig was originally from the Chicago area having worked at Argonne National Lab and attended University of Chicago and Illinois Institute of Technology. He was a mathematician and a musician (pianist). He joined BBN in 1962 as part of Tom Marill's Artificial Intelligence Department. He first worked on Mentor, an intelligent teaching system. (Later Cynthia Solomon was hired to make a Lisp version of Mentor.) In 1965 Feurzeig formed BBN's Educational Technology Department and started working on programming languages as educational environments. Then in collaboration with Papert and others he started the project that became Logo. He said [Feurzeig 2006] this about it:

> The need for a new language designed for, and dedicated to, education was evident. The basic requirements for the language were:
> (1) Third-graders with very little preparation should be able to use it for simple tasks.
> (2) Its structure should embody mathematically important concepts with minimal interference from programming conventions.
> (3) It should permit the expression of mathematically rich non-numerical algorithms, as well as numerical ones.

2.1.2 Seymour Papert. Seymour Papert grew up in South Africa. His father was an entomologist. For his first couple of years Papert and his parents chased after the tsetse fly. Papert went to University of Witwatersrand in Johannesburg where he pursued mathematics and philosophy. He got a first doctorate in mathematics there and then was awarded a scholarship to a mathematics doctoral program at University of Cambridge, UK. Towards the end of his doctoral work at Cambridge Papert moved to Paris to work on functional analysis. In Paris Papert attended lectures by Jean Piaget at

La Sorbonne. Piaget liked having mathematicians in his research group in Geneva, Switzerland and asked Papert to join.

2.1.3 Jean Piaget. Papert was a researcher in Piaget's Genetic Epistemology Group from 1958 through 1963. Genetic Epistemology according to Piaget, its originator, "attempts to explain knowledge and in particular scientific knowledge, on the basis of its history, its sociogenesis, and especially the psychological origins of the notions and operations upon which it is based." [Piaget 1971]

Fig. 1. Jean Piaget in the Swiss Alps (Ioanna Papandropoulou, 1973, courtesy of © Fondation Jean Piaget / Ioanna Papandropoulou)

Papert comments on Piaget: "when he died at age 84, Piaget had created several new fields of science: developmental psychology, cognitive theory and evolutionary epistemology. He championed a new way of thinking about children. One might say that Piaget was the first to take children's thinking seriously."

Papert goes on to say that Piaget "is revered by generations of teachers inspired by the belief that children are not empty vessels to be filled with knowledge (as traditional pedagogical theory has it), but active builders of knowledge—little scientists who are constantly creating and testing their own theories of the world." [Papert 1999, p. 105]

Papert talks about number as an example of Piagetian thinking. "Number is not something with an independent objective existence that children happen to have a particular conception of. Instead, the study of number is the study of something in evolution, something in the process of construction. Children don't conceive number, they make it. And they don't make it all at once or out of nothing. There is a long process of building intellectual structures that change and interact and combine." [Papert 1988, p. 4]

Many educators consider the important ideas from Piaget to be about stages of development. Not so for Papert who found Piaget most enlightening when he talked about children constructing structures of knowledge. Papert said "In my Piaget, stages and even most senses of 'active learning' are quite secondary. I focus instead on his constructivism and structuralism." [Papert 1988, p. 4]

"The core of Piaget is his belief that looking carefully at how knowledge develops in children will elucidate the nature of knowledge in general." [Papert 1999, p. 105]

2.1.4 Marvin Minsky. In 1961 Marvin Minsky and Papert met at a conference in England. They each presented papers on reinforcement learning mechanisms with changing probabilities. The two papers were surprisingly similar in both their findings and their ways of thinking. From this meeting they decided to work together. In January 1964 Papert arrived at MIT and joined Minsky's Artificial Intelligence Group (later to become the AI Lab).

They collaborated very closely for about 20 years and then less closely for another 20 years. Papert and Minsky thought about thinking, about children's thinking and about machine's thinking. Initially their collaboration focused on machine vision, robotics, computational geometry and advising students. Their discussions about how to make machines intelligent included discussions about children's acquisition of knowledge and the contributions of computers to children's learning.

Minsky describes programming:

> Programs "make things come to be, where nothing ever was before. Some people find a new experience in this, a feeling of freedom, a power to do anything you want. Not just a lot—but anything. I don't mean like getting what you want by just wishing. I don't mean like having a faster-than-light spaceship, or a time machine. I mean like giving a child enough kindergarten blocks to build a full-sized city without ever running out of them. You still have to decide what to do with the blocks. But there aren't any outside obstacles. The only limits are the ones inside you." [Minsky 2019a, page 5; Solomon et al. 1986]

> "Making Logo programs is a lot like building with construction toys—but it's even better. You can make drawings of things and structures, but you can make procedures, too. You can make them use words. You can make things change their forms. And you can make them interact: just make the properties of some of your objects depend on some features of other objects. As toys, those programs have their faults: you can't take Logo cars outside and roll them down a real hill—but, in exchange, their parts don't get loose and fall out and get lost. And the basic experience is still there: to see how simple things can interact to make more wonderful things." [Minsky 2019a, page 8; Solomon et al. 1986]

During his collaboration with Papert, Minsky continued to write about children, computers, and school. Six of his essays can be found in *Inventive Minds: Marvin Minsky on Education* edited by Cynthia Solomon†[2] and Xiao Xiao [Minsky 2019b].

2.1.5 Other Early Contributors. Daniel Bobrow was one of Minsky and Papert's students and upon finishing his doctorate in 1964 became head of the Artificial Intelligence Group at BBN. At the time Feurzeig was leading the BBN education group. Around 1965 Papert began consulting with Bobrow and then Feurzeig at BBN, and discussed with them the idea of a computer language for children.

Cynthia Solomon† had joined the MIT AI group in 1962. She wanted to learn to program. A friend introduced her to Marvin Minsky and she took a job as his secretary. As hoped, she did learn a bit of Lisp programming. By 1966 she was a member of Feurzeig's education group at BBN, working on a Lisp project. For the first few years of Logo development Papert and Solomon were in constant collaboration and it was hard to say who contributed what. This cross-fertilization of ideas with Minsky and Papert, and Papert and Solomon, fed into both work with machines and work with children. In the summer of 1969 Papert and Solomon stopped their work at BBN with Feurzeig and started the Logo Group as part of the MIT AI Lab. The Logo Group was often called a lab but it always remained a part of the MIT AI Lab. Feurzeig and his group continued work on Logo independently from Papert and Solomon.

2.2 Sputnik

In the 1950s work was underway to improve American high school as well as elementary school education especially in math and science. When the Russians sent Sputnik, the first artificial satellite,

[2]The dagger symbol † is used to indicate a reference to one of the authors of this paper.

into orbit on October 4, 1957, a major crisis was felt in the United States. The result was a surge in funding for improving schools, especially math and science education.

By the 1960s "new math" projects were being tested in classrooms around the country. Logo, offering another new path to children learning mathematics, fit in well with the times. The 1968-69 NSF BBN Logo teaching project (see Section 4) had three leading math researchers and a science education researcher as project evaluators. Andrew Gleason at Harvard University had been co-director of the 1963 Cambridge Conference on School Mathematics [Cambridge Conference on School Mathematics 1963]. Robert Davis at Syracuse University, was director of the Madison Project for elementary education. Max Beberman at University of Illinois directed UICSM for high school mathematics. Beberman was considered the father of "new math." Robert Karplus, a physics professor at UC Berkeley led an effort to create a new science curriculum for elementary school, Science Curriculum Improvement Study (SCIS) at the Lawrence Hall of Science. The conversations with this group fortified Papert's idea of a "mathland," an environment as conducive to learning math, as living in France would be to learning French.

2.3 Places

2.3.1 BBN, Time-Sharing, and Interactive Programming Languages. BBN was a hotbed of forefront computing activities. Around 1962, before he joined the Stanford faculty, John McCarthy designed a time sharing system at BBN. (McCarthy, of course, is also the inventor of Lisp.) Coincidentally, Dartmouth Professors John Kemeny and Thomas Kurtz visited McCarthy at BBN. As a result the Dartmouth Time Sharing System was envisioned with the goal of opening computers up to the entire college community. Kemeny and Kurtz designed an interactive programming language they named BASIC. System and language were operational by 1964. McCarthy's early proselytizing for time sharing encouraged Fernando J. Corbató to develop the Compatible Time Sharing System (CTSS) at MIT, often cited as the first operational time sharing system, about a year before the BBN system. As is evident from the names, CTSS influenced the Incompatible Timesharing System (ITS) for the PDP-6 and PDP-10 at the MIT AI Lab, although the design of ITS was based on *disagreements* with that of CTSS. ITS, in turn, strongly influenced SITS, the Small Incompatible Timesharing System, developed by Ron Lebel for the Logo Group's PDP-11.

During that time Cliff Shaw from the Rand Corporation visited BBN and showed his interactive programming language, Joss. BBN researchers implemented their own version of it, calling it Telcomp. Feurzeig modified Telcomp to include strings. Stringcomp was used in Feurzeig's project of eight elementary and middle school classrooms. It was while visiting these classrooms in 1966 that Seymour Papert saw a need for a different kind of language for learning—a language for children. Thus began Logo. [Walden et al. 2011]

2.3.2 MIT Hackers and Their Computer Culture. The MIT AI Group/Lab included Marvin Minsky, students and staff. It didn't have its own computers until 1963, but it did have access to computers at MIT: a batch processing IBM 704, a DEC PDP-1, and the TX-0. The PDP-1 was a single user platform attracting not only Minsky but undergraduates who also hung out at MIT's Tech Model Railroad Club (TMRC) or MIT's radio station (then WTBS).

In the early 1960s cutting edge work with computers was performed primarily by students, both graduate and undergraduate. At this time faculty of the MIT AI Group were either members of the Math Department or the Electrical Engineering Department, since an MIT computer science department didn't yet exist. Minsky showed his respect for the students (whom he recognized as often knowing more than the faculty about computers) by working with the growing number of undergraduates hanging out around the PDP-1.

This PDP-1 was also home to the original Space War, the first graphical computer game. Solomon† says that for years it was the only video game she would play. Minsky said that it gave him the opportunity to be the first professor to ban video games in his lab during working hours.

Fig. 2. Dan Edwards and Pete Sampson Playing the SpaceWar! video game (courtesy of MIT Museum Collections)

Creating a computer science presence at MIT was in the works and in the summer of 1963, under the direction of Professor Robert Fano, Project Mac opened up to an outstanding group of visiting and permanent researchers in its new quarters at 545 Tech Square, on the 8th and 9th floors. The 9th floor housed computers including the AI Group's own PDP-1, which was soon traded in for a PDP-6; later, a PDP-10 was added.

Papert stepped into this environment in December 1963. This plethora of computing power and computing knowledge was an eye-opener for Papert. The undergraduate whiz kids were called "hackers." The word hacker then, and even now at MIT, meant a wizard at some kind of technology. You could be a telephone hacker, or a steam tunnel hacker—or a computer hacker. The AI Lab hackers were incredible computerists. They were curious and adventurous, following in the MIT tradition. If they dug themselves into a hole they dug themselves out of it, sometimes with and sometimes without help from others. The learning was impressive and so were their contributions. This is very different from the pejorative sense of hacker in current media.

But perhaps more important than their technical contributions was the pervasive attitude of "I can do this, and so can you [whoever you may be] if you want to learn," no matter what the "this" was. Brian Harvey† vividly remembers, as a freshman new to the AI Lab, reporting a bug in the text editor and being told to fix it himself. That was both terrifying and exhilarating, very different from the usual "hands off the computer" attitude toward non-experts.

Lisp, the language created by John McCarthy, was the language of choice for the AI graduate students. Lisp (the name stands for "list processing") encourages recursive functions and symbolic manipulation of programs as data. This fit into the ideas for AI research at the time, and also served as a model for Logo.

In a research statement in 1971, Minsky and Papert wrote:

> "The Artificial Intelligence Laboratory is a center for research into the nature of intelligence. It differs in two ways from the host of 'psychological' laboratories which would so describe themselves. It has a powerful theoretical bias towards

Fig. 3. Marvin Minsky and Seymour Papert (Cynthia Solomon collection)

a stream of ideas one might describe as 'computational' or 'cybernetic' and it is firmly dedicated to the idea that theories of intelligence will grow only in an environment in which they can be translated into practical experiments of which we know of two kinds: (1) programming machines to perform intelligent functions, and (2) teaching people."

"Our interest in the development of intelligence in children is long-standing. Both of us were extensively involved with psychology from our student days, though formally we graduated in mathematics. One of us worked for five years (1959-63) in Piaget's Center in Geneva, certainly the most distinguished 'natural intelligence laboratory' in the world."

"We actually want to teach computer science to young children because we believe that it has a unique ability to elucidate vitally important concepts; our classroom use of the computer itself is subservient to our intention of teaching these concepts."

"Amongst the concepts computer science has elucidated is that of thinking. We conjecture (on the basis of plausible arguments and a little empirical evidence) that giving children the ability to think better about their own thinking could have a very powerful effect on their intellectual development. The exploration of this conjecture is typical of the long range goals of our experimental work in education."

2.3.3 Eighth Floor Culture and Ninth Floor Culture. The ninth floor at 545 Technology Square housed the computers of the AI Lab. More importantly it housed the system hackers who worked on the hardware, operating systems, and programming languages used by everybody in the Lab. The eighth floor (and later, expanding to the seventh) was the domain of professors, graduate students, academic researchers, and administrators. It was the unique culture of the MIT AI Lab that there was much interaction between the floors. Theory met practice. The graduate students, and even some of the professors, didn't mind getting their hands dirty with low-level coding. The system hackers often made significant contributions to research and published papers. Many histories of computer science have emphasized how much the development of AI was rooted in the hacker

culture of the 1960s and 1970s [Levy 1984; Raymond 1996]. Much of that hacker culture originated at MIT at the time.

Artificial Intelligence was, at the time, a brand-new subject, and a subject that met with considerable skepticism from more traditional computer scientists and mathematicians. Minsky taught the one graduate course on it. Mostly the hackers taught themselves. Levy recounts the development of hacker culture at the Tech Model Railroad Club, a group of hobbyists who did things out of sheer joy rather than seeking employment or credentials. People such as Richard Greenblatt, Bill Gosper, and Richard Stallman became the archetypes for the hacker culture.

Minsky and Papert's management style gave the hackers free rein. They cared more for fundamental scientific questions rather than money, status or power. They trusted people to do the right thing. They disliked bureaucracy and rules. The fortunate circumstance of unconditional support from the US Department of Defense Advanced Research Projects Agency (ARPA) let them operate without much oversight.

What Papert found in the hacker community were extraordinary examples of learning and problem solving that took place organically. Hackers had to learn an enormous amount of technical detail, without being formally taught. They didn't care about credentials or boundaries between fields. They had a collaborative social culture, where much activity occurred after midnight, at Chinese restaurants, or both.

2.4 The Synthesis: Constructionism

In the 1980s, Papert gave a new name to the coming together of the influences described in the previous sections: constructionism. In a proposal to the National Science Foundation he said

> From constructivist theories of psychology we take a view of learning as a reconstruction rather than as a transmission of knowledge. From a rich body of educational experience we take the view that learning is particularly effective when it is embedded in an activity the learner experiences as constructing a meaningful product (for example, a work of art, a functioning machine, a research report or a computer program.) [Papert and Group 1986]

> Constructionism also goes beyond (while again including) the cognitivist principle that underlying deep structures are central to learning science: to the cognitive deep structures it adds a number of deep dimensions: affective, aesthetic, and socio-cultural to which we give at least as much weight as the cognitive factors. [Papert and Group 1986, p. 10]

3 TECHNICAL CHARACTERISTICS OF LOGO

This section of the paper is focused on describing technical decisions about Logo as a language.

Logo is, first of all, a dialect of Lisp. Its original authors were Lisp users at MIT and BBN; since the idea was that Logo would be a language for manipulating (human) language, Lisp's emphasis on symbolic programming seemed like a great match. In retrospect, this choice was by far the most important aspect of Logo's design, mainly because of Lisp's interactive development model, next because of the use of recursion as the main control structure, and third because of Lisp's support for symbolic computing.

Logo's design differs from mainstream Lisp in several important ways. In the detailed discussion below, we start with the aspects taken from Lisp, and then describe the (less important, we now think) differences. We then single out two differences that really mattered: paying attention to wording and dynamic scope.

Pavel Boytchev's meticulously researched *Logo Tree Project* [Boytchev 2014] lists just over 300 dialects of Logo, most but not all of which trace their ultimate ancestry to the 1968 BBN PDP-1 Logo. Some of the listed dialects are Logo in name only, restricted to a turtle graphics library. This section makes no effort to include the details of every dialect; we follow the two main branches of the tree: those descended from Terrapin Logo (based originally on the MIT Logo Group's version for the Apple II computer) and those descended from the Logo Computer Systems, Inc. (LCSI) dialect of Logo (starting with Apple Logo). Several people worked in both groups at different times.

3.1 The Design Process

HOPL papers are expected to discuss the design process, but due to the freewheeling nature of the Logo development community, there was no uniform or deliberate design process. (Again, though, in retrospect the most important aspects of the technical design of Logo were immediate consequences of the Lisp background of the designers, with no "process" at all.)

Even within a group, the process was different in different contexts. For example, when the LCSI team was designing Apple Logo, everyone was very conscious of the fact that as an official Apple product, this implementation would have high visibility, and so decisions made in this release would likely live forever, so each small detail was argued out.[3] The MIT PDP-11 Logo was taken as the fallback position whenever consensus could not be reached on a question. One of the most contentious questions was about the convention for naming predicate functions. (See Section 3.6.2.)

By contrast, the design process for Atari Logo was entirely constrained by the limited memory available. At a typical weekly design meeting, Brian Silverman†[4], who led the team that actually wrote the code, would list all the features that had been requested at the previous meeting along with the precise number of bytes required to implement them and the total number of available bytes after the previous week's work. The team would then argue about which features to leave out.

The design work through the 1970s happened largely on the PDP-11 and was focused on the core Logo language, settling details such as the assignment command (see Section 3.6.1). In the '80s and '90s, the work was driven partly by an explosion of competing microcomputers, along with creative development of special-purpose Logo versions to support a particular microworld, such as art or music, or in a few cases to introduce object oriented programming. The core language was largely fixed; relatively minor redesign happened largely in response to a desire to clean up rough edges.

The development of Logo at BBN and MIT was funded by NSF (the USA National Science Foundation). The commercial implementations at LCSI, Terrapin, and several other companies were self-funding.

3.2 Aspects of Logo Taken Unchanged From Lisp

Many aspects of Logo's design weren't the result of deliberate choices, but rather were taken for granted as part of Logo's Lisp heritage.

3.2.1 Interactive Development. An extremely important idea taken from Lisp and Telcomp is that Logo is *interactive,* not compile-then-run like Pascal or early BASIC, its chief competitors as a language for children in the early days.[5] BASIC was typically interpreted rather than compiled,

[3]Harvey, personal recollection, throughout this subsection except where otherwise noted.
[4]The dagger symbol † is used to indicate a reference to one of the authors of this paper.
[5]In those days, there was a clear-cut distinction between languages that were interpreted/interactive and those that were compiled/batch-processed. Today there are crossbred technologies such as JIT compilation, interpreted virtual machines, and so on. Our emphasis here is on the user experience, not the implementation.

but still emphasized writing a complete program before running anything. Interactivity, which is valuable for ease of program development even by adults, is essential in a language for children. In an interactive language you can type an instruction and see something happen immediately.

Logo also emphasizes the use of procedures, which might seem incompatible with immediate evaluation. But in using Logo, it's common for children to experiment interactively, and when a chunk of program is debugged, *then* encapsulate it in a named procedure.

At about the time Logo was developed, an ideology of *top-down programming* as the only "allowed" technique became popular among adult computer scientists: design first, top-down; only then, start coding. Logo's programming style was and is proudly antithetical to such disciplines [Harvey 1991]. Sherry Turkle and Seymour Papert introduced the use of the French word *bricolage* (tinkering) to label the Logo style of work, in which children experiment freely until they obtain a result that seems worth preserving as a procedure [Turkle and Papert 1990].

3.2.2 Late Name Binding. The goal of interactive programming requires the ability to make reference to a procedure that has not yet been written.

```
? to procedureA
> procedureB
> end
```

(The ? is the Logo prompt. The `to` command enters a mode in which a multi-line procedure body is entered, and the > is a special prompt character for the body lines. The word `end`, on a line by itself, signals the end of the procedure definition.)

In a compiled language, the entire program is in one or more files, and the compiler can read those files once to determine where in memory each procedure definition will be, and then again to translate each procedure *invocation* into that procedure's location. But in an interactive language such as Logo, the user enters the program at the keyboard, and if `procedureA` (in the example above) is defined before `procedureB`, the invocation can't be translated from the name to its location until `procedureA` is *run*. (Saying this another way, procedures must be defined before they can be called, but they shouldn't have to be defined in any particular order.)[6]

Much the same principle applies to variable names, locations, and values. Not only is Logo dynamically scoped (see Section 3.7), but variable declarations are executable:

```
? make "foo 87 ; a global variable
? to bar :flag
> if :flag [local "foo]
> make "foo 99
> end
? bar "true
? print :foo
87
? bar "false
? print :foo
99
```

[6] Again, modern compiler technology has changed this picture in some of its details. But the user's-eye view hasn't changed; whatever an implementation may do internally, the effect must be that names are bound to locations, and thence to procedure code, as late as possible. (We don't tell children about locations; in fact, the hope is that *all* this mechanism is invisible to the user.)

(This is not presented as an example of great Logo style!) Procedure bar takes one input, indicated by the formal parameter flag. In the body of bar, the local primitive command creates a local variable named foo. There is also a global variable foo because of the first make instruction. The instruction bar "true makes the if succeed, so a local foo is created, and that's the variable set to 99 by the following make instruction. When bar returns, the global foo still has the value 87. But the instruction bar "false makes the if fail, and so it's the global foo that's set to 99. The name foo in the make instruction can't be translated to a location until that instruction itself; looking for a binding on entry to bar isn't good enough.

3.2.3 Applicative Order. Like pretty much all but the purely functional programming languages, Logo uses *applicative order evaluation:* the argument expressions in a procedure call are evaluated before the procedure is called. This evaluation order allows for the situation in which the value of a variable used in an argument expression changes during the procedure call; since the argument value has already been calculated, it can't be affected by the change in the variable's value.

3.2.4 Eval/Apply. Every Lisp interpreter has largely the same structure: two main procedures with mutual recursion. The central procedure is eval, whose job is to turn an *expression* into a *value.* Eval actually takes two arguments, an expression and an *environment,* which is a collection of variable bindings. If the expression is a procedure call, eval calls apply, with two inputs: the procedure (not its name; turning names into values is part of eval's job) and a list of the actual argument values provided in the expression. The mutual recursion comes in the case of a user-defined procedure: apply then has to set up an expanded environment that includes bindings from the function's formal parameters to the provided actual arguments, and then it calls eval with the procedure body as the expression, and the newly created environment.

Interpreters for *most* programming language, not just Lisp dialects, have more or less this structure, with internal procedures similar to eval and apply. (Even in declarative languages, resolve and unify have a similar mutual recursion.) What makes Lisp special is that they (eval and apply) are made available to users of the language, generally under those names. This makes it possible for users to build language extensions, "little languages" with only a subset of Lisp's capabilities, or language experiments in which some aspect of Lisp semantics is changed.

The Logo version of eval is called run. It takes only one input, an instruction list or expression list. In the latter case, run outputs the value of the expression.

```
? run [print "hello print "goodbye]
hello
goodbye
? print run [sum 2 3]
5
```

In the first example, the input to run is an instruction list containing two print instructions, which are carried out in order. In the second example, the input is an expression (only one is allowed), and run outputs the result, 5, which is then used as the input to print.

Run takes only one input because environments are not first class in Logo. Since the goal is to turn children into mathematicians, not to turn them into computer scientists, teaching about environments isn't one of the goals. So run always uses the current environment. This means that run with a constant input is equivalent to the input itself:

```
? run [print "hello print "goodbye]
hello
goodbye
? print "hello print "goodbye
```

```
hello
goodbye
```

Run becomes useful when invoked with a variable input:

```
? to foreach :list :action
> if emptyp :list [stop]
> local "list.item
> make "list.item first :list
> run :action
> foreach butfirst :list :action
> end
? to ? ; defining a procedure named question-mark
> output :list.item
> end
? foreach [Yakko Wakko Dot] [print sentence "Hello, ?]
Hello, Yakko
Hello, Wakko
Hello, Dot
```

Foreach takes two inputs: a data list and an instruction list. Higher order procedures can be defined in several ways, but in this example we use the ? character as an implicit formal parameter for the instruction list. Sentence "Hello, ? makes a list whose elements are the constant text Hello, and one item from the data list. This implementation takes advantage of dynamic scope: The ? procedure isn't defined inside foreach, but it's invoked (via the run) from inside foreach. So it has access to foreach's local variable list.item, which has been given first :list as its value.

Not every version of Logo includes user-visible apply, but the ones that do provide it generally call it apply, with the same two inputs as in Lisp: a procedure and a list of actual argument values:

```
? print apply "sum [2 3 4]
9
```

You'll notice that actually the first input to apply in this example is the *name* of a procedure. Procedures aren't first class in most Logo dialects, especially not primitive procedures such as sum. But, because of dynamic scope, a procedure is just its text, so some dialects' versions of apply accept expression lists in place of procedures:

```
? print apply [? * ?] [5]
25
```

using the same ? technique as in foreach.

3.2.5 Lists. Traditionally, most languages used arrays—contiguous blocks of memory in which the address of an element can be computed from its index number—as the main data aggregation mechanism, because finding an arbitrary array element given its subscript takes constant time. But adding an element to a traditional array is slow and complicated; a new, larger block of memory must be allocated, and then every element copied from the old block into the new one.

Lisp traditionally used *linked lists* as its primary data aggregation mechanism (its *only* such mechanism in early versions). Abstractly, a list provides the primitive operations *first item* (car), *all but first item* (cdr), and *prepend one new item to a list* (cons), all in constant time. This structure was a good match to the use of recursion as the main control mechanism in Lisp:

```
(define (squares numbers)
  (if (null? numbers)
      '()
```

```
(cons (square (car numbers))
      (squares (cdr numbers)))))
```

As in the earlier discussion of binding (Section 3.2.2), modern data structures allow language implementors to have their cake and eat it too, using associative arrays (e.g., built from hash tables) on top of which traditional arrays and traditional lists can both be provided as abstractions.

It's worth noting that Logo also provides `last` and `butlast` functions so that users can think of lists as symmetric, and those were slow (linear time) for traditional linked lists. So efficiency was never really the main point. Rather, the list operations allow an aggregate to be traversed without the need for auxiliary index variables, simplifying the code.

Most versions of Logo do not allow list mutation, so the sharing of structure in traditional linked lists is not problematic.

The earliest versions of Logo had no data aggregation beyond text strings. But once the need for a general data aggregate was seen, Lisp-style lists were the obvious choice, because Logo inherits Lisp's emphasis on recursion as the main control structure. (See Section 3.3.3 for more details about the history of words and sentences in Logo.)

3.2.6 Recursion and Tail Call Elimination. The only primitive iteration facility in Logo is the `repeat` command, which carries out a given set of instructions a fixed number of times. For anything more subtle, such as the equivalent of a `for` loop or a `repeat-while` loop, Logo uses recursion. With young children, Logo teachers sometimes start with a very simple infinite recursion, so the first version of a procedure to draw a circle would be

```
to circle
forward 5
right 1
circle
end
```

The turtle would keep tracing over the same circle until the child manually stops the program. (A disadvantage of this approach, especially for older learners, is that it encourages the idea that a recursive call means "go back to the beginning," which can be hard to eradicate when a need for embedded recursion comes up.) With such heavy reliance on recursion as the idiom for iteration, it's very important that Logo implementations provide tail call elimination [Steele 1977] for commands. Not all implementations, however, provide tail call elimination for operations, which is technically much harder if you want tail call elimination to be invisible to users in error messages.

3.2.7 Automatic Memory Management. Today nearly every modern programming language handles its own memory management, rather than letting fallible human beings do it. But in the early days of Logo, automatic garbage collection was generally considered too slow, too complicated, and, in short, one of those ivory-tower Lisp ideas that nobody other than college professors would want. For Logo, though, it was always taken as given that we weren't going to make young children suffer through `malloc()` and `free()`, even on microcomputers with very limited memory.

3.2.8 Case Insensitivity. Although this has changed in recent years, at the time Logo was developed all dialects of Lisp were case-insensitive with respect to names of procedures, variables, and so on. That is still the rule in Logo, as it is in all natural languages that use the Latin alphabet. (Yes, a boy named Max might use the function `max` in his program, but if the function is the first word of a sentence, it's still not the boy.) Also, the move to case-sensitivity in later programming language design has been motivated largely by the convention of capitalizing the names of data types or object classes, and Logo does not require type declarations. Encouraging young programmers to use `max` and `Max` as the names of two different values is, we think, asking for bugs.

3.3 Aspects of Logo Changed From Lisp

3.3.1 Parentheses. Although this is a minor syntactic change in the overall picture of Logo, the first thing that Lisp users notice is that the language is not fully parenthesized. Lay people have often singled out parentheses as their reason not to program in Lisp. Logo avoids this controversial notation.

In Lisp, the notation for a procedure call is `(procedure arg arg ...)`. The parentheses around the call tell Lisp how many arguments are being supplied; even more important, since parentheses also delimit lists in Lisp, a Lisp program is already a data structure, so very little additional parsing is needed to implement Lisp in Lisp.

In Logo, most procedures, primitive as well as user-defined, have fixed arity (that is, a fixed number of arguments). Parentheses are therefore not needed, as long as the interpreter knows the arity of all procedures. For example, the parsing of

```
(print f g 1 2 3 4)
```

depends on the arity of `f` and `g`. At one extreme, if they both take no arguments, then all the numbers are arguments to `print`. If each of `f` and `g` take two arguments, then the statement is equivalent to

```
(print (f (g 1 2) 3) 4)
```

The fact that the interpreter must know all procedure arities to parse a program means that Logo, very unusually, is not a context-free language, so the usual automated parser generators aren't helpful. This is one reason why there's no BNF syntax as an appendix to this paper. But the caveat becomes important *to users* only in quite advanced Logo programs in which a procedure redefines other procedures that it uses, changing their arity, after the procedure body has been converted to an equivalent Lisp tree-structured program. In other words, for most Logo programmers it's not an issue at all.

Most early Logos had a `text` operation that takes a procedure name as input, and outputs the definition of the procedure in the form of a list of lists. But each of the sublists is an entire line of the definition, which may or may not correspond to one instruction.

There are a few variadic primitives, such as `sum` and `print`. These procedures have a *default* arity (two and one, respectively), and can be used with default arity without parentheses. For numbers of inputs other than the default, parentheses are used. Note that they are used Lisp-style, enclosing the procedure name and its inputs:

```
? print sum 7 8
15 ? print (sum 7 8 1)
16
? print (sum)
0
? (print 3 4)
3 4
```

Dylan is another parenthesis-free (but not context-sensitive) Lisp dialect [Shalit et al. 1996].

3.3.2 Commands and Operations. In early Lisp, every function returns a value.[7] Logo introduces the concept of *commands*, which are procedures that don't return a value. A call to a command, including its actual argument expressions, is an *instruction*, although Logo users often use the word "command" to mean either a command or an instruction. The body of a procedure must be a

[7] More recent dialects allow functions to return any number of values, including zero.

sequence of instructions, with no value-producing expressions except in the actual argument slots of a procedure call.

A procedure that does return a value was originally called an *operation*. In later LCSI Logo dialects the name was changed to *reporter*. A call to an operation, including its actual argument expressions, is called an *expression*. (Numbers, which are self-evaluating, are also expressions.) Since a procedure body must consist of instructions, not unattached expressions, there must be an `output` command that takes one argument and returns that value from the procedure in which this command appears.

```
? to square :n
> output product :n :n
> end
? print square 5
25
```

A note on the name `output`: In Logo documentation the arguments to a procedure are called "inputs." The model is essentially the "function machine" of introductory algebra classes: A procedure is a box with input hoppers on top and an output chute on the bottom. If this model is used in teaching Logo, the name `output` for the primitive that provides an output from its calling procedure is natural. Many Logo users have talked about this command as if it were a special aspect of the syntax, but it is *syntactically* just an ordinary procedure. Some later versions used `report` instead of `output` because many students confused `output` with `print`. The word "input," by the way, is used in the Logo literature to mean both formal parameter and actual argument. When it's necessary to make the distinction, one says "input name" or "input value," respectively. But most often the meaning is clear from the context, and a goal of Logo design is to encourage children to focus on the mathematics—the meaning of the procedure they're writing—rather than on the computer science.

There have been Logo dialects that allow an expression by itself as implicitly providing a return value from the procedure in which it appears, but they are out of the mainstream. In most dialects, expressions are not allowed even in the toplevel interpreter loop:

```
? sum 2 3
You don't say what to do with 5
? print sum 2 3
5
```

In other words, Logo uses a read-eval loop, not a read-eval-print loop as in other Lisp dialects. In part this avoids the need to display (or have a special case to avoid printing) something like `#[undefined]` as the "return value" of a command that has no natural return value.

Note that the error message "You don't say what to do with 5" does tell you the value you wanted, while also teaching that a command is expected. It's friendlier and more helpful than something like "Missing command" would be. This is an example of Logo designers' careful attention to wording. (See Section 3.6 for more on this.)

3.3.3 Words and Sentences. Although the word wasn't coined until more than one example existed, Logo was designed around the idea of *microworlds*, spaces for thinking about a particular problem domain with specific learning goals. The most famous Logo microworld is *turtle geometry*, a kind of differential geometry in which explorations can range from simple polygons for young children to non-Euclidean geometry and relativity for college students [Abelson and diSessa 1980].

But the first Logo microworld, strongly influencing its design, was English words and sentences (The name "Logo," suggested by Wallace Feurzieg, comes from λόγος, the Greek word for "word.")

This choice came from Papert's insight that elementary mathematics, the subject in which most previous experiments in programming for children focused, is something many children find difficult and painful. By contrast, every child can speak and understand English (or the child's native language, whatever it is) and most enjoy playing with words, e.g., in nonsense poetry or word games such as Boggle.

At first, words and sentences were represented as text strings in Logo, and indicated syntactically by the paired quotation marks that are still used for text strings in most programming languages. A text string was considered a sentence if it contained a space character, or a word otherwise. The selector `first`, for example, would output the first letter of a word, or the first word of a sentence; `butfirst` outputs all but the first letter, or all but the first word. This design was already revolutionary in abstracting away the task of searching for spaces in a string. But its obvious weakness is that it is impossible to represent a sentence of length one, so a procedure that `butfirst`s its way recursively through a sentence suddenly finds itself dealing with letters instead of with words. So the internal representation of sentences was changed to lists.

As a result, `first` and `butfirst` for a sentence are just the Lisp `car` and `cdr`. (However, Logo also provides selectors `last` and `butlast`, which take $O(n)$ time when applied to a singly linked Lisp-style list. This is a small illustration of the general principle that in a language for children efficiency isn't important; children almost always operate on small data sets. What matters is the naturalness of the user interface, and there's no reason to privilege the prefix of a word or sentence over its suffix. Think, as an example, about the algorithm to compute the plural of a word.)

At the same time the notation was changed to `"foo` for a word (a compromise, using the open-quote-only notation for a Lisp symbol but keeping the double-quote character familiar to children) and `[foo bar]` for a sentence (the square brackets both quote and delimit the sentence). Another advantage of double-quote instead of the Lisp single-quote to begin a symbol is that it doesn't raise syntactic hurdles when dealing with words such as `"I'm` or `"John's`. Logo later added the ability to nest square brackets, allowing lists of lists and thereby making Logo suitable for a study of data structures by older children.

Some programmers object to Logo's treatment of words and sentences because the same procedure, `first` for example, accepts both strings and lists as input. They would prefer two separate procedures, perhaps named `firstletter` and `firstword`, or perhaps `word.first` and `sentence.first` if the complainer likes object oriented programming. Our answer is that such complaints mean that the complainer is ignoring the data abstraction and thinking below the abstraction, at the underlying data types used in the implementation of Logo. Yes, strings and lists are quite different data types, and each should have its own selectors. But the abstract types "word" and "sentence" have a lot in common, and are part of the same linguistic microworld. It's completely natural that they should share selectors; that sharing also has the pedagogic virtue of teaching a single recursive pattern in which the first piece of some aggregate is separated from the rest of the aggregate, with the latter used as input to a recursive call.

Similarly, the `sentence` constructor feels very strange to a programmer who thinks in terms of primitive data types. It's *sort of* like `(apply append arguments)`, except that the domain of `append` is lists, while `sentence` accepts words as well as sentences (that is, strings as well as lists) as inputs. It's actually equivalent to

```
(define (sentence . args)
  (apply append
         (map (lambda (arg) (if (list? arg) arg (list arg)))
              args)))
```

But above the abstraction barrier, this is a perfectly natural behavior. "`Sentence` takes any number of words and sentences, and combines them into one big sentence."

3.3.4 Logo Is a Lisp-2. Lisp systems are divided into two categories: *Lisp-1* and *Lisp-2.* In a Lisp-1, there is only one namespace for all entities with names, including procedures among other kinds of data. In a Lisp-2, procedures have their own namespace, so the same symbol can name both a variable and an unrelated procedure.[8] This works in Common Lisp, a Lisp-2, because the full parenthesization makes it easy to distinguish the function being called (immediately after an open parenthesis) from argument expressions. Logo doesn't have that advantage. In the early days of Logo, all Lisps were Lisp-2s, so Logo fell into that position more or less automatically. But no mainstream Logo has been a Lisp-1, even in recent years, because too many Logo primitive constructors are named after the data type they construct: list, word, sentence. Logo programmers often use the name of a data type as a formal parameter:

```
? to plural :word
> output word :word "s
> end
? print plural "computer
computers
```

This wouldn't work if the parameter `word` shadowed the primitive procedure named `word`. NetLogo is a Lisp-1, and does not permit shadowing of primitive names.

Given the choice to be a Lisp-2, and without fully parenthesized expressions, Logo has to have a way to distinguish variable references from procedure calls. In the example above, `word` represents the procedure, and `:word` represents the variable. The colon, which is pronounced "dots" by Logo programmers, is not just a piece of syntax that precedes any use of a variable. It abbreviates an invocation of the primitive procedure `thing`, which takes a variable name as input and reports the value of that variable. Thus, `:word` is an abbreviation for `thing "word`. The solitary double-quote character serves the same purpose as the single-quote character in other Lisps. Because dots abbreviate thing-quote, they are used only in reading a variable, not in assigning to one. Some dialects allow multiple colons, in which case `::foo` abbreviates `thing thing "foo`.

3.3.5 Defining Procedures Programmatically. The operation `text` converts a defined procedure into a list structure that could be examined by a program:

```
? to squiral :size
> forward :size
> right 90
> squiral :size+2
> end

? show text "squiral
[[size] [forward :size] [right 90] [squiral :size+2]]
```

The command `define` converts a list structure in `text` format into a procedure:

```
? to ordinals :number :names
> if emptyp :names [stop]
> define (first :names) `[[list] [output item ,:number :list]]
> ordinals :number+1 butfirst :names
```

[8] Some Lisps, and some Logos, have a third namespace for *property lists.* For our purposes this detail is unimportant, and nobody calls them "Lisp-3s."

```
> end

? ordinals 2 [second third fourth fifth sixth seventh]
? print fourth [I want to hold your hand]
hold
```

The backquote (`) character in `ordinals` is a "quasi-quotation" operation, an idea borrowed from Lisp. It reports its list input, except that within the list, words or lists preceded by comma are *evaluated,* and the resulting value is used instead of the item. So, for example, the `fourth` procedure defined by `ordinals` is

```
to fourth :list
output item 4 :list
end
```

Some of the later LCSI dialects left out this capability, but it survives in Terrapin Logo and Berkeley Logo, among other dialects.

3.3.6 Infix Arithmetic. In an obvious concession to children's mathematical experience, Logo allows infix arithmetic `+ - * /`, in the usual notation, as well as relational operators <, =, and > (but not two-character <= etc., in most dialects). Parentheses are used for grouping if needed.

Infix arithmetic works fine by itself, and prefix functions work fine by themselves, but there is room for confusion when they are combined. The most usual combination is infix arithmetic providing inputs to a command:

```
forward :x+100
```

As a result, many Logo dialects make infix operators more tightly binding than prefix procedure calls. This may be a little confusing when using prefix numeric functions, as in `sin :x + 3`, which computes the sine of `:x+3`, perhaps not what the user intended. This is a small reflection, in Logo, of the impossible-to-remember 15 levels of precedence of the infix operators of the C language. But the much more common pitfall comes with the infix relational operators:

```
? print first "hello = "h
f
```

Users who write such instructions always expect to see `true` as the result, because they mean `(first "hello) = "h`. But instead Logo tests `"hello = "h`, gets the answer `false`, and computes the first letter of that result. Logo's Boolean values are simply the words `true` and `false`. Some more recent dialects ameliorate the problem with a more fine-tuned binding precedence: `+ - * /` are the tightest binding, then prefix *operations,* then infix `< = >`, and finally prefix commands.

3.4 No Standard

Some dialects use better binding precedence rules? Isn't there a standard? Well, no. At a Logo conference many years ago, some Logo developers attempted to write a Logo standard, but the *only* point of agreement was that to be called Logo, a language should support the word and sentence primitives: `first`, `butfirst`, `last`, `butlast`, `word`, and `sentence`.[9] The first four are selectors; the last two are variadic constructors. The selectors `butfirst` and `butlast` report a smaller value of the same type (word or sentence) as their input. `First` and `last` report a one-letter word if given a word as input, or a word if given a sentence. (See Section 3.3.3.)

The failure to agree on a standard has three roots.

[9]There are a few exceptions even to this. TurtleArt [Bontá et al. 2010] is one.

(1) Most importantly, there was a broad agreement that Logo is a perpetual research project, to be reinvented as needed for varied audiences, varied hardware capabilities, and so on. A standard would be constraining. As one example, there have been several projects adding object oriented programming to Logo, with very different approaches.
(2) Second, even the commercial Logo developers have created modified versions; perhaps the most dramatic such change to the language was LCSI's LogoWriter, which embedded a core Logo in a rudimentary word processor, just as Richard Stallman embedded a Lisp interpreter in Emacs. Other examples include versions modified to support various robotics hardware. In the early days, with limited microcomputer memory, adding music support on the Apple II meant removing turtle support, for example.
(3) Finally, neither side convinced the other in the Terrapin/LCSI split, coming up next.

Lisp, too, had no standard for its first 20 years; the hacker culture did not lend itself to freezing out further experiments by premature standardization.

3.5 Special Forms: the Terrapin/LCSI Split

In Lisp, almost everything is done with a procedure call, which looks like this:

```
(proc arg1 arg2 ...)
```

Each of the space-separated parts of this expression is evaluated; the first must have a procedure (a function) as its value.[10] Once all the expressions have been evaluated, the procedure is called with the values of the argument expressions as its arguments.

In this section we focus on the "*almost* everything." Some things can't readily be done as function calls. The classic example is the `if` conditional. In a recursive procedure such as

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```
[11]

If the argument `n` is zero, the function should return 1, and *the last input to* `if`*, the last line of the definition above, must not be evaluated.* Otherwise there will be an infinite recursion trying to compute factorial of -1, factorial of -2, and so on. So `if` has a special evaluation rule: Its subexpressions are *not* evaluated before `if` does its job. Lisp hides this issue by using the same parenthesis notation for `if` as for procedure calls. An expression starting with the word `if` is called a *special form.* ("Form" just means "expression" in this context.) `Define`, in the Scheme example above, is another special form; the `(factorial n)` part shows what a call to `factorial` will look like, but it can't actually call `factorial` when it hasn't been defined yet. Again, the `define` expression *looks like* a procedure call, but isn't one. Part of learning Lisp is learning to recognize the special form names (*keywords*).

What about Logo? Like Lisp, it's almost all done with procedure calls. But there have been arguments about the exceptions.

The earliest versions of Logo freely included ad hoc syntax for each important control structure:

```
if :x<0 then right 90 else left 90
```

This notation was an effort to look like English rather than to follow a uniform notation. In *really* early versions there was an interactive protocol for assigning a value to a variable:

```
? make
name: foo
```

[10] In a Lisp-2, that first expression is evaluated using the procedure namespace rather than the variable namespace.
[11] We are showing Lisp code in the Scheme dialect.

```
thing: 87
```

In this example what the user types is **`in boldface`**.

In 1981, two different commercial Logo versions were released for the Apple II: first Terrapin Logo, essentially the MIT Logo Group's Apple II dialect, and later Apple Logo, from LCSI, also based on the Logo Group work, but not on the specific MIT code for the Apple. Most later versions, even ones not produced by these two companies, can be characterized as Terrapin-like or as LCSI-like. The biggest difference is that the LCSI designers decided that Logo should have no special forms, but should instead do everything through procedure calls.

Why make syntactic uniformity a goal? The `if.then.else` notation was easy for beginners, but, like the Lisp notation for special forms, it hides the fact of delayed evaluation of the "arguments" to `if` (in quotes because a special form isn't actually a procedure call so it doesn't exactly have arguments). The LCSI designers felt that that hiding might be a contributing factor to the widely believed difficulty of understanding recursion. Making the delayed evaluation visible in the notation might help. A second reason was the popularity among Logo developers of the exercise of writing a Logo interpreter in Logo; a completely uniform evaluation rule simplifies that exercise. In retrospect, those reasons can be seen as an implicit decision to aim Logo at somewhat older children than were taught in earlier Logo research.

In LCSI Logo, the `if` instruction above becomes a call to the `if` *procedure,* which takes three inputs:

```
if :x<0 [right 90] [left 90]
```

The first input is a Boolean value, reported by the `<` procedure. The second and third inputs are instruction lists, only one of which will be evaluated by `if` depending on the value of the first input. Square brackets both delimit and quote a list, so `[right 90]` is equivalent to `'(right 90)` in Lisp. The entire expression is therefore equivalent to

```
(if (< x 0) '(right 90) '(left 90))
```

in Lisp notation. How can Logo represent a thunk as merely the text of its body, without wrapping it in a lambda expression to capture its closure? See Dynamic Scope below (Section 3.7).

The use of quoted instruction lists didn't start with LCSI. From the beginning, the main looping facility has been

```
repeat 4 [forward 100 right 90]
```

with the instructions to be repeated inside square brackets. Some Logo developers, however, treat the square brackets as syntax in a repeat special form; in those versions of Logo, you can't say

```
? make "actions [forward 100 right 90]
? repeat 4 :actions
```

as you can in LCSI versions.

There is one special form in all versions of Logo, even the LCSI ones, namely `to`, the command to define a procedure:

```
? to plural :word
> output word :word "s
> end
? print plural "computer
computers
```

The `to` command doesn't evaluate its inputs, and it enters an interactive editor (in some versions, a display editor rather than the Teletype-era one shown here) in which the user enters the body of the procedure. The keyword `end` terminates the body; it is part of the syntax, and is not understood outside of the `to` interaction. The first input to `to` is the name of the procedure being defined; any

additional inputs are formal parameters. The colon on the title line (`to plural :word`) is not an invocation of `thing`; it's there just to remind the user that this is a variable name. Different dialects accept one or more of these alternative notations:

```
to plural :word
to plural "word
to plural word
```

The theory behind the `"word` notation is that a formal parameter is more like an assignment to a variable than like looking up its value, and the notation will remind the user of `make "foo 87`. The quotation mark in the `make` instruction isn't a special syntax; it indicates that the variable's (quoted) name is the first input to `make`. Scheme's `set!` is a special form that accepts *only* an implicitly quoted name, but Logo allows the variable name to be computed:

```
make (word "foo :index) 100+:index
```

(The parentheses are here only for readability for humans; Logo would parse the instruction correctly without them.) The restriction in Scheme allows very efficient compiled code, but Logo designers almost always put expressiveness above efficiency, since Logo programs are generally small and therefore fast anyway.

The semantics of `make` is that if a variable with the specified name is in scope (including a global variable), a new value is assigned to that variable. If not, a new *global* variable is created with the specified name and value. If you want to make a new local variable, you use the `local` command and then `make`. The reason for the choice to default to global for newly created variables is based on the fact that young children, Logo's first audience, rarely use explicitly created locals; the only local variables in most Logo programs are the parameters of procedures, and it's unusual to change their values. To a professional programmer this will feel like bad hygiene, but Logo programs tend to be too small to have conflicting uses of the same global name, provided that students are taught not to call variables `x`. As always, the focus is on the mathematics, not on the computer science.

3.5.1 Tokenization. In all versions of Logo, the space character, square brackets, and parentheses delimit tokens. In older Terrapin-style dialects, the infix arithmetic and relational operators also delimit tokens (and are tokens themselves). In newer LCSI-style dialects, as well as in NetLogo, the infix operators do not delimit tokens; to use them as operators, they must be surrounded by spaces. For example, 2+2 is a token, whereas 2 + 2 is 4.

The reason for the LCSI design is clearest when considering the hyphen/minus character. Consider this list:

```
[555-1212 868-9827]
```

Should this be parsed as a list of two words, or of six words? If you live in the United States, it's obviously a list of two telephone numbers, not subtraction problems. On the other hand, if the context is

```
print product 555-1212 868-9827
```

then you'd like the hyphens to be taken as minus signs, for a total of eight tokens on this line.

Some more recent Logo dialects do context-sensitive tokenization. Sometimes this requires retokenizing a line as the context changes, such as the use of `run` (Logo's name for `eval`) in this example:

```
? make "test [555-1212 868-9827]
? print count :test
2
? print first :test   ; prints an eight-letter word with the hyphen
```

```
555-1212              ; as the third letter
? run sentence [print product] :test  ; Now the hyphens are treated
5886063                               ; as separate tokens (minus signs).
? print product 555-1212 868-9827     ; an equivalent instruction
5886063
```

3.6 Paying Attention to Wording

Much design attention in Logo has gone into English wording, including primitive names and error messages, things that aren't traditionally considered noteworthy in language design. For example, it's the 21st Century, and there are *still* programming languages that use the equal sign character as the assignment operator.

3.6.1 Assignment. What does it mean to associate a value with a variable name? Pedagogically, there are (at least) two different answers, and so different versions of Logo have used different names for the variable assignment primitive. One answer, most appropriate for variables whose value is not going to change during the running of the program, is that assignment *gives a name to a constant value:*

```
name 3.141592654 "pi
```

The other answer, more appropriate for variables that vary, such as the index variable in a loop, is that assignment *puts a value in a box:*

```
make "index :index+1
```

Different Logo dialects have used each of these, and some dialects offer both.

3.6.2 Predicates. All Lisp dialects use some special notation for predicate functions (ones that report a Boolean value), so that the operation that means "make me a list containing these values" can be distinguished from "is this value a list?" The former is the `list` operation. The latter was `listp` in traditional Lisps ("p" for "predicate'), but became `list?` in Scheme. This may seem like a trivial issue, but it led to heated debates among Logo designers. One cited advantage of the "p" notation is in reading programs out loud; "list pee" is audibly different from "list." On the other hand, question mark enthusiasts argued that there are non-predicate procedures whose names happen to end in "p"; they asked sarcastically whether `stop` should be pronounced "stow pee." The advantage of the question mark is that it's visually unambiguous.

Another argument for "p" was that *every* operation is a question, not just the yes-or-no questions: "How much is 2+3?"[12] Logo dialects have more or less followed the Lisp history: Most early dialects use "p" and most late dialects use "?" but there are exceptions both ways. The very first Logo dialect used "q" to indicate predicates; this arguably has the virtues of both of the other options, because it's clearly pronounceable as "list queue" but is unlikely to come at the end of a word adventitiously. It's not clear why "q" didn't remain standard (except in British dialects); the designers of Apple Logo, for example, were aware of the possibility but decided against it.

No known Logo dialect, though, has adopted the Scheme convention of "!" for mutators. Most Logo dialects did not allow list mutation at all, because that choice allows two lists to share storage without the risk that changing one will surprise the user by changing the other. There is a Logo tradition of starting the names of "dangerous" primitives with "." (e.g., `.deposit` and `.examine` in microcomputer Logo dialects that allow direct access to memory), and some dialects that allow mutation of list pointers call the relevant primitives `.setfirst` and `.setbutfirst` because of the possibility of creating circular structures or mutating shared lists.

[12]Recall that in Logo terminology an operation is a procedure that returns a value, as opposed to a command.

3.6.3 Error Messages. Even the Scheme standard, which requires that implementations provide tail call elimination, is silent about the text of error messages. Logo's developers, however, considered it unacceptable for a child to see a typical programming language error message such as

```
RangeError
Invalid code point NaN
```

People new to programming often have a fear of errors, especially if they are learning in a school, where errors are typically punished with bad grades. It's common for beginners to react to a single error in a large program by deleting the entire thing and starting over. Logo teachers struggle to convince students to "love their bugs." An understandable error message, especially one that suggests how to solve the problem, makes the error at least somewhat less frightening.

In some recent programming languages, such as Elm, Go, and Rust, there has been an awareness of *programming language ergonomics,* including error messages. Newer programming environments for older languages, such as C++, also attempt to replace problem-centered messages with *solution messages.* Logo began this effort in the 1960s.

A story Paul Goldenberg recounts from the early days of Logo is that one day a child typed the command `love` and got the response

```
love has no meaning.
```

The child tearfully reported this to Paul. The next day that error message was changed to

```
I don't know how  to love
```

in which the extra space before "to" hints at the command needed to solve the problem. But that didn't end the discussion; there are still arguments about whether "I don't know how" gives a child a misleading idea that the computer is conscious, but the alternative "You haven't said how" seems to blame, and therefore discourage, the child. Again, Logo dialects have differed on this issue. Perhaps the argument was more intense in Logo's early days, when it was unusual for children to have access to interactive computers, and so children were more likely to develop wrong ideas about their capabilities. But speech-understanding systems such as Siri, Google Assistant, and Alexa are again encouraging children to imagine programming as a conversation with an intelligent entity.

3.7 Dynamic Scope

Probably the design decision in Logo that may be the most surprising among modern programming language experts is the use of dynamic scope. The first Lisp interpreters used dynamic scope, although John McCarthy later said that that was simply a mistake—that Lisp should have followed the rules of lambda calculus. A consistent (whether interpreted or compiled) use of lexical scope was one of the great contributions of Scheme to the Lisp community. Nevertheless, we shall argue that retaining dynamic scope was the right choice for a language intended for children rather than for professionals. (Even in languages for adults, there have been design features that regain some of the virtues of dynamic scope, such as implicit parameters in Haskell and `fluid-let` in Scheme.)

3.7.1 Arguments for Lexical Scope. There are three main reasons to prefer lexical scope:

- A compiler can translate variable names to memory addresses (or to a short instruction sequence to find the relevant stack frame and the offset within the frame) at compile time, providing greatly improved runtime performance.
- Referential transparency: It is visually clear to the programmer, reading the program source code, which binding of a name applies to each specific reference. Names cannot be "captured"

by a calling procedure. Referential transparency is also important to compilers, because it allows easier inference about the program, for proofs of correctness or for optimization.
- If your language also has lambda or the ability to nest procedure definitions, you can use persistent local state variables to create an object oriented programming framework.

We shall argue that none of these is relevant to the expected uses of Logo. One Logo dialect, Object Logo, had the intent of becoming a system programming language for the Apple Macintosh, and so its designers went against Logo tradition and made it lexically scoped. But Apple wasn't persuaded.

3.7.2 Arguments for Dynamic Scope. We emphasize that the goal of Logo has always been to create a powerful programming environment for young children. No one suggests that the following arguments apply to large-scale professional programming.

- We argue that dynamic scope is what naïve users expect: If a variable exists (and hasn't been shadowed explicitly), it's available for use.
- There is no need to capture the closure (the defining environment) of a subprocedure, so *a procedure is simply its text.* (This is why there is no need for a lambda operator in Logo.) This means the body of a procedure, which is an instruction list, can simply be `run` (Logo's name for `eval`). That allows for an easy-to-explain implementation of higher order functions, albeit with the possibility of name capture.
- As for name capture and referential transparency, those are concerns for large teams of programmers, or for aggressive optimization of production programs. Logo teachers can just advise children not to use the same name for two purposes.
- When an error is signaled, the evaluator can pause evaluation, and the user can use Logo itself as the debugging language; nothing like gdb, with special commands to navigate the procedure call stack, is required. All possibly relevant variables are accessible for inspection. We discuss this point further in the next subsection.
- The flip side of name capture is that a toplevel procedure can create a "semi-global" variable that's available to all its subprocedures (and sub-subprocedures, etc.) but disappears when the program run is finished. This is particularly convenient if a child designs a picture with several components (although this practice can make the subprocedures harder to test or re-use).

```
to house :size
frame
door
windows
people
trees
end
```

The subprocedures `frame`, etc., take no explicit arguments, but can use `:size` to refer to the input given to `house`.

3.8 Technical Support for Debugging

The practice of debugging was a central part of Logo *culture,* especially in the early days. See Section 4.3.3 for a longer discussion of the culture of debugging. The emphasis was much more on debugging a child's ideas than on debugging the actual code, so debugging technology (breakpoints and so on) were relatively unimportant, and were missing altogether in some of the later LCSI

implementations. But every Logo implementation that did provide debugging tools used the ones described here, starting at least as early as the PDP-11 Logo at MIT.

21st Century programming is largely done using an Integrated Development Environment (IDE) that includes syntax-aware editing as well as breakpoints, single stepping, stack traces, and other debugging tools. Even after IDEs were invented, Logo never had one. IDE commands are typically single keystrokes or mouse clicks, an entirely different language from the programming language they support. This means that a beginner must, in effect, learn two very different formal languages at once, which would be a formidable barrier to children. An important design decision was that the debugging language for Logo should be Logo itself. This is possible only because of the interactive style (the read-eval loop) that Logo inherited from Lisp.

3.8.1 Pausing Execution. The most important technical feature to support debugging Logo in Logo is the `pause` primitive, which can be used as a command or as an operation. It displays a Logo question-mark prompt and opens a read-eval loop, *in the environment in which the* `pause` *occurs.* This is how dynamic scope enters the debugging picture. However deep in procedure calls you may be, the variables of all those waiting procedures are available for inspection or modification. In this explanation for programming language experts, we are talking about scoping rules and environment nesting, but a child doesn't need any of that vocabulary. Because of dynamic scope, all the variables that exist are visible, just as a non-expert would expect, unless the child has used the same name for two different variables, which quickly becomes apparent.

The Logo user can explicitly edit a `pause` into a procedure as a breakpoint, but many versions of Logo have had an even more powerful debugging aid: the ability to set a "pause on error" flag or, more generally, to set an arbitrary sequence of instructions to be carried out in the event of an error. The latter can be used to pause just in specific cases:

```
make "erract [if errnum=11 [pause]]
```

(`Erract` is for "error action"; `errnum` (the actual error information format varies considerably between versions) reports which kind of error happened.) If a pause happens, the user sees the Logo error message, including information about exactly what line in what procedure caused the error, and then a question-mark prompt. If exploring the environment at the location of the error doesn't reveal the problem, the user can then work backward, using explicit `pause` commands inserted in the relevant procedure(s).

A third way to pause execution is by typing an interrupt character, different for different computers and operating systems.

The `continue` command is used to resume execution. The command allows an optional input, which is needed when `pause` is used as as operation, or, equivalently, if a primitive operation throws an error that leads to a pause. In that case, the input to `continue` becomes the value of the `pause` or of the operation that gave the error.

3.8.2 Tracing and Stepping. The `trace` command takes a procedure name or a list of procedure names as input. Whenever any traced procedure is called, Logo prints a message on entry to the procedure, including the actual argument values of this call; when the procedure outputs or stops, Logo prints a message including the output value if any. The entry and exit lines are shown indented by a number of spaces equal to the depth of the call stack. The `untrace` command takes a name or list of names and disables tracing of the named procedures.

```
to launch
countdown 3
print "blastoff!
end
```

```
to countdown :number
if :number=0 [stop]
print :number
countdown :number-1
end

? trace [launch countdown]
? launch
( launch )
 ( countdown 3 )
3
  ( countdown 2 )
2
   ( countdown 1 )
1
    ( countdown 0 )
    countdown stops
   countdown stops
  countdown stops
 countdown stops
blastoff!
launch stops
?
```

This trace output is actually somewhat of a lie, in a Logo implementation that includes tail call elimination. There's only one little person (that is, one stack frame) handling `countdown` when it's not being traced. And there have been Logo implementations in which "countdown stops" would only be printed once for this example. But most Logo dialects that implement `trace` give a result like the one shown here, for two reasons. The more important reason is that we don't want to expose to kids complexities that are in the interpreter only for efficiency reasons. The secondary reason is that, in fact, tracing a procedure makes it *effectively* non-tail recursive. It's as if the procedure were

```
to countdown :number
print (list "( "countdown :number ") )
if :number=0 [stop]
print :number
countdown :number-1
print [countdown stops]
end
```

That final `print` line comes after the recursive call, so there's still work to be done when the recursive call finishes.

The `step` command similarly takes a procedure name or list of procedure names as input. Whenever a stepped procedure is called, Logo types each line of the procedure body (i.e., prints the line but without a newline at the end) followed by a >>> prompt, and then waits for the user to type a newline character before carrying out the instructions on that line. The user can instead type the system pause character, to examine variables or even modify the code of the procedure before continuing. It's noteworthy that the unit of stepping is a line rather than an instruction. Logo allows multiple instructions on a line, but experienced Logo programmers use that capability

with groups of instructions that form a meaningful unit in the programmer's thought. The `unstep` command is analogous to `untrace`.

```
? untrace [launch countdown]
? step [launch countdown]
? launch
[countdown 3] >>>
[if :number=0 [stop]] >>>
[print :number] >>>
3
[countdown :number-1] >>>
[if :number=0 [stop]] >>>
[print :number] >>>
2
[countdown :number-1] >>>
[if :number=0 [stop]] >>>
[print :number] >>>
1
[countdown :number-1] >>>
[if :number=0 [stop]] >>>
[print "blastoff!] >>>
blastoff!
?
```

3.8.3 `Print` *as a Debugging Aid.* In an interactive pause, the user wants to find out why an error or other unexpected behavior happened. This investigation mostly happens by examining the values of variables. In an IDE, there's generally a special notation to allow watching a variable dynamically, but in Logo, we use the same `print` command that programmers use to interact with the user of a working program.

`Print` instructions can also be inserted in a program to provide debugging information without pausing. This is everyone's first debugging technique, precisely because it uses a mechanism that the programmer already understands, but in some circles it's disparaged as unsophisticated. It's true that a programmer who inserts print statements in the code has to remember to clean up after debugging by removing them. But the Logo approach is to prefer this minor chore over the intellectual strain of learning a special debugging language in parallel with learning Logo itself.

4 LOGO BEFORE PERSONAL COMPUTERS

In this paper the history of Logo is divided into two periods. The dividing line is around 1980. By then the personal computer revolution had started and Papert's book *Mindstorms* [Papert 1980] came out. Logo moved from time-shared computers to personal ones. This section highlights some of the key events and ideas from before 1980.

Logo was invented in 1966 primarily by Papert and Feurzeig at BBN in Cambridge, Massachusetts. While research continued at BBN under the direction of Feurzeig, in 1969 the Logo Group was started at the MIT AI Lab, a lab that Papert co-directed with Minsky. In the early 1970s a Logo research and development group was formed at University of Edinburgh's AI Department.

We begin this section with a first-person account from Cynthia Solomon, the only one of the present authors who was involved with Logo from its very beginning.[13] There are first-person accounts from some other authors later in the paper and later in the history.

4.1 Reflections from Cynthia Solomon

In 1966 Seymour Papert consulted for Wally Feurzeig, head of the education group at BBN. Wally had a school project involving five schools. As part of their math class the students were learning a version of Telcomp, an algebraic programming language similar to BASIC. Seymour visited the classrooms and was struck by the absurdity of using an algebraic programming language to help students to understand algebra. Seymour decided that what was needed was a programming language specifically designed for children.

During that academic year Seymour talked about this new language mostly with Daniel (Danny) Bobrow (head of BBN's AI Group) and Wally Feurzeig. By the end of summer 1966 Seymour specified this new language and presented it to a small group. At that time I was a member of Wally's group at BBN. The group, Seymour, Wally, Danny, Dick Grant (an MIT student) and me, thought of the new language as "Baby Lisp" or "Lisp without parentheses." Danny immediately started implementing the language in Lisp on BBN's time-shared SDS 940. He gave it over to me and later Dick took it over.

The guiding idea was that Logo would be a language for playing with words and sentences. Seymour said: "What is one of the chief activities of children? Why, playing with words and sentences." Danny at first called the language Ghost, after the children's word game. That name didn't satisfy Wally who came up with a much more suitable name, Logo, from the Greek $\lambda\acute{o}\gamma o\varsigma$, word.

By summer 1967, there was a working Logo written in Lisp and running on BBN's SDS-940. It was tried out with 10 year old children at the Hanscom School, Lincoln, Massachusetts. Seymour taught, Wally and I were observers. Before and after each class Seymour and I held debriefing and debugging sessions.

Fig. 4. Seymour Papert, Cynthia Solomon, Danny Bobrow, and Wally Feurzeig (Papert and Solomon courtesy of Frank Frazier; Bobrow courtesy of MIT Museum; Feurzeig by Wikimedia user Daderot, 2006, CC0)

[13] Anecdotes not otherwise credited throughout Section 4 are based on Solomon's personal recollections.

A result of this experience was a totally revised Logo ready for implementation on a time-shared Digital Equipment Corporation (DEC) PDP-1, also at BBN. This system was ready for the 1968-69 school year.

In the fall of 1968 12 Teletype terminals were put in a classroom at Muzzey Junior High School in Lexington, Massachusetts. They were connected from the school to the Logo time-shared DEC PDP-1 at BBN. The students were 12 year olds (seventh graders) who clustered in the average school performance range. Instead of their regular seventh grade math course they were taking a year-long computer math course.

The activities included writing programs for Pig Latin, Twenty Questions (Guess My Number), Nim, sentence generators, math teaching programs, and story-telling. These were projects that followed a style of programming encouraged by Logo: projects that could be divided into small procedures, which could be debugged separately from the whole. This emphasized the importance of giving things names so that you can talk about them.

Alan Kay visited in the Spring of 1969 and noted:

> …I visited Seymour Papert, Wally Feurzeig, and Cynthia Solomon to see the Logo classroom experience in the Lexington schools. This was a revelation! And was much more important to me than the metaphors of "tools" and "vehicles" that were central to the ARPA way of characterizing its vision. This was more like the "environment of powerful epistemology" of Montessori, the "environment of media" of McLuhan, and even more striking: it evoked the invention of the printing press and all that it brought. This was not just "augmenting human intellect," but the "early shaping of human intellect." This was a "cosmic service idea." [Kay 2013].

Fig. 5. Students of Muzzey Junior High School using Teletype terminals (courtesy of Frank Frazier, circa 1968)

But that is not how things were at the start. For the first month or so, this course was taught by a math teacher with very little programming experience. She concentrated on the meaning and syntax of Logo operations. Children worked on problems like this one:

```
What is the value of
LAST OF FIRST OF BUTFIRST OF "THE GOOD CAT"
Answer: "D"
```

This was not an ideal approach and the atmosphere in the classroom reflected that fact. We knew that it is possible to write exciting programs with a small subset of Logo commands and with little awareness that operations can be composed or chained. We believed that, at least for younger students, initial exposure to Logo should minimize emphasis on syntax and the variety of operations. Instead. concentrating on concepts provided an immediate pay-off in programming outcomes that led to excitement for the children.

Realizing we had not adequately prepared the teacher Seymour and I stepped in as the teachers. We focused on children creating projects.

Before the Muzzey experiment, we had some experience in teaching children to program in Logo and other languages. Working in an actual school, however, forced us to recognize some misleading facets of our previous experience. Most important was that our teaching method depended heavily on what came to be referred to as "computer culture." We had at our fingertips many examples, analogies, ways of looking at programming, jokes, turns of phrase, and other useful aids for establishing a lively interaction with the children. However confronting a class daily was new; our previous experience teaching programming to children had not prepared us for it.

One early favorite project was to write a program to translate English into Pig Latin, a child's game in which you strip consonants off the front of the word and attach them to the end, followed by "ay," e.g., "school" to "oolschay." The point of Pig Latin was to communicate with other children in a way that adults wouldn't understand. Many heuristics can be applied to the problem of writing a translation program. For example, it is natural to plan the solution by subdividing the task. You can think about how to obtain a sentence-translating procedure, call it `PIG`, from a word-translating procedure, call it `PIGWORD`, without worrying, for the moment, about how `PIGWORD` itself works. Similarly, when working on `PIGWORD` one need not concern oneself with how it is to be incorporated into `PIG`. Unlike an algebra word problem or a real-life problem, the subdivision into component parts is particularly clear. The pedagogical purpose is for children to obtain from these simple cases good planning habits and clear paradigms for their strategies. The hope being that they will later be able to generalize these strategies and apply them to other problems.

```
TO PIG :SENTENCE
IF EMPTY? :SENTENCE [OUTPUT []]
OUTPUT SENTENCE (PIGWORD FIRST :SENTENCE) (PIG BUTFIRST :SENTENCE)
END

TO PIGWORD :WORD
IF VOWEL? FIRST :WORD [OUTPUT WORD :WORD "AY]
OUTPUT PIGWORD (WORD BUTFIRST :WORD FIRST :WORD)
END

TO VOWEL? :LETTER
OUTPUT MEMBERP :LETTER "AEIOU
END

? PRINT PIG [THE RAIN IN SPAIN]
ETHAY AINRAY INAY AINSPAY
```

4.2 Logo Goes to the MIT AI Lab in 1969

The BBN work set the stage for what was to become a powerful, flexible, and usable programming language for children. It highlighted the ease with which learners can quickly become experts.

In the summer of 1969 Solomon left BBN to join Papert at the MIT AI Lab and the Logo Group was formed.

4.2.1 The Birth of the Logo Turtle 1969/70. The Logo turtle idea appeared as the Muzzey experience was coming to an end. The students did well with a diversity of programming projects focusing on words and sentences, with numbers being special words. But based on work with Minsky and Piaget, Papert was again struck by an expanded view of domains for children's explorations. What if there were a concrete object to play with? A device that could be controlled by a child? In England William Grey Walter had built Elmer and Elsie, two automatons, that could walk around a space. He called them tortoises. Picking up on Elsie and Elmer, the Logo creatures were dubbed turtles. The early turtles were tethered. They soon had a counterpart living on a graphics screen, initially known as display turtles.

4.2.2 Logo Activities at the MIT AI Lab. Thanks to the talent and cooperation of Marvin Minsky and others at the MIT AI lab, the Logo group was able to develop two floor turtles (a large yellow canister put together by Russell Noftsker and John Roe and a plexiglass turtle built by Tom Callahan), a display turtle (programmed by Hal Abelson), a music box (built by Marvin Minsky), and a version of Logo running on the Lab's time-shared PDP-10 (with the chief programmer being Ron Lebel). Many new ideas for expanding procedural thinking and debugging were also developed as the Logo Group taught children to walk on stilts, juggle, and more. This work spawned lots of ideas that became part of the paper "Twenty Things to do with a Computer" [Papert and Solomon 1971].

On April 11, 1970, the first public Logo symposium, called Teaching Children Thinking, was held at MIT and attended by over 700 people.

Text from the poster:
"Is it possible to make a more direct attack on teaching children to think?
Seymour Papert will survey the state of knowledge and describe an unorthodox plan for elementary education.

Marvin Minsky will direct a discussion with Robert Davis, Allen Newell and Patrick Suppes. The goals are to disseminate information, to provoke discussion and to recruit collaboration."

Fig. 6. Poster of the Logo Symposium: Teaching Children Thinking (Cynthia Solomon collection)

For the 1970-71 school year Papert and Solomon taught a group of 5th graders (10 year olds) at the Bridge School in Lexington (Massachusetts, USA).

4.2.3 1970: Turtles and Turtle Geometry. Turtles gave rise to Turtle Geometry, which is a local geometry focused on the turtle's position and heading. The basic turtle commands are very simple. They consist of six primitives: `FORWARD`, `BACK`, `RIGHT`, `LEFT`, `PENUP`, and `PENDOWN`. The turtle gives children immediate feedback as it responds to their commands. Figure 8 illustrates the use of the main turtle commands.

Exploring the turtle's behavior led to an important theorem: the Total Turtle Trip Theorem. If the turtle repeatedly moves a fixed amount and then turns a fixed amount it will eventually either return to its starting state or be bounded by two parallel lines. One can envision this by playing

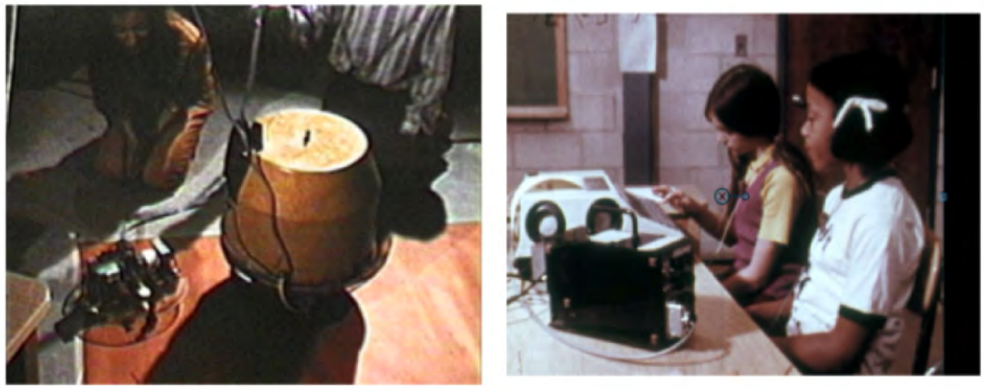

Fig. 7. Floor Turtles (left) and Music Box (right). (Cynthia Solomon collection)

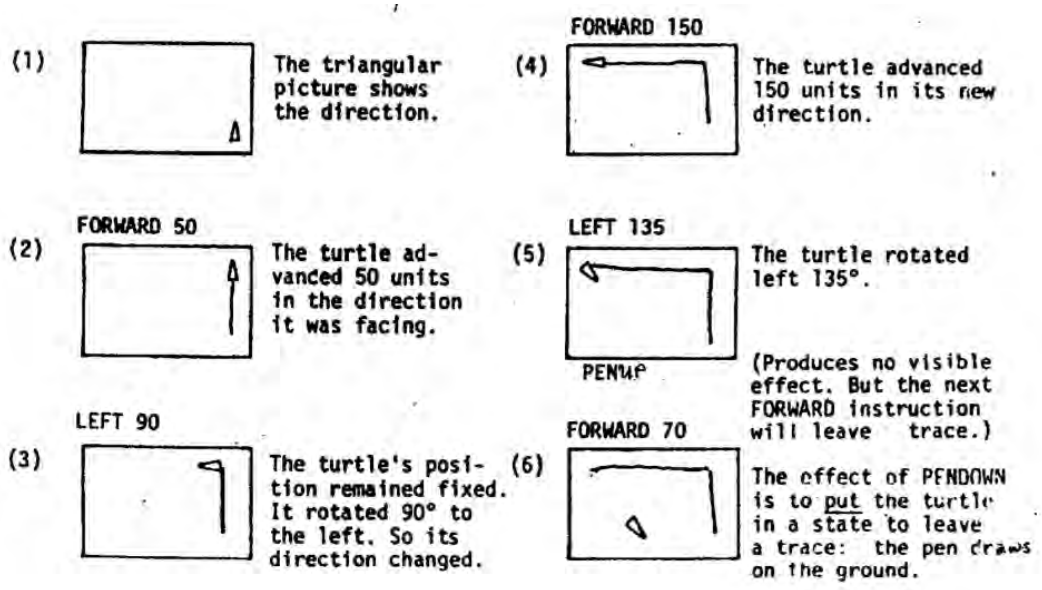

Fig. 8. Basic Turtle Commands (Cynthia Solomon collection)

with the POLY procedure. In its simplest form, POLY has two inputs: step and angle. In Logo, this is written

```
to POLY :step :angle
forward :step
left :angle
POLY :step :angle
end
```

The pictures in Figure 9 show the effect of invoking this procedure with different inputs: the first input, step, specifies the length of a side; the second input specifies the angle.

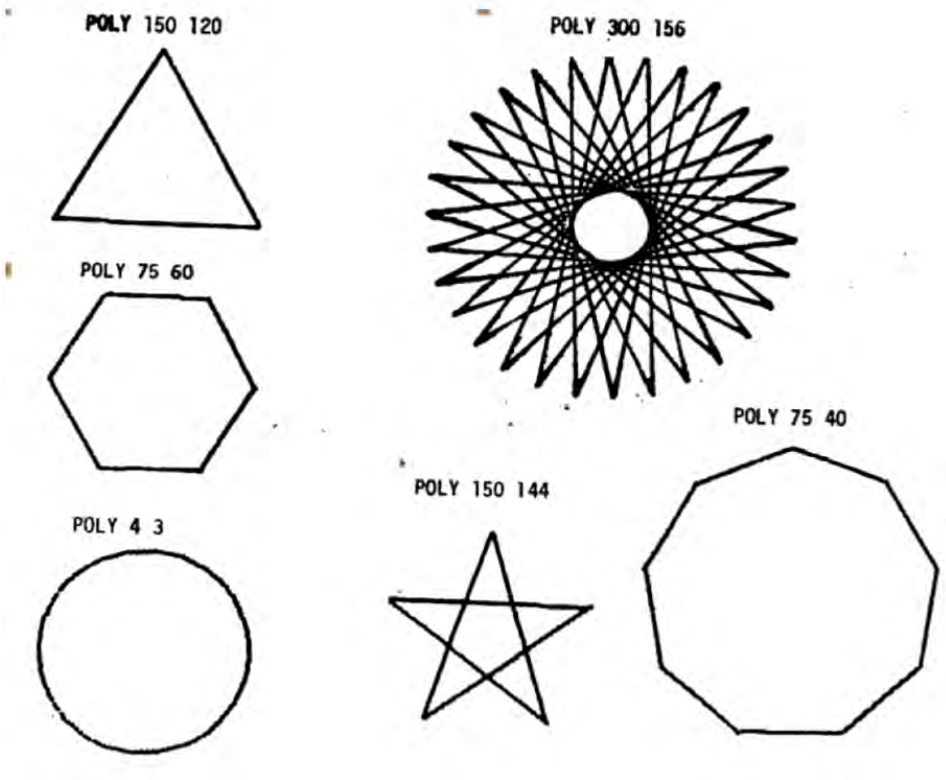

Fig. 9. Result of running POLY with different inputs (Cynthia Solomon collection)

A slight modification to line 3 causes the procedure to draw spirals. This version no longer illustrates the Total Turtle Trip Theorem, since the distance that the turtle moves is no longer fixed. We can also change the name of the modified procedure to POLYSPI.

```
TO POLY :STEP  :ANGLE
1   FORWARD :STEP
2   LEFT :ANGLE
3   POLY   :STEP   :ANGLE
END
```

```
TO POLYSPI :STEP :ANGLE
1   FORWARD  :STEP
2   LEFT :ANGLE
3   POLYSPI  :STEP+5 :ANGLE
END
```

Fig. 10. POLY and POLYSPI programs and POLYSPI output (Cynthia Solomon collection)

One can ask: what are good inputs? Can the turtle make any polygon? What are the inputs for a regular polygon, a star, or a rosette? Can you predict how many vertices will be in the figure? What about how many times the turtle turns 360 degrees? — *This is called the winding number.*

4.2.4 1973–76: Button Boxes and Slot Machines. Radia Perlman, an undergraduate student at the Logo Group in the 1970s, was interested in preschool-age children [Perlman 1974], for whom both the small keyboard buttons on a computer terminal and the need to read and write code were barriers to using Logo. She designed several preschool-specific control devices, of which two were implemented: first the button boxes and then the slot machine [Perlman 1976].

There were four different button boxes (See Figure 11.) with different buttons that were made to plug into each other. The boxes were not all introduced at once, but in succession as learning progressed. The four boxes were

(1) The action box: Buttons forward, back, left, right, pen up, pen down, lamp on, lamp off, toot.
(2) The number box: Buttons 1 through 10, and stop.
(3) The memory box: Buttons start remembering, stop remembering, do it, and forget.
(4) The four procedure box: Red, green, blue, and yellow buttons.

Beginners would be given only the action box, in which each button had an immediate effect in the connected floor turtle. The buttons were labelled with pictures, not with text. Only after a child was perfectly comfortable drawing a shape or navigating a maze with the action box would the teacher plug the number box into it. A number button could be pushed before an action box to repeat that action several times.

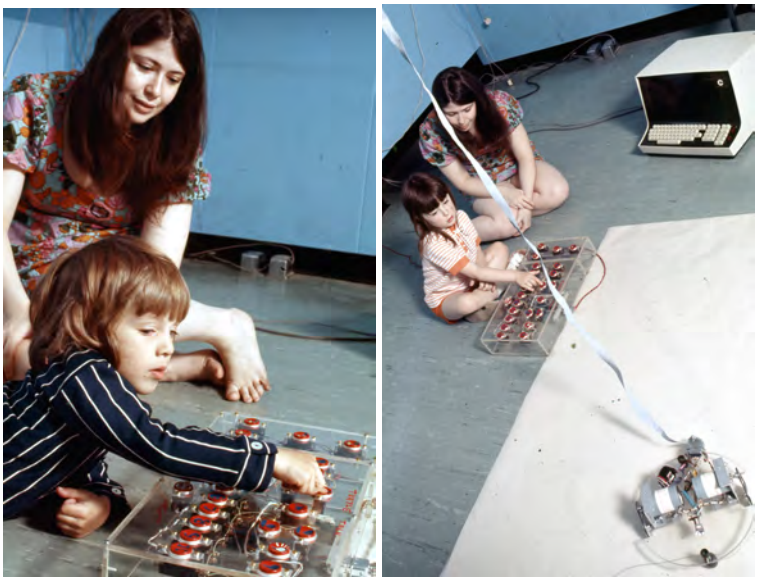

Fig. 11. Perlman and children with button boxes (Cynthia Solomon collection)

The slot machine was an interface for more advanced learners. It consisted of several long plastic bars with slots spaced along each top surface. Each slot bar was a different color. The machine was programmed by inserting cards in the slots. Cards were of different heights. Action cards were the tallest; in front of an action card could be a shorter number card; and in front of that could be a still shorter condition card. So, a single slot could say "if the turtle is not touching a wall, move forward five steps" using one of each kind of card.

4.2.5 Also in the 1970s: Developments in the Logo Group. The MIT Logo Group and the AI Lab created a new implementations of Logo for the DEC PDP-11/45 with stand-alone graphics terminals.

These terminals were based on Tom Knight's terminals for the AI Lab's PDP-10. The PDP-11 system was built primarily by Ron Lebel, the lead Logo systems programmer. In the summer of 1972, the system was used at a math education conference at the University of Exeter, England, where the group worked with turtles, the Minsky music box, and children. DEC had sold a PDP-11/45 to the University of Exeter and so happily also sent along with it the MIT Logo Group's equipment. The Logo Group going to Exeter included Tom Knight and Ron Lebel as well as Hal Abelson, Cynthia Solomon†[14], Jeanne Bamberger, Margaret Minsky†, and Seymour Papert.

A noteworthy event occurred before the conference. One of the 12-year olds, Jonathan Pledge, exploring turtle behavior created a universal maze-solving algorithm. See Figure 12.

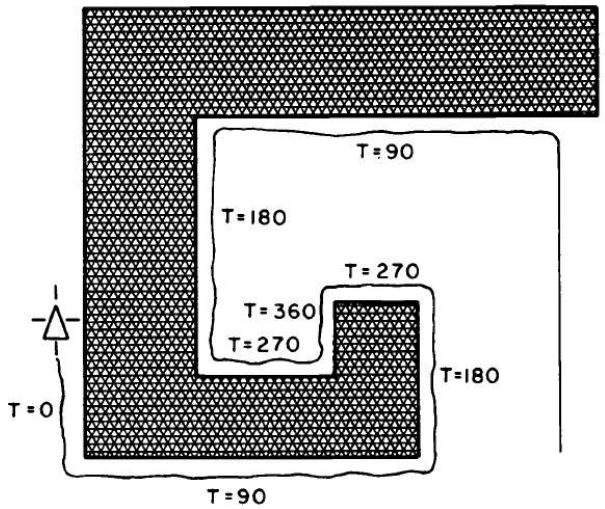

Fig. 12. The Pledge algorithm for escaping from a turtle trap. From *Turtle Geometry: The Computer as a Medium for Exploring Mathematics* [Abelson and diSessa 1980] p. 178. (Courtesy of the authors)

(1) Select an arbitrary initial direction, call it "north," and face that way.
(2) Walk straight "northward" until you hit an obstacle.
(3) Turn left until the obstacle is on your right.
(4) Follow the obstacle around, keeping it on your right, until the total turning (including the initial turn in step 3) is equal to zero.
(5) Go back to step 2.[15]

After the conference, the company General Turtle was formed to make and sell turtles for whoever wanted them. In approximately 1975, Marvin Minsky designed a stand-alone turtle graphics system for General Turtles, the TT2500. It was connected to a Digital Equipment Corporation LSI-11 computer running Logo. These computers were used in the "Brookline project" at the Lincoln Elementary School, Brookline, Massachusetts, with Dan Watt as the principal teacher [Papert et al. 1978, 1979a,b].

[14]The dagger symbol † is used to indicate a reference to one of the authors of this paper.
[15] *Turtle Geometry,* pp. 177–179. Courtesy of the authors.

In 1978 Papert was contacted by Cecil Green of Texas Instruments. TI wanted a version of Logo for the upcoming TI 99/4. His grandchildren were attending the Lamplighter School, a private school in Dallas. The school was going to be given TI 99/4 computers when they were released. The Logo Group did the development. The team included Ed Hardebeck and Gary Drescher. The TI 99/4 had a sprite chip in it. There could be up to 32 colored sprites with individual speeds, but only one turtle for drawing. The sprites could be animated and could sense collisions with other sprites. The version was completed by 1980. The TI 99/4 was the first microcomputer with Logo on it available to schools and homes.

4.3 Powerful Ideas from the Early Days

This section points out powerful ideas that children in a Logo learning environment encounter. Identifying and giving these ideas names so that children can talk about them is in itself the powerful idea of Thinking About Thinking.

4.3.1 Anthropomorphization and Body Syntonics. A central theme in Papert's thinking about how children learn about topics such as computers, mathematics, or even their own thinking and learning, is that we understand things in relation to our own bodies. In contrast, traditional school geometry, for example, views the world or the plane from a bird's eye view: the X and Y axes represent an absolute coordinate system that is foreign to the child. Instead, the natural perspective for a child is to think of what the child wants the turtle to do, in relative terms. To devise or debug a Logo program, the child can "play turtle." Turning left, from the turtle's eye view, might be turning North, or it might be West, or Southeast, in a traditional coordinate system. When the child "becomes the turtle," there is no confusion as to what turning left means. In his later work, Papert used the phrase Body Syntonics to extend this idea. Anthropomorphization is related, but not identical. The child can imagine how the turtle is deciding what to do, by thinking of it as a little person, or (in the case of subprocedures or recursion, see "little man model" in the next section) a set of people who are asking one another to perform certain steps. Conversely, anthropomorphization allows the child to think, "the turtle has a bug" rather than "I made a mistake," thereby focusing on the intellectual challenge, rather than the negative emotions of the situation. The child uses computation as a helpful metaphor for understanding their own thinking, and even for empathy with the thinking and confusions of others.

4.3.2 Metalanguage. The goal of Logo was more than simply developing a programming language for children. Perhaps equally important was to develop a "metalanguage": a collection of metaphors, a vocabulary, and a way of talking both about the language and about the powerful ideas it embodied.

As an example of vocabulary, instead of talking about variables, we talked about associating a "name" and a "thing." Naming a thing was used to describe what most languages refer to as assigning a value to a variable.

Instead of the more common

```
foo = 23
```

Logo sets the value of a variable with

```
name 3.1415 "pi “
```

" if the association was meant to be permanent, or

```
make "x :x+1
```

if the variable was actually varying.

As an example of a metaphor, recursion was described using the "little man model"(since then renamed "little people model"), an anthropomorphic metaphor. Figure 13 illustrates the execution of `PRINT ITEM 2 [A B C]` where `ITEM` is defined as

```
TO ITEM :NUM :SENT
IF :NUM = 1 THEN OUTPUT FIRST :SENT
OUTPUT ITEM :NUM-1 BUTFIRST :SENT
END
```

This figure is a re-creation of the very early original, using modern Logo notation. The text has not been changed, apart from the Logo syntax.

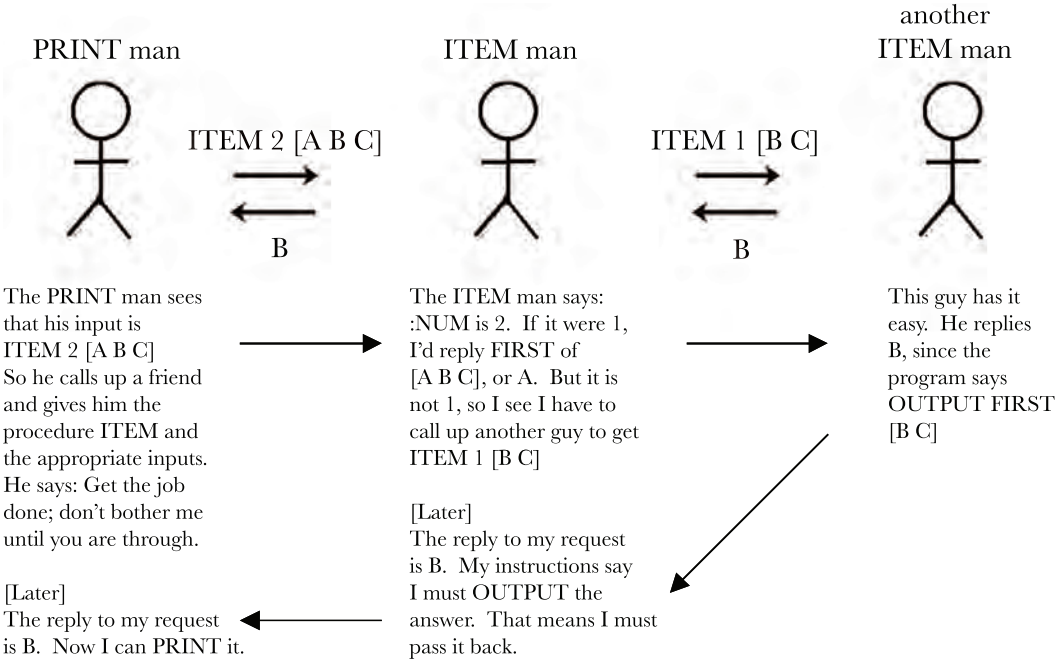

Fig. 13. The 'Little Man' view of Logo computations (Cynthia Solomon collection)

4.3.3 Debugging. Logo did provide tools to help with the debugging of code, when we talk about debugging as a big idea, we're thinking more about a child debugging his or her understanding of the computing process and of the algorithm that the Logo program should implement. For example, a common beginner's misunderstanding is to think that the `left` and `right` commands move the turtle leftward or rightward on the screen, rather than turning it. A Logo teacher seeing a child struggling with this problem is likely to suggest that the child stand up, move into an unobstructed area in the room, and "play turtle," carrying out the instructions to draw a square, the universal first Logo exercise. When the child's leg muscles understand the turning commands, that's the time to return to the computer and revisit the misbehaving code.

In the early days, Logo teachers were careful to say "the turtle has a bug," rather than the child or the program having a bug. In playing turtle, the child's task is to teach the turtle, the child's relationship with the turtle becomes more like a teacher's relationship to a learner.

Another example of a common early misunderstanding is to think that a recursive call means "go back to the top." That misunderstanding works in the very simplest recursive commands, in

which there are no inputs and the recursive call is the last instruction in the procedure. But as the programming tasks get more intricate, the "go back" model fails. The teacher will try to help the child debug his or her understanding by reminding him or her about the Little People model.

The basic message that comes from ideas about debugging is that we learn from our mistakes; that the intricate process of making things work or learning new skills has to do with hypothesizing, testing, revising, and so on. Children are encouraged to collect, classify, and celebrate their bugs. Sometimes bugs, serendipitously, are adopted as features worth perpetuating, sometimes procedures must be constructed to deal with the phenomena caused by their appearance, and sometimes the bugs and their side effect need to be removed. In this pursuit, children become creative researchers studying behavior, making up theories, trying out ideas, etc.

4.3.4 Procedurization. Converting a set of steps into a named procedure is one of the most powerful ideas a child can learn from a short exposure to Logo. It enables the child to use abstraction, one of the "big ideas" in adult Computer Science, without needing a formal understanding of this concept. Without knowing phrases such as "managing complexity," a child can break down the steps to solve a hard problem into simpler subprocedures, and, significantly, give each procedure a name. Each can be debugged separately, and then used repeatedly.. The usefulness of procedurization increases as the child learns to generalize by providing inputs, such as to parameterize the size of a polygon to be drawn.

5 THE '80S AND BEYOND: HUNDREDS OF LOGO DIALECTS

The years 1977-79 saw the introduction of several personal computers designed for the general public: the Apple II, the Commodore Pet, the Atari 800, the TI 99/4, and the Radio Shack TRS-80. (There had been earlier microcomputers designed for hobbyists.) The IBM PC came out in 1981 and the Macintosh in 1984. These machines and others led to an explosion of Logo implementations.

Pavel Boytchev's meticulously researched *Logo Tree Project* [Boytchev 2014] lists 303 versions of Logo through 2015. (Some of them are marked as incomplete versions, often with just turtle graphics.) Readers interested in the details of a particular version are referred there; in what follows, we try to capture major trends.

Of the true Logo implementations (ones with the word and sentence functions, not just turtle graphics; see Section 3.4), almost all derive their design directly or indirectly from one of two sources: (1) versions from the MIT Logo Group, for the Texas Instruments 99/4 and for the Apple II; (2) versions from LCSI, starting with Apple Logo.

We distinguish four historical developments during this period:

We discuss each of these in the following sections.

5.1 Supporting the Myriad New Personal Computers

• TI Logo was developed at the MIT Logo Group. Work started in 1978. It was tested at the Lamplighter School, Dallas, Texas, in 1979, and released in 1981. The 99/4 computer was a commercial failure, but it was crucially important to Logo history because it allowed a version of Logo that included a graphics coprocessor that, in addition to the usual graphics, introduced *sprites* (32 color graphic elements of 16x16 pixels). Each sprite could be moved on-screen and manipulated as a single entity. TI Logo used the sprites, and introduced the `tell` command:

```
tell 12
setspeed 100
```

to send a message to sprite 12. This was a first small step in the direction of object oriented programming in personal computer Logo. (There was also a standard turtle with a pen.) TI built the sprite chip to support animation for video games, but it also allowed multiple-turtle animation in Logo as a fourth microworld (after natural language, turtle geometry, and the dynaturtle (Section 5.2)), even on non-TI computers with a sprite chip add-on. Eventually, computers became fast enough to support animation without special hardware.

• 1981 also saw two versions of Logo for the Apple II. One, from the MIT Logo Group, was licensed to a few companies, most notably Terrapin, which went on to develop versions for other machines and, as of 2020, is still actively developing its version of Logo [Terrapin 2020a,b] and selling floor turtles to use with it. The second Apple version, released as the official Apple Logo, was developed by LCSI. Its design was a deliberate rethinking of personal computer Logo, based on MIT PDP-11 Logo rather than on the earlier microcomputer versions. LCSI, too, went on to produce many further versions of Logo, including a Sprite Logo for the Apple II that came with an add-on circuit board with the TI sprite chip.

• The Apple II had an address space of 48Kb. This was universally viewed as not enough to support any programming language other than BASIC, and Apple sold a "language card" that added an additional 16Kb of bank-switched memory, initially for UCSD Pascal, but also used by Logo. So when Atari wanted a Logo for their 400 and 800 microcomputers that would fit in a 16Kb cartridge, it was quite a challenge, met by Brian Silverman†[16] and his team at LCSI.

• The goals of Berkeley Logo [Harvey 1988]† (also known as UCBLogo) were (1) to allow Logo program portability across DOS, MacOS, Unix, and Windows systems; (2) to support the second edition of the *Computer Science Logo Style* books [Harvey 1985, 1986, 1987] without dialect footnotes; and (3) to establish an ambitious minimum Logo standard for future commercial Logo implementations. It succeeded at the first two goals. It didn't entirely succeed at the third goal, but *some* commercial versions, such as Imagine (Section 5.3.5) and recent versions of Terrapin Logo [Terrapin 2020a,b], did draw inspiration from UCBLogo.

Since UCBLogo is free software (GPL licensed), distributed with source code, several other free Logo implementations were built on it, including Andreas Micheler's aUCBLogo [Micheler 2004], George Mills's MSWLogo [Mills 2016], David Costanzo's FMSLogo [Costanzo 2005], and Chronis Kynigos's E-slate [Kynigos et al. 2000]. UCBLogo was moribund for a while but is now (2020) in active development.

5.2 Design for Microworlds

Perhaps the most important moment in Logo history was the early introduction of *turtle graphics* as a new *microworld*—that is, a new topic to explore through the medium of programming—in addition to its initial focus on words and sentences (arguably the first Logo microworld, although that concept wasn't articulated until turtle graphics appeared as a second one). The third microworld, in the 1970s, was Andrea diSessa's *dynaturtle,* to teach Newtonian mechanics. In this microworld, you can only apply a force to the turtle, with the `kick` command, which takes an acceleration magnitude as input, using the turtle's current heading as the direction of acceleration, and adds the resulting vector to the turtle's vector velocity. [diSessa and Abelson 1986; Sherina et al. 1993] And, as already mentioned, Logo for the TI 99/4 introduced a fourth microworld, *animation:* sprites with costumes and `setspeed` to start a sprite moving independent of the program. (Animation was part of Logo practice from the beginning, but personal computers of the 1980s were too slow to support

[16] An author of this paper. The dagger symbol † is used to indicate a reference to one of the authors of this paper.

animation well without sprite hardware.) In this section we consider other special-purpose Logo developments to support specific microworlds.

• In 1983 Terrapin released Music Logo, designed by Jeanne Bamberger. Due to the limited memory size of the Apple II, Music Logo did not have a turtle.

• In the early days of turtle graphics, floor turtles—actual robots—were as prominent in working with students as display turtles. But the first generation of personal computer implementations had no support for robots, because few schools had them. A collaboration with LEGO led to a revival of interest in robots, starting with LEGO TC Logo for the Apple II in 1987.

The next big step was to eliminate the need for a computer to control the robot by embedding a processor in a (large) Lego brick. MIT prototype "programmable bricks" inspired the LEGO Mindstorms RCX (1998) and later Mindstorms models [Beland et al. 2000].

The MIT programmable bricks were often programmed with LogoBlocks, an early blocks programming language [Begel 1996].

• It's common for software users to want to automate repetitive tasks; therefore, many large programs include a programming language. Sometimes this is an ad hoc scripting language, such as AppleScript [Apple 1993] in macOS, but sometimes a general-purpose language is used, such as the versions of Lisp in GNU Emacs [Free Software Foundation 1985] and AutoCAD [Autodesk 1982]. Logo has played this role in two very different contexts: LogoWriter (1986) from LCSI and HyperStudio (1989) [Wagner 1989] by Roger Wagner. While both are meant for child users, in HyperStudio the application (multimedia presentations) is the main point, with Logo as scripting language, just as in software for adults. In LogoWriter, as in other versions of Logo, Logo is the main point, and the application (a text editor, with turtle graphics) is a microworld.

LogoWriter's microworld is a return to Logo's original focus on *text.* Controversially, users could also manipulate text directly on the screen, avoiding programming altogether. LogoWriter introduced the metaphor of the screen as a two-sided piece of paper, with the project's visible result (text and turtle graphics) on one side and its Logo procedures on the "flip side," always accessible for editing.

Apart from its technical innovations, LogoWriter was important outside the United States because it ran on PCs, which were mandated in Brazil, for example; and because it was translated into many more languages than just French and Spanish. [Valente 2020]

• 1989 saw the release of Mitchel Resnick's ∗Logo (pronounced "star logo") for the Connection Machine. It had hundreds of turtles. It also divided the screen background into thousands of "patches". The turtles and patches could all be programmed in Logo. The pedagogic purpose was to support a *simulation* microworld, especially to explore *emergent phenomena,* in which simple behaviors by large numbers of independent agents give rise to unexpected large-scale behavior by the group as a whole, e.g., birds flying in V-shaped formation or ants looking for food. The name ∗Logo was inspired by ∗Lisp, the massively parallel version of Lisp that ran on the *Connection Machine:* Daniel Hillis's 1985 Ph.D. thesis project [Hillis 1986], a massively parallel computer containing thousands of small processors.

At first this massive parallelism was available *only* on the Connection Machine, but eventually stock personal computers became fast enough to simulate the parallelism. StarLogo for personal computers was developed at Resnick's group at the MIT Media Lab [Resnick 1994, 1996] following the ∗Logo design. Starlogo implemented a MIMD (multiple instruction multiple data) style parallelism for the turtles and a SIMD (single instruction multiple data) style parallelism for the patches. The interpreter tried to hide this distinction from the user and most often succeeded.

In 2009, StarLogo TNG was released [Klopfer et al. 2009], with a Scratch-like blocks interface and a three-dimensional world. The current version of StarLogo is the web-based StarLogo Nova [MIT Scheller Teacher Education Program 2019].

NetLogo (1999), created by Uri Wilensky at Northwestern's Center for Connected Learning, is a StarLogo-based dialect of Logo in which the interactive behavior of multiple agents is central. An extensive programming effort has made NetLogo extremely efficient, increasing the number of agents that can be simulated. NetLogo is widely used for agent-based modeling not only in K-12 teaching but also in academic research by adults, e.g., in economics. NetLogo has not made a move from text to block programming as StarLogo has done, reflecting its growing adult audience compared with StarLogo's continuing emphasis on teaching.[17] NetLogo has also added several primitive user interface capabilities.

Goldstein and Lieberman's† *Germland*, a cellular-automata microworld, was meant to show students that a small set of program rules in a simplified environment could lead to complex behavior and emergent phenomena [Goldstein 1973]. It was inspired partly by contemporaneous AI Lab artificial-life research on the *Game of Life* by Bill Gosper, Tom Toffoli, Ed Fredkin, and others. It could be seen as a precursor to *StarLogo*.

• In 1993 LCSI released MicroWorlds for the MacIntosh. It was primarily a microworld for storytelling and videogames. It included features that had always been desirable, but until then could not be implemented because of the limited resources of earlier personal computers. For example MicroWorlds included an unlimited number of turtles. These were implemented as software sprites. There was also an unlimited number of threads, allowing turtles to act independently of each other. In addition to turtles there was a collection of user interface objects like sliders, buttons and text fields.

5.3 Object Oriented Programming

5.3.1 An Early Research Version: Director. Inspired by the Actor model of computing [Hewitt et al. 1973] Ken Kahn† developed an object-oriented programming language on top of Lisp Logo [Goldstein 1975]. Director [Kahn 1976] was designed to support the programming of many interacting animated sprites (though this predated the use of the term "sprites"). Sprites could send messages to other sprites and messages could be delegated to other sprites to support code sharing. The syntax of Director relied upon pattern matching and like Lisp Logo could resemble Logo or Lisp. Director ran only on large computers capable of running MacLisp, and attracted only a small number of users.

5.3.2 Programs as Collapsable Nested Boxes: Boxer. Boxer(1983), by Andy diSessa [diSessa and Abelson 1986], was an early object-oriented dialect of Logo. It was the first step toward a visual version of Logo. Users could put code and data inside a box, drawn on the screen. This basic capability could be used for anything from the conditional commands of an if command to an object, boxing its methods and fields. Boxes could nested, and any box could be zoomed open or closed. This was a solution to the problem of very large programs that couldn't show the user the overall organization of their code. Boxer provided the encapsulation aspect of object-oriented programming wherein a box was an object: packaging methods and data and making them invisible outside the box. Boxer ran on expensive single-user computers, Sun workstations and Lisp Machines; later there was also a version for the Macintosh though it was not widely distributed.

5.3.3 Object-Based Functional Programming: TLC Logo. John Allen's TLC Logo [Allen et al. 1984] (the acronym is for "The Lisp Company") was also released in 1983. It *did* run on inexpensive personal computers, and so it introduced a thoroughgoing object orientation to a wider potential audience. Turtles could `spawn` other turtles, and send them messages. Each turtle had its own thread of execution.

[17] As we go to press, NetLogo has released a beta version with a block programming capability.

TLC Logo followed the "everything first class" philosophy: Turtles, procedures, environments, closures, packages, lists, vectors, input/output streams and other types were possible values in all contexts. It used dynamic scope, but allowed the user to provide a lexical environment as part of a procedure call. APPLY, EVAL, and a REPL were all usable by users. It was pretty good at honoring both its Lisp roots and its Logo roots, e.g., `2 + 3` and `+ 2 3` both gave 5. It had a `?` primitive that evaluated to "the question mark object"; other primitives took this to mean "tell me your current value." Thus, `ask :turtle3` changed the current turtle, but `ask ?` just returned the current turtle (the object, not its name, of course). You could create subclasses of any built-in class. The documentation was very up-front about TLC's membership in the functional paradigm conspiracy.

These two examples signify the beginning of a return to *computer science* as a microworld, with projects such as writing a Logo interpreter in Logo itself.

5.3.4 Everything Is an Object: Object Logo. The next milestone was the 1986 release of Object Logo [Drescher 1987], an object-oriented dialect of Logo. Gary Drescher designed a prototype circa 1983 at the Atari Cambridge Research Lab, and Coral Software developed and sold a microcomputer version of it [Coral 1986]. The central idea of Object Logo is that an object is a dynamic-binding environment: while commands are executed inside an object, the object's local versions of any functions or variables override any global versions. One object can be constructed as a specialization (KindOf) another, inheriting the other's functions and variables except where the specialized object provides its own versions. Specialization can be used either to construct what are traditionally considered subclasses (defining functions with modified or extended behavior) or traditional instances (objects with their own versions of state variables). But the language does not require that distinction, and a "class" object can also serve as a prototype "instance."

Object Logo showed that prototyping OOP, done wholeheartedly, results in a very simple object hierarchy. See Lieberman's† OOPSLA86 paper for a discussion of simple prototyping [Lieberman 1986].

Object Logo also extended OOP to things other than turtles. Windows, menus, external files, and many more object types were built into the language and programmable by users. Unusually, it had multiple inheritance. Version 6.1 of UCBLogo includes a subset of the Object Logo OOP design.

5.3.5 Objects Across Computers: Comenius Logo, SuperLogo, Imagine Logo. Comenius Logo (1992) [Blaho et al. 1994, 1993; Kalas and Blaho 1994] and *Imagine Logo* (2001) [Blaho and Kalas 2001; Blaho et al. 2000], from Comenius University in Slovakia, introduced shapes and animations as first class data; programmable choosers for missing inputs of partial procedure calls; prototyping OOP with Turtle, Page, Pane, ToolBar, Button, Web, Net, etc. as built-in classes; true operating system multithreading both in response to events (e.g. mouse click or drag) and explicitly launched in code, including a critical section capability; a built in programmable Web browser class; and Net class for communication among several running Imagine programs on the Internet. (In a great demo project, a turtle glides through a doorway on one computer and glides out of the doorway on the next computer over, remembering all the local state of the original turtle.)

Imagine's design pays explicit attention to exposing objects to users at different levels of cognitive complexity, from traditional single-turtle Logo programming, through multiple traditional turtles, then multiple turtles with private variables and methods, to copying objects, making instances of prototype instances, and finally the standard class/instance model.

5.4 Localization

Logo's vocabulary was often translated into other languages. As early as the 1970s there were versions in French [INRP 1981]:

```
REPETE 4 [AVANCE 100 DROITE 90]
```

draws a square.

By the 1980s there were Logo versions in several European languages, as well as in Arabic, Hebrew, Russian, Chinese, Japanese, and Wolof. More often than not each language had its own version. This helped in sorting out different wording and word order in things like error messages. There was no operating system support for font rendering and input methods so these had to be invented or re-invented on a version by version basis. Typically these version were done in conjunction with educators who spoke the relevant natural language as a first language.

A particular challenge in Logo's approach to (natural) language is that English Logo doesn't easily translate to other languages. English has virtually no inflection; one can define a procedure with `to foo` (the infinitive) and invoke it with `foo` (the imperative). In inflected languages, say French, one would expect, e.g., `pour fooer` for the infinitive and `fooez` for the imperative. The translator has to find wordings that make one name natural in all contexts.

By the 1990s most operating systems had the facilities to make localizations easy for developers. This led to a proliferation of other language versions, amongst others Thai, Greek, Portuguese, Brazilian and French.

The development of many language versions was coupled to research activities and deployment in schools in many different countries [Papert et al. 1999; Seye Sylla 1985] including Costa Rica, Russia, Argentina, Australia, Brazil, Thailand, Senegal, and others. Fonseca [Papert et al. 1999] says that in Costa Rica Logo reached "approximately 225,000 children annually−—i.e., one out of every two school children in the country."

6 VISUAL PROGRAMMING LANGUAGES

The present and near future of programming languages for children seems to be visual languages that work by snapping blocks together, each block representing a procedure call. The block style of programming has the advantage that you don't have to know how to type; very young Logo beginners spent a long time hunting around the keyboard. Many (but not all) of the block-based languages are, in various ways, children of Logo. One important exception is Blockly [Fraser 2015; Pasternak et al. 2017], not a language, but a block-based user interface for any language, developed at Google, which has been used in Scratch 3.0 and in the Android version of App Inventor.

6.1 From Text to Blocks: A Personal Reflection from Brian Silverman

In the early 1990s the discussions of some sort of visual Logo kept swirling around. I was a bit skeptical about visual languages. I had discussed this with Seymour Papert. He felt it was just a switch of representation that wouldn't make much of a difference. I pretty much agreed. Mitchel Resnick didn't share our skepticism. He pushed for putting it to the test. Try it out any see what happens. I thought trying it out would be good. We could then see that Seymour was right and get the distraction behind us.

At that time I was working with Mitchel's group at the MIT media lab. The group had worked on "programmable bricks" that controlled LEGO contraptions, They turned out to be the precursor to LEGO Mindstorms, Initially the programming for the bricks was done in Logo. We felt that these bricks could provide a good environment to experiment with alternate programming language ideas. There were many opportunities for kids to build elaborate projects without much need for technical sophistication.

We encouraged a then-undergraduate, Andy Begel, to design and implement a simple blocks language. As part of his Advanced Undergraduate Project Andy created Logoblocks [Begel 1996]. We tried it out with kids and teachers. To my surprise, but not Mitchel's, it worked really really

well. What Seymour and I hadn't anticipated is that the fact that it appeared simple allowed people to be more willing to engage in the early stages than they otherwise would have. In some sense it wasn't simpler, but it appeared simpler and that mattered..

The first version of LogoBlocks was written in Macintosh Common Lisp, great for prototypes not that great for distribution. We ported LogoBlocks to Java to ease deployment. Then people ran workshops, teachers worked with children, and we became convinced that the concept was solid. One could almost say that the idea of programming with blocks snapped into place.

In the early 2000s, Mitchel, Natalie Rusk, and I were working on the design of two projects.

One was the PicoCricket [Playful Invention Company 2014], with a programming language based on LogoBlocks. The PicoCricket project also pulled in Robbie Berg and Paula Bontá. Paula ultimately became a major contributor to the design and implementation of several blocks languages. The PicoCricket kit was an attempt to make an alternative to LEGO's Mindstorms that was less "powerful" and more playful.

The other project was Scratch [Maloney et al. 2004]. Scratch found its early inspiration in Squeak eToys, LCSI's Microworld Logo, and LogoBlocks. The Scratch team team then also included John Maloney, who did much of the initial coding. A number of Mitchel's students also joined in for varying periods. In some sense, we hoped Scratch could play a similar role in the world that Logo did – be a constructionist construction kit. My own involvement tapered off fairly early, before the project really started growing.

Around 2004, I had a design quibble with Mitchel about the UI for blocks programming. Scratch uses only a fraction of the screen for blocks, leaving the blocks canvas, in my mind, too cramped. The PicoCricket system always felt more spacious. It controlled things off the computer so the whole screen was available for blocks. Paula and I decided, as a design experiment, to make a version of PicoCricket system that controlled a Logo display turtle. That way we could explore alternative ways to have the code and work share the screen. We allowed the code to overlay the work. I loved it. Mitchel didn't.

We called that experimental version TurtleArt. It does not have words, lists, or data manipulation. It does have a color model that loosely resembles 18th century artist P. O. Runge's Farbkugel [Schopenhauer and Runge 2010]. Over the next few years Paula and I refined the software. Artemis Papert joined in as our "artist in residence." We showed the project to lots of people, did workshops, handed copies out to friends, and got lots of feedback. An amusing bit of early feedback was from Stephen Wolfram. He told me "bad idea, you shouldn't make programming look easier than it really is." Seymour's early reaction was: "when will you stop wasting your time and get back to real work."

Despite the early criticism, in time there were three releases, one bundled with the OLPC XO, a Java based version, and an iPad app.

6.2 Brian Harvey's Personal Narrative on Snap*!*: Scheme Disguised as Scratch

In 2009, the University of California, Berkeley, was one of several universities developing a new kind of introductory computer science course, meant for non-CS majors, to include aspects of the social implications of computing along with the programming content. Scratch wasn't quite expressive enough to support such a course (it lacked the ability to write recursive functions), so Prof. Daniel Garcia and I thought "What's the smallest change we could make to Scratch to make it usable in our course?" After 20 years teaching *Structure and Interpretation of Computer Programs* [Abelson et al. 1984], the best computer science text ever written, I knew that the answer to "what's the smallest change" is generally "add lambda." I joined forces with German programmer Jens Mönig, who had developed BYOB (Build Your Own Blocks), an extension to Scratch with custom (user-defined) blocks, including reporters and predicates. At that time we were hoping to convince the Scratch Team to adopt our ideas, so we took "smallest change" very seriously. BYOB 3.0 [Harvey

and Mönig 2010], with first class procedures and first class lists, added only eight blocks to Scratch's palette. (The code is almost all Jens's. My contribution was part of the user interface design, plus teaching Jens about lambda.) Version 3.1 added first class sprites with Object Logo-style inheritance. The Berkeley course, *The Beauty and Joy of Computing* (BJC) [Garcia et al. 2012], is also used by hundreds of high schools, especially since the College Board endorsed it as a curriculum for their new AP CS Principles exam. Unfortunately, some teachers have no sense of humor, and so BYOB version 4.0, a complete rewrite in JavaScript, was renamed Snap*!* [Harvey 2019].[18]

Since Scratch seemed to be positioned as the successor to Logo, it was a goal for Snap*!* to restore the features from Logo that are missing in Scratch. The most important missing feature, the ability to define functions (and therefore to use recursive functions), is at the core of the new language. (Scratch introduced user-definable command blocks in version 2.0, but still doesn't support user-defined reporters.) Scratch had also replaced the structured text (word and sentence) functions with a flat text string data type. We wanted to be backward compatible with Scratch, so we implemented words and sentences as a library, defining `first`, `last`, `butfirst`, and so on. (Since block languages allow multi-word procedure names, and you don't have to type the long name in order to use the procedure, the library names are, e.g., `all but first letter of`.)

Lists are first class and can be arbitrarily deep in sublists. The usual higher order functions on lists are provided; the graphical representation of lambda is built into the blocks representing higher order functions, and so beginning users can use higher order functions in simple cases without thinking hard about function-as-data at all, but the full power of lambda is available to more advanced programmers. It took us three tries to get the lambda design right, but we're very proud of its pedagogic benefits.

Another of our goals for Snap*!* is to be a complete version of Scheme; it was largely as a way of planting that flag that we added `call with current continuation`, not taught in BJC (nor even in SICP) but used to implement tools such as `catch` and `throw` as library procedures written in Snap*!* itself. As of this writing, macros are only half-implemented; users can define procedures whose inputs are unevaluated (more precisely, thunked, since procedures are first class), but cannot yet inject code into the caller's environment.

Snap*!* is lexically scoped, not least to allow the use of closures as objects, but a planned extension is "hybrid scope": variable names follow lexical scope, but instead of giving an error message when no binding is found in the lexical environment, the evaluator will instead look in the dynamic environment. So name capture is impossible, since the global environment is examined before the dynamic environment. (Only if a mistyped name matches another name can the user get the wrong variable rather than an error message. But mistyping can't really happen in a block language.) This, too, is an effort to be a Logo as well as a Scheme.

Since Snap*!* is free software (AGPL), it has served as the starting point for at least a dozen significant extensions, including BeetleBlocks [Koschitz and Rosenbaum 2012; Rosenbaum et al. 2011] for 3-D graphics and 3-D printing; TurtleStitch [Mayr-Stalder and Aschauer 2016] for controlling sewing machines to do embroidery; Edgy [Bird et al. 2013] for studying graph theory; NetsBlox [Ledeczi and Broll 2016] for access to online data APIs and collaborative editing of projects; and others. The ability to write new Snap*!* blocks in Javascript, from the Snap*!* editor, has allowed many other user-level extension libraries, including support for robots and other hardware. Snap*!* features such as first class procedures help authors develop these extensions, even if the users of an extension don't see that.

[18]For non-Anglophones, "BYOB" is used in party invitations as an abbreviation for "bring your own booze."

6.3 Access to Smartphone Hardware: App Inventor and Pocket Code

App Inventor [Wolber et al. 2015] is a programming language for Android phones and tablets, developed by a team led by Harold Abelson (who led the development of MIT Logo for the Apple II and co-authored *Turtle Geometry* [Abelson and diSessa 1980]), first at Google and later at MIT. Its design principle is "What would Logo have looked like if we'd had mobile telephones back then?" The fact that many children have phones means that they can build a program to satisfy some real need that they have, and carry the program around with them. An app is created in a block editor running in a browser; it can be run on an Android device immediately as it is edited, and then compiled and downloaded to the Android device for standalone use. The advantage that App Inventor has over other block-based languages is that it has access to the properties of the telephone: it can dial numbers, send text messages, read the accelerometer, and so on. Although it lacks first class procedures, it does make these interface components first class.

Pocket Code [Slany 2012; Slany et al. 2018] is an app that allows users to create and execute Catrobat programs on Android and iOS smartphones, without the need for a laptop or tablet. Although not directly descended from Logo, the block-based visual programming language Catrobat is heavily inspired by Scratch. The user interface was designed for use on tiny screens, and so the UI elements (the stage, the scripting area, etc.) are separate views rather than all present at once.

6.4 Ken Kahn's Personal Narrative on ToonTalk: Concurrent Constraint Programming

I was directly inspired to create ToonTalk by a comment Seymour Papert made in a Logo meeting in the second half of the 1970s. Papert described Logo as child-engineering the best ideas in computer science and AI *of the time*. After more than a decade doing research on logic programming and concurrent distributed computing, I thought that there were new powerful computer science and AI ideas that could be engineered to become accessible to children. Specifically, research on concurrent constraint programming [Saraswat 1993] that unified ideas from logic programming, constraint programming, and concurrency. Vijay Saraswat and I created Pictorial Janus [Kahn and Saraswat 1990]. This language was purely visual; the syntax was based only upon the topology of drawings.

When in 1992 I discovered that Pictorial Janus was not accessible by children (or most adults) I concluded that children needed a programming language that was much more concrete that relied heavily upon animation. This led to ToonTalk, another attempt to child-engineer concurrent constraint programming [Kahn 2001]. ToonTalk programs are constructed by demonstration inside an animated world, training virtual robots to manipulate data and communicate by giving messages to birds. Unlike other visual programming languages, ToonTalk represents programs as animations, not as static graphics. I strove to match Logo's goal of providing a low threshold and a high ceiling. Morgado's doctoral thesis (which I co-supervised) demonstrated the extent to which preschool children (many of whom had yet to learn to read) could master ToonTalk [Morgado 2015]. As an example of a high ceiling, ToonTalk was used in a graduate-level course on concurrent programming at Keio University. I conceived of ToonTalk as a new Logo [Kahn 2001, 2007]. From 1998 to 2007, ToonTalk had several national publishers and was the basis of large-scale research projects [Kahn et al. 2011; Mor et al. 2004].

7 CRITICAL PERSPECTIVES ON LOGO

In this section, we focus on critiques of Logo's role in children's learning. Logo's goals, educational philosophy, and approaches to assessment have been the subject of numerous internal and external critiques, giving rise to intense debates. While it would be impractical to discuss the five decades of literature on this subject, we highlight a few examples, to illustrate the nature of these conversations.

Most critiques of Logo have come from education researchers, who sought evidence regarding claims such as that children could develop enhanced problem-solving or computational thinking skills through exposure to Logo. There have been multiple areas of misunderstanding by all involved: some resulting from over-optimistic claims, some from neglecting key assumptions behind the claims, some from varying educational objectives, and some from differing research paradigms. Most research done on Logo by the early teams from BBN and MIT focused on detailed interactions with small numbers of children; direct observations by researchers led to continuous improvements in both the language and teaching practices [Papert et al. 1979b]. Those working in the educational research paradigm expected to see controlled experiments, wherein a treatment group had access to computers running Logo and a control group did not. They sought statistically significant, quantitative measurements of specific skills or achievement, often on traditional tests. Although some researchers ([Clements and Gullo 1984]) reported positive results from such studies, others ([Pea 1983], [Pea et al. 1985]) reported disappointing results. The negative findings adversely influenced Logo funding and decisions whether to adopt Logo in schools throughout the world.

In November 1982, Cynthia Solomon†[19] visited The Bank Street School, where Roy Pea and Midian Kurland were conducting their studies. Solomon had the opportunity to observe the classroom. They were working with children 11-12 years old. Two teachers had prepared for teaching Logo by attending a three-week workshop. Teachers and children were using three different Logo implementations on two different types of computers, with various feature differences. One of the teacher handouts given to the children for instruction contained several bugs, noticed by some of the children. The main conclusion of the study was that teaching children Logo did not help children learn broader skills such as planning.

Taking into account Solomon's observations, Pea and Kurland's findings should not have been surprising to Logo enthusiasts, given all the contributing factors (what was taught, how was it taught, which hardware was used, which Logo version, what was measured, how was it measured, and so on). It is difficult to establish and control all of these factors, including teacher preparation involving both programming itself and a vision of programming's role.

Jim Howe and Tim O'Shea [Howe and O'Shea 1978] and John Self [O'Shea and Self 1983] obtained better outcomes when Logo-knowledgeable researchers were present in the classroom, as opposed to when teachers new to Logo taught on their own. If there are to be consistent positive outcomes in educational settings, significant investment in teacher preparation is sure to be crucial. The central importance of teacher preparation remains a crucial lesson today, as schools attempt to bring computer science into the classroom.

In Chapter 7 of their book *Windows on Mathematical Meanings* [Noss and Hoyles 1996], Richard Noss and Celia Holyes describe research into how Logo might help to transform education, with an emphasis on mathematical reasoning. They provide a fresh perspective on some of the earlier critiques, including examples of effective and ineffective uses of Logo, sample measures from the UK mathematics curriculum, performance on puzzle-solving tasks, performance on achievement measures for algebra and geometry, and discussion of alternative pedagogical styles and strategies.

8 LOGO'S INFLUENCE ON AI AND COMPUTER SCIENCE

The flow of ideas between artificial intelligence and Logo was not only in one direction. Important ideas first introduced in Logo also fed back to artificial intelligence, and to computer science more generally. This was especially evident in research areas pursued by graduate students at the MIT AI Lab. Henry Lieberman, Mark Miller, and Ken Kahn share personal reflections.

[19]The dagger symbol † is used to indicate a reference to one of the authors of this paper.

8.1 Henry Lieberman's Reflections on Logo, AI, and CS

I came to the MIT Logo Group around 1971, when I was an undergraduate student. Two things attracted me to Logo: First, I had always had an interest in alternative education. I was a member of the MIT Experimental Study Group, an undergraduate program that allowed students to design their own curricula and work on independent projects. We had read several of the great advocates of educational reform: Dewey, Neill, Freire, et al. I was drawn in by Papert's vision for self-directed and experiential learning.

Second, I was becoming excited by AI, which seemed to me to hold the route to understanding human intelligence. I took courses from Marvin Minsky, Patrick Winston and Carl Hewitt, whose research on understanding learning in machines seemed like it might yield useful insights into learning in people. Papert brought these strands together in saying that we should encourage students to "think about thinking" and give them the best available tools of AI to express their ideas in procedural form.

Ira Goldstein, a graduate student in the AI Lab, created Lisp Logo [Goldstein 1975], an implementation of Logo in Lisp. I went to work as an apprentice to Goldstein. My first project was Germland [Goldstein 1973], a cellular automata simulation Microworld meant to drive home the point that complex interactions could arise from computation using simple rules.

After a few such experiments, it became clear that Lisp Logo was a good vehicle, both for experimenting with other AI microworlds enabled by code in Lisp and for experimenting with extensions and new languages on top of Logo. I took over the development of Lisp Logo from Goldstein, initially focusing on graphics. When the first color and projection displays became available to the Logo group, I implemented the first color Logo. I taught a class to high school students with a Microworld demonstrating additive and subtractive color mixing. We also used two-color glasses to implement the first 3D Logo graphics. Ken Kahn†, some students, and I won a prize in the first computer art contest sponsored by Byte magazine [Byte 1976]. Kahn's reflection in Section 8.3 illustrates some of the higher level language experiments that Lisp Logo enabled.

My experience with Actors led me to propose that object systems be organized around the idea of *delegation*: One "little man" asks another to do something if he needs help [Lieberman 1986]. Delegation uses the fundamentally dynamic message-passing paradigm, in contrast to the more static *inheritance* model of Smalltalk. Delegation, and the desire to implement learning-by-example, meant sharing knowledge between objects by the use of *prototype* objects rather than the *class-instance* mechanism of Smalltalk. Prototype object systems are now common in several programming languages, such as JavaScript.

Though many Logo programming environments did not provide much in the way of debugging tools, Papert's emphasis on the problem-solving processes of debugging did lead me to imagine what better debugging tools for beginners might look like. Ron Baecker, one of the early pioneers of program visualization and animation, did a film of a simulated animated Logo debugger [Baecker 1975] that visually replaced expressions with their values as the program was executed. I implemented a debugger for Lisp in this style with Christopher Fry [Lieberman and Fry 1995]. Marc Eisenstadt and Mike Brayshaw implemented a full Prolog interpreter deploying Baecker-style program visualization in the 'Transparent Prolog Machine' [Eisenstadt and Brayshaw 1988].

Our interest in the debugging problem inspired several theses in the AI Lab that looked at the structure of simple Logo programs, and investigated the diagnostic reasoning necessary to recognize and fix certain kinds of bugs. They created models that could be useful in automatic programming systems, intelligent tutoring systems and advanced programming environments. Ira Goldstein's MYCROFT [Goldstein 1975] automatically debugged simple turtle graphics programs.

Mark Miller† wrote his thesis on SPADE (*Structured Planning and Debugging*) [Miller 1979; Miller and Goldstein 1977], as further discussed below.

8.2 Mark Miller's Reflections on Logo, AI, CS, and Psychology

I began graduate work at MIT in 1972, with strong interests in both cognitive/experimental psychology and computer science. As an undergraduate, I had been inspired by my UC San Diego advisor, Don Norman, and fascinated by Marvin and Seymour's book, *Perceptrons* [Minsky and Papert 1969]. I spent my first year at MIT in what was called the Department of Brain and Behavior. Activities included single-cell recording of neurons, a line of research inspired by early work on neural networks [McCulloch and Pitts 1943], now foundational in modern AI. While intrigued, I decided to change departments and joined the AI Lab. The cognitive scientist in me remained deeply curious about how children think and learn, so I joined the Logo group. I also embarked on an extended introspective study: I enrolled, as a student, in a course, *The Mechanics of Solids*, chosen because it was completely foreign to me. I kept a detailed, contemporaneous journal of my own efforts to "construct the knowledge" in this course. My goal was to apply the AI theories of Minsky and Papert, creating an explanation of the notes in my journal. I submitted this idea as a dissertation proposal, but received no feedback.

Alongside Henry, I began working with Ira Goldstein. I made minor contributions to Lisp Logo, including code for the music box, a very efficient turtle graphics package ("TV Hare"), and a scalable font for turtle-drawn text. Driven by my earlier training in psychology, I also conducted a few pilot experiments with children and teachers using Logo. In one study, I tried to understand whether learning Logo was better for children than learning BASIC. I ran a pilot study in Brookline, MA, involving two groups of students, learning both languages: half learned Logo first, then BASIC; the other half learned the languages in the reverse order. Pilot data suggested that the BASIC-first group never fully understood Logo, in several ways: their programs drew using Cartesian ("God's eye view") coordinates rather than turtle-centric, relative coordinates; they avoided breaking down programs into sub-procedures; and they never used recursion. The Logo-first students learned both languages, but did not understand why anyone would use BASIC! I also ran a pilot study exploring the possibility that learning Logo might transfer to improved performance on problem-solving tasks, such as puzzles (similar to [Noss et al. 1997]), working with two classroom teachers. One taught Logo without any discussion of powerful ideas or applicability to other tasks; the other emphasized breaking problems into sub-problems, debugging, and other heuristic strategies, drawing analogies to writing essays and solving puzzles. Improved puzzle performance was observed in the second group, but not the first. Such experiments were not mainstream in the lab, so I never replicated these pilot studies with large enough N; I did not publish this work.

Meanwhile, Seymour had been running some informal seminars on what he called, "Loud Thinking." I was strongly influenced by these, as well as the work of Newell and Simon [Newell 1972] on analysis of human problem solving protocols. I began collecting keystroke-by-keystroke protocols of students working in Logo. My ambitious goal was to explain their behavior as episodes of planning and debugging, analogous to the operation of Gerry Sussman's AI research ("Hacker") [Sussman 1973].

Protocols of student Logo programming provided an ideal microworld for AI, analogous to the blocks world. The relationship between automated analysis of programming protocols and the AI goal of "program understanding" seemed strong. As a result, a series of papers jointly authored with Ira Goldstein pursued this topic and led to my dissertation. I programmed—leveraging Lisp Logo—a prototype for a future children's programming environment that would interact with the student in terms of *plans* and *bugs*, rather than mere *code* [Miller 1979].

For me, the significance of teaching children thinking, and empowering them through computer science, was profound. Although I never contributed to a commercial version of Logo, I seemed to follow Logo, almost wherever it went. The entire time I was a graduate student, I also worked at BBN. I knew Wally, but worked in another group, investigating AI approaches to the use of computers in education, which led to contributions in cognitive science [Collins et al. 1975]. After BBN and MIT, I headed for Texas Instruments, where Seymour was regularly consulting on Logo at The Lamplighter School. Personal computing was about to explode on the world, and TI seemed likely to advance Logo in major ways. But my only direct contributions to TI Logo were occasionally explaining to a TI engineer what `CAR` or `CDR` meant, as they tried to decipher TI 990 Assembler code written by the MIT "hackers." The best part of my time at TI was that Seymour would stay at my house when in town to consult and I had the pleasure of long and deep conversations with him. One evening, he told me that Piaget had read my first dissertation proposal (Journaling Solid Mechanics) and praised it. Seymour was puzzled that I had not pursued it!

Eventually, after seeing the Apple II (with its own Logo) and then a "Lisp Machine" board manufactured by TI for Apple, I ended up spending a decade at Apple, becoming Lab Director for Learning and Tools in the Advanced Technology Group. Now, I teach Computer Science, emphasizing that bugs are good, through such Logo-inspired techniques as giving out small plastic bugs, to celebrate the "coolest" ones, as the student explains the nature of their bug and how they went about debugging.

8.3 Ken Kahn's Reflections on Logo, AI, and CS

Soon after entering the MIT AI Lab as a doctoral student in 1973, I became fascinated by how Logo was making some of the things I was learning about AI programming accessible to children. In particular I was excited about the ways that AI could play multiple roles in education [Kahn 1977]. I became interested in providing children with tools in Logo for exploring natural language processing [Kahn 1975]. I also explored how Hewitt's ideas of actors could be applied to enhance Logo. This led to my work on Director [Kahn 1979a], which built upon Logo (and later Lisp Logo) to provide support for concurrency, delegation, and animation [Kahn 1976]. My ideas about AI and children led to my doctoral thesis on an AI system that could turn stories into simple animations [Kahn 1979b]. Ten years after leaving MIT I returned to research on programming languages for children (see 6.4). Most recently I've been working on enabling children to do AI programming in Snap*!*.

9 LOGO, SCHOOL, AND CHANGE

The early Logo researchers viewed themselves as *mathematics* educators. Logo was an instrument to help young children *do mathematics,* rather than learning about other people's mathematics encapsulated in a curriculum. The paradigmatic research paper title was "Teaching children to be mathematicians versus teaching about mathematics" [Papert 1972b].

It was for the sake of the mathematics that Logo researchers and teachers developed what we have been calling the "Logo environment": each child or group working on a project of their own choice, time allocated for children to reflect on their work process and their thinking, the teacher as helper rather than as authority.

The advent of inexpensive personal computers and the publication of *Mindstorms* in 1980 suddenly brought Logo to large numbers of teachers without the benefit of teacher preparation workshops run by the Logo developers. Unsurprisingly, many of these teachers viewed Logo differently, as *mainly* about empowering children, and only secondarily, if at all, about mathematics.

The sight (and sound!) of a roomful of self-directed children is polarizing. Some teachers instantly love it; others find it chaotic and unconducive to learning. Among those who love it, some began

talking and writing about "the Logo philosophy." By this they didn't mean doing math versus learning about math; they meant child-centered learning, focusing on the actual child in the room rather than on the imagined future adult, and varying degrees of democracy in the classroom. These ideas, of course, didn't start with Logo. Rousseau published *Emile, or On Education* in 1762 [Rousseau 1762]; this is generally taken as the starting point of the modern progressive education movement. But Logo introduced progressive education to a group of teachers who hadn't been aware of it.

How does the teacher in a Logo classroom feel about the rest of the school, as an institution? And how does the school, as an institution, support or discourage student-centered learning? In some contexts, progressive teachers will feel supported by fellow teachers and by school administrators. In other contexts, progressive teachers will feel the opposite. The former are likely to talk about Logo as an incremental reform to an already-okay school. Some of the latter may use the word "revolution" to describe what their institution needs, although such an extreme word is controversial among the authors of this paper.

Logo makes a dramatic difference in the lives of children when Logo-the-language is embedded in Logo-the-environment. Logo at its best has motivated teachers to rethink their relationship with curriculum and with children, and has encouraged children to reflect on their own relationship with learning.

ACKNOWLEDGMENTS

Many people throughout the world—far too many to acknowledge properly—contributed ideas, insights, and implementations to the Logo language, literature, and culture. Similarly, several people not listed as authors contributed important insights and editorial suggestions to this article; the authors would especially like to acknowledge the following individuals for their valuable contributions: Harold Abelson, Andrew Begel, David Cavallo, Gary Drescher, Mark Eisenstadt, Paul Goldenberg, Steven Hain, Edward Hardebeck, Celia Hoyles, Alan Kay, Shriram Krishnamurthi, Ron Lebel, Russell Noftsker, Richard Noss, Tim O'Shea, John Roe, Oliver Steele, Franklyn Turbak, and José Valente. (Of course, they are not responsible for our remaining mistakes.) Our thanks to Ioanna Papandropoulou and the Fondation Jean Piaget for permission to use the photograph of Piaget, to the MIT Museum for the Spacewar picture and the Bobrow picture, to Wikimedia user Daderot for the Feurzeig photo, and to Frank Frazier for several photos as noted. Mary Dalrymple provided invaluable help with LaTeX and BibTeX.

APPENDICES

A LOGO TIMELINE

Versions or children of Logo are in Caps and Small Caps
Books are in Italics
Other events are in Roman

Logo prehistory, 1959—1965

1958–63 Papert with Piaget (Geneva)
1964 BASIC
1964–ca. 1980 Papert with Minsky

Logo in the lab, 1966—1979

1966 first BBN Logo
1967 Hanscom school

1968–69 Muzzey Junior High School
1969 FIRST MIT LOGO, *Perceptrons* (Minsky and Papert)
1970 "Teaching Children Thinking" Symposium at MIT
1970–71 Bridge School, floor turtles, display turtle
1971 "Twenty Things to Do with a Computer"; Smalltalk
1972 Exeter conference, Pledge algorithm; Actors
1973 EDINBURGH LOGO
1973–76 Button Boxes, Slot Machines
1975 LISP LOGO, GTI 2500 vector display
1976 GTI 3500 standalone Logo computer
1977 Brookline project
1978 Work starts on TI LOGO

Logo on personal computers, 1980—2008

1980 *Mindstorms* (Papert); *Turtle Geometry* (Abelson and diSessa)
1981 MIT LOGO FOR APPLE II (Terrapin), TI LOGO released, APPLE LOGO, IBM LOGO (LCSI)
1982–84 Atari Cambridge Research Lab
1982 *Byte* Logo issue, *Logo for the Apple II* (Abelson)
1983 BOXER, TLC LOGO (beginning of OOP in personal computer Logo);
British Logo Users Group (BLUG);
Learning with Logo (Watt); *Learning and Teaching with Computers* (O'Shea & Self)
1984–86 MIT Logo conferences
1984 *Thinking about TLC Logo* (Allen et al.)
1985 LOGOWRITER; *Computer Science Logo Style* vol. 1 (Harvey)
1985–91 a flood of post-elementary Logo books
1986 OBJECT LOGO; *Constructionism*; *Teaching with Logo* (Watt and Watt)
1987 LEGO TC LOGO; first Eurologo conference
1988 BERKELEY LOGO; *Computer Environments for Children* (Solomon)
1989 *LOGO
1993 MICROWORLDS
1994 *The Children's Machine* (Papert); *Turtles, Termites, and Traffic Jams* (Resnick)
1996 *The Connected Family* (Papert);
Windows on Mathematical Meaning (Hoyles and Noss)
1998 TOONTALK
2005 IMAGINE LOGO

Logo's children, 2004–∞

2004 SCRATCH
2005 TURTLEART
2009 APP INVENTOR
2010 BYOB/SNAP*!*; POCKET CODE; first Constructionism conference (Eurologo renamed)
2019 First Snap*!* conference

B LOGO PUBLICATIONS

B.1 Logo Books: Beyond Square-Triangle-House

- Papert's most influential book was *Mindstorms* [Papert 1980], which introduced both the Logo classroom and turtle geometry to teachers. It was a required textbook in teacher credentialing coursework in computer education for many years. The others are *The Children's Machine* [Papert 1994] and *The Connected Family* [Papert 1996].

- From the beginning there were curriculum materials for Logo beginners, in the form of handouts made by individual teachers. Once the commercially distributed microcomputer implementations of Logo appeared in the 1980s, versions of the curriculum for beginners were embodied in books distributed with the Logo software [Abelson 1982; Berger et al. 1988; Solomon 1983].

  Daniel Watt wrote a textbook for beginners not tied to a microcomputer version, making a more serious effort than most to reproduce the ideal Logo classroom in a book [Watt 1983].

- Because even very young children could program in Logo, in the early days many people thought that Logo was *only* for young beginning programmers. A second wave of curriculum tried to overcome this by addressing a range of ages from middle school to university [Boecker et al. 1991; Friendly 1988; Thornburg 1986]. Another kind of book was a collection of projects with commentary [Birch 1986; Solomon et al. 1986].

  The MIT Press published a particularly ambitious series of advanced Logo-based curricula, starting with *Turtle Geometry* [Abelson and diSessa 1980]. The series spanned a range of topics including computer science [Harvey 1985, 1986, 1987], linguistics [Goldenberg and Feurzeig 1987], visual modeling [Clayson 1988], algebra [Cuoco 1990], and pre-calculus [Lewis 1990].

  Most books that are tied to particular versions of Logo differ from each other only in notational details. But when the Logo dialect is far from the mainstream, the associated books teach novel ideas well worth reading even if you're not programming in that dialect. The most obvious example is *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds* [Resnick 1994], which describes StarLogo and uses it to ground a theory about emergent phenomena in several different contexts. Several NetLogo textbooks explore complex systems using agent-based modeling [Banos et al. 2015, 2016; O'Sullivan and Perry 2013; Railsback and Grimm 2019; Wilensky and Rand 2015]. Other books tied to specific dialects that are worth reading for the general ideas they introduce include *Thinking about TLCLogo: a graphic look at computing with ideas* [Allen et al. 1984] and *Object Logo* [Coral 1986].

- All of the above are addressed to students or self-directed learners. Another set of Logo books addressed teachers and education researchers [Hoyles and Noss 1992; Noss and Hoyles 1996; Solomon 1988; Watt and Watt 1986].

  Windows on Mathematical Meanings: Learning Cultures and Computers [Noss and Hoyles 1996] is an excellent example of theoretical and empirical Logo research. It includes several in-depth studies of mathematics learning in narrowly focused Logo microworlds. It also provides insight into the variety of ways that teachers and schools incorporated Logo in UK classrooms in the 1980s and 1990s.

  There is a large literature on the use of computers in education more generally. Logo features in many such books, especially those starting from an artificial intelligence perspective [Lawler and Yazdani 1987; Yazdani 1984; Yazdani and Lawler 1991]. Edinburgh Logo and its role in teaching secondary mathematics is discussed in works by Tim O'Shea and colleagues [Howe and O'Shea 1978], [O'Shea and Self 1983]. David Sewell wrote a review of educational

computing from the perspective of cognitive psychology, generally friendly to Logo but with some criticism of early claims that the language itself would magically transform learning [Sewell 1990].

B.2 Logo Conference Proceedings

Much of the history of Logo development and of teaching with Logo is captured in conference proceedings. Here are some of them:

- Three Logo conferences were held at MIT: Logo84, Logo85, and Logo86. There was also a *Constructionist Learning* symposium in 1990 as part of AERA [Harel 1990].
- A continuing series of mostly-biennial Logo conferences has had two name changes during its history. There were three *British Logo Users Group* conferences, 1983–85. Participation in BLUG grew beyond Britain, and so the conference was renamed *Eurologo*. There were 11 conferences with that name, 1987–2007, four of which have proceedings online: [Eurologo 1997, 2001, 2005, 2007]. The organizers of what would have been Eurologo 2009 felt that the conference should appeal to people beyond Logo educators, and so they took an extra year off for planning and changed the name to *Constructionism*. There have been six conferences by that name so far, 2010–18 [Constructionism 2010, 2012, 2014, 2016, 2018].
- There have also been many Logo-related papers at non-Logo-specific conferences, such as the World Conference on Computers in Education (WCCE), Psychology of Mathematics Education (PME), Computer Using Educators (CUE), and many others. We have not tried to hunt these down.

B.3 Logo Periodicals

What follows is an incomplete list of Logo journals and newsletters. In particular, it's incomplete because the present authors are all primarily English speakers.

- *Logo Update* was published from 1993 to 2001 by the Logo Foundation, run by Michael Tempel. The organization still exists, although the newsletter is defunct. All issues are available at the Logo Foundation web site [Foundation 2001].
- In September, 1982, Tom Lough started *The National Logo Exchange* with Steve Tipps and Glen Bull as a monthly newsletter for Logo teachers and parents. In January, 1986 *The International Logo Exchange* was launched with Dennis Harper as the editor-in-chief. In September, 1986 these two publications were combined and renamed *Logo Exchange*. The International Council for Computers in Education (ICCE) acquired the publication in 1987, designating it as the official journal of the ICCE Special Interest Group for Logo-Using Educators (SIG-Logo). In 1989 ICCE was renamed the International Society for Technology in Education (ISTE). Logo Exchange continued as the ISTE journal for SIG-Logo until the fall of 1999, when the SIG was dissolved. The Logo Foundation web site includes the complete collection of 117 issues [Exchange 1999].
- The British Logo Users Group published a more-or-less-annual *Logo Almanack* in the autumn[20], and occasional *Logos* in summer or winter. The first *Almanack* was in 1983. In 1992, a new publication, *Eurologos Incorporating Logo Almanack* heralded the first Eurologo conference, and was "published by the British Logo Users Group for the scientific committee of Eurologo." Later volumes were simply titled *Eurologos*.
- *The Council for Logo in Mathematics Education* (CLIME), founded by Ihor Charischak, issued its first Newsletter in 1987. In 1988, CLIME became an affiliate of the National Council of Teachers of Mathematics. CLIME published *CLIME News* and *CLIME Connections*, with articles

[20] In honor of the Kinks song?

about teaching math with Logo, and *CLIME Microworlds,* with a diskette of Logo programs along with a paper collection of articles about using the programs to provide mathematical microworlds for math teaching and learning.
- *Kaleidoscopes,* published from 1985 to 1987, was largely, but not entirely, oriented toward mathematics in Logo. It was published by Computers for a New Education (Alison Birch, Larry Davidson, and Phil Lewis, all of whom are authors of books mentioned earlier).
- *Byte* magazine, which at the time published an annual programming language issue, published its Logo issue August, 1982 [Byte 1982].

B.4 Logo-Related Web Sites

- Gary Stager runs a web site devoted to Seymour Papert's speeches and writings: http://dailypapert.com
- Michael Tempel runs the Logo Foundation, which offers workshops for teachers and other resources: https://el.media.mit.edu/logo-foundation
- Here's the Wikipedia article on Logo: https://en.wikipedia.org/wiki/Logo_(programming_language)
- Companies and nonprofits that offer Logo products have web sites: microworlds.com, terrapin.com, people.eecs.berkeley.edu/~bh/logo.html, and many others, which can be found in the Logo Tree (http://elica.net/download/papers/logotreeproject.pdf).
- This is a 2014 online magazine article about Logo, with pointers to several other web-accessible resources: https://www.kidscodecs.com/logo-programming-language/
- One phenomenon of the '80s was web rings, collection of web sites on a particular topic that were arranged in a ring, so that each site had pointers to the next and previous sites in the ring. Many of the sites in the Logo ring are no longer maintained, but some are. Here's the list: https://www.ringsurf.com/ring/logoring/

B.5 Other Histories of Logo

In 1984 Feurzeig, one of the originators of Logo, published a short history of Logo's early days [Feurzeig 1984]. In 1997, Angelos Agalianos submitted a doctoral thesis to the Institute of Education, University of London on the history of Logo [Agalianos 1997]. She interviewed 23 Logo researchers and developers including Bobrow and Feurzeig. In 1999, students of MIT's course 6.933J wrote a history of Logo based upon interviews with five of Logo's early developers [Chakraborty et al. 1999]. The Logo Foundation has maintained a history of Logo on their website for decades [Foundation 2020].

C ABOUT THE AUTHORS

Cynthia Solomon was one of the creators of Logo, starting in 1966 with Seymour Papert, Daniel Bobrow, and Wallace Feurzeig at BBN. Solomon and Papert then took Logo research to the MIT AI Lab. Solomon was a founder of Logo Computer Systems, Inc (LCSI), and directed the development of Apple Logo, the first commercial version of Logo. She was Director of the Atari Cambridge Research Lab, where the focus was on building an object-oriented Logo with integrated turtle graphics and music.

Brian Harvey is the author of the three-volume *Computer Science Logo Style* [Harvey 1985, 1986, 1987]; the lead developer of Berkeley Logo [Harvey 1988]; a co-developer, with Jens Mönig, of Snap*!* [Mönig and Harvey 2020], a visual language with first class procedures, lists, and sprites; and a lead developer of *The Beauty and Joy of Computing* [Garcia et al. 2019], a high school curriculum using Snap*!*.

While **Ken Kahn** was a doctoral student at the MIT AI Lab he joined the Logo Group part-time, beginning in 1975. He helped with teaching, and developed layers of software on top of Logo to support student projects involving natural language or animation. Later he developed several programming languages for children directly inspired by Logo.

Henry Lieberman joined the MIT AI Lab Logo Group in 1971 as a student researcher, and was a full-time staff member from 1974 until 1977. He worked with Ira Goldstein on the Lisp implementation of Logo, on the first bitmap, color, and 3D graphics systems for Logo, on several microworlds, and taught high school students. He was also a consultant to Cynthia Solomon's Atari lab.

Mark L. Miller joined the MIT Logo Group in 1972. His PhD thesis in AI showed how intelligent programming environments might help students learn powerful ideas about planning and debugging. He studied Logo teaching styles, and wrote graphics and music applications. He joined Texas Instruments in 1978, and, with Papert, helped develop their plans for a future Logo computer.

Margaret Minsky learned Logo as a child. She was an assistant teacher at the Logo Exeter Exhibition in 1972, and as an undergraduate taught Logo and Physics in an NSF-sponsored summer program for high schoolers. She served on the MIT Logo Group staff until 1979, and the LCSI Apple Logo development team, 1981-1983. She co-edited a project book, *LogoWorks*.

Artemis Papert first encountered Logo as a child. She is an artist creating art in both traditional and digital media. Her digital art is created using the TurtleArt software, for which she has created a lot of its artistic "literature." After a first career as a research biologist she retrained in the healing art of shiatsu. Artemis has led TurtleArt workshops for a wide variety of groups in many countries.

Brian Silverman led the research and development at LCSI in the 1980s and 1990s. Before that he was an undergraduate at MIT and spent time at the Logo Group. After that he was part of the early Scratch design team.

REFERENCES

Harold Abelson. 1982. *Apple Logo.* BYTE/McGraw-Hill.

Harold Abelson and Andrea A diSessa. 1980. *Turtle geometry: The computer as a medium for exploring mathematics.* MIT press, Cambridge, MA.

Harold Abelson, Gerald Jay Sussman, and Julie Sussman. 1984. *Structure and Interpretation of Computer Programs.* The MIT Press, Cambridge, MA. https://web.archive.org/web/20200129045953/https://mitpress.mit.edu/sites/default/files/sicp/full-text/book/book.html Also at (NON-ARCHIVAL) https://mitpress.mit.edu/sites/default/files/sicp/full-text/book/book.html.

Angelos S Agalianos. 1997. *A cultural studies analysis of logo in education.* Ph.D. Dissertation. Institute of Education, University of London. https://web.archive.org/web/20200301032707//id/eprint/10018887/7/Agalianos,%20Angelos%20S.pdf

John R Allen, Ruth E Davis, and John F Johnson. 1984. *Thinking about TLCLogo: a graphic look at computing with ideas.* Henry Holt & Co.

Apple. 1993. AppleScript. https://web.archive.org/web/20200221151316/https://en.wikipedia.org/wiki/AppleScript

Autodesk. 1982. AutoCAD. https://web.archive.org/web/20200225122817/https://www.autodesk.com/products/autocad/overview

Ron Baecker. 1975. Two Systems Which Produce Animated Representations of the Execution of Computer Programs. *SigCSE Bulletin* (1975). https://doi.org/10.1145/800284.811152

Arnaud Banos, Christophe Lang, and Nicolas Marilleau. 2015. *Agent-based spatial simulation with NetLogo.* Vol. 1. Elsevier.

Arnaud Banos, Christophe Lang, and Nicolas Marilleau. 2016. *Agent-based Spatial Simulation with NetLogo, Volume 2: Advanced Concepts.* Elsevier.

Andrew Begel. 1996. LogoBlocks: A Graphical Programming Language for Interacting with the World. (1996). https://web.archive.org/web/20180327052635/http://andrewbegel.com/mit/begel-aup.pdf Also at NON-ARCHIVAL https://andrewbegel.com/mit/begel-aup.html.

Christopher Beland, Wesley Chan, Dwaine Clarke, Richard Park, and Michael Trupiano. 2000. LEGO Mindstorms: The Structure of an Engineering '('R')'evolution. (2000). https://web.archive.org/web/20200130053944/http://web.mit.edu/6.933/www/Fall2000/LegoMindstorms.pdf Also at NON-ARCHIVAL http://web.mit.edu/6.933/www/Fall2000/LegoMindstorms.pdf.

Anne R. Berger, Richard C. Carter, Ross J. Harris, Carolyn K. H. Ing, Mary Jo Moore, and Peter von Mertens. 1988. *PC Logo Tutorial.* Harvard Associates.

Alison Birch. 1986. *The Logo Project Book: Exploring Words and Lists.* Terrapin, Cambridge, MA.

Steven Bird, Mak Nazečić-Andrlon, and Jarred Gallina. 2013. Edgy. https://web.archive.org/web/20190723115758/http://snapapps.github.io/edgy/ Also at NON-ARCHIVAL https://snapapps.github.io/edgy/.

Andrej Blaho and Ivan Kalas. 2001. Object Metaphor Helps Create Simple Logo Projects. In *Proc. of Eurologo 2001.* Oesterreichische Computer Gesellschaft, Linz, 55–65. https://web.archive.org/web/20200130055225/https://www.ocg.at/sites/ocg.at/files/EuroLogo2001/K12Blaho_Kalas.pdf

Andrej Blaho, Ivan Kalas, and Monika Matusova. 1994. Environment for Environments: A New Metaphor for Logo. In *Proceedings of the IFIP TC3/WG3.5 International Working Conference on Exploring a New Partnership: Children, Teachers and Technology.* Elsevier Science Inc., USA, 153–166. https://dl.acm.org/citation.cfm?id=717322

Andrej Blaho, Ivan Kalas, and Peter Tomcsanyi. 1993. Comenius Logo: Environment for Teachers and Environment for Learners. In *Proc. of 4th EuroLogo Conference, Supplement.* Athens, 1–11.

Andrej Blaho, Ivan Kalas, and Peter Tomcsanyi:. 2000. Imagine-nowa generacja srodowiska tworzcego uczenia sie (Imagine - New Generation Of Creative Environment For Learning). In *Informatyka w szkole.* IIU, Wroclaw.

Heinz-Dieter Boecker, Harold Eden, and Gerhard Fischer. 1991. *Interactive Problem Solving Using Logo.* LEA.

Paula Bontá, Artemis Papert, and Brian Silverman. 2010. Turtle, Art, TurtleArt. In *Proc. of Constructionism 2010 Conference.* Paris.

Pavel Boytchev. 2014. Logo Tree Project. (2014). https://web.archive.org/web/20180820132053/http://elica.net/download/papers/LogoTreeProject.pdf Rev. 2.13. Also at (NON-ARCHIVAL) urlhttp://elica.net/download/papers/logotreeproject.pdf.

Byte. 1976. About the Cover... and the Contest. *Byte* (December 1976).

Byte. 1982. *Byte Magazine Logo issue.* Vol. 7. McGraw-Hill, USA. Issue 8.

Cambridge Conference on School Mathematics. 1963. *Goals for School Mathematics.* Houghton Mifflin. https://web.archive.org/web/20100627005713/http://csmc.missouri.edu/CCM/cambridge.php Also at (NON-ARCHIVAL) http://csmc.missouri.edu/CCM/cambridge.php.

Anit Chakraborty, Randy Graebner, and Tom Stocky. 1999. LOGO: A project history. https://web.archive.org/web/20031130144606/http://web.mit.edu/6.933/www/LogoFinalPaper.pdf Unpublished paper for MIT's Course 6.993J.

James Clayson. 1988. *Visual Modeling with Logo.* MIT Press, Cambridge, MA.

Douglas H. Clements and Dominic Gullo. 1984. Effects of computer programming on young children's cognition. *Journal of Educational Psychology* 76, 6 (1984), 1051–1058. https://psycnet.apa.org/doi/10.1037/0022-0663.76.6.1051

Allan Collins, Eleanor H. Warnock, Nelleke Aiello, and Mark L. Miller. 1975. Reasoning from incomplete knowledge. In *Representation and Understanding: Studies in Cognitive Science*, Allan Collins and Daniel G. Bobrow (Eds.). Academic Press, Inc., New York, NY, 383–416.

Constructionism. 2010. *Proc. of Constructionism 2010 Conference.* Paris. https://web.archive.org/web/20170223053752/http://etl.ppp.uoa.gr/constructionism2010/constructionism_2010.zip Also at (NON-ARCHIVAL) http://etl.ppp.uoa.gr/constructionism2010/constructionism_2010.zip.

Constructionism. 2012. *Proc. of Constructionism 2012 Conference.* Athens. https://web.archive.org/web/20171020205040/http://constructionism2012.etl.ppp.uoa.gr/-pid=31.htm Also at (NON-ARCHIVAL) http://constructionism2012.etl.ppp.uoa.gr/-pid=31.htm.

Constructionism. 2014. *Proc. of Constructionism 2014 Conference.* Vienna. https://web.archive.org/web/20190718074244/http://constructionism2014.ifs.tuwien.ac.at/Schedule.html Also at (NON-ARCHIVAL) http://constructionism2014.ifs.tuwien.ac.at/Schedule.html.

Constructionism. 2016. *Proc. of Constructionism 2016 Conference.* Bangkok. https://web.archive.org/web/20160925045403/http://e-school.kmutt.ac.th/constructionism2016/?p=772 Also at (NON-ARCHIVAL) http://e-school.kmutt.ac.th/constructionism2016/?p=772.

Constructionism. 2018. *Proc. of Constructionism 2018 Conference.* Vilnius. https://web.archive.org/web/20190331064827/http://www.constructionism2018.fsf.vu.lt/proceedings/ Also at (NON-ARCHIVAL) http://www.constructionism2018.fsf.vu.lt/proceedings/.

Coral. 1986. *Object Logo.* Coral Software Corp.

David Costanzo. 2005. *FMSLogo.* Retrieved August 11, 2019 from https://web.archive.org/web/20200122210105/http://fmslogo.sourceforge.net/ Also at (NON-ARCHIVAL) urlhttp://fmslogo.sourceforge.net/.

Albert Cuoco. 1990. *Investigations in Algebra.* MIT Press, Cambridge, MA.

Andrea A diSessa and Harold Abelson. 1986. Boxer: A Reconstructible Computational Medium. *Commun. ACM* 29, 9 (Sept. 1986), 859–868. https://doi.org/10.1145/6592.6595

Gary Drescher. 1987. Object Logo. In *Artificial Intelligence and Education*, R. Lawler and M. Yazdani (Eds.). Ellis Horwood. https://doi.org/10.1145/2048147.2048218

Mark Eisenstadt and Mike Brayshaw. 1988. The transparent Prolog machine (TPM): an execution model and graphical debugger for logic programming. *Journal of Logic Programming* 5, 4 (1988), 277–342.

Eurologo. 1997. *Proc. of Eurologo 1997 Conference.* Budapest. https://web.archive.org/web/20171018181117/http://eurologo.web.elte.hu/prog.htm Also at (NON-ARCHIVAL)http://eurologo.web.elte.hu/prog.htm.

Eurologo. 2001. *Proc. of Eurologo 2001 Conference.* Oesterreichische Computer Gesellschaft, Linz. https://www.ocg.at/sites/ocg.at/files/EuroLogo2001/

Eurologo. 2005. *Proc. of Eurologo 2005 Conference.* Warsaw. https://web.archive.org/web/20160316192152/https://eurologo2005.oeiizk.waw.pl/fullpapers.php Also at (NON-ARCHIVAL) https://eurologo2005.oeiizk.waw.pl/fullpapers.php.

Eurologo. 2007. *Proc. of Eurologo 2007 Conference.* Bratislava. https://web.archive.org/web/20080411233516/http://www.di.unito.it/~barbara/MicRobot/AttiEuroLogo2007/proceedings/authors.html Also at (NON-ARCHIVAL) http://www.di.unito.it/~barbara/MicRobot/AttiEuroLogo2007/proceedings/authors.html.

Logo Exchange. 1982-1999. *Logo Exchange.* https://web.archive.org/web/20191029145937/https://el.media.mit.edu/logo-foundation/resources/nlx/index.html

Wally Feurzeig. 1984. The Logo Lineage. *Digital deli* (1984). https://web.archive.org/web/20031018181133/https://www.atariarchives.org/deli/logo.php

Wallace Feurzeig. 2006. Educational technology at BBN. *IEEE Annals of the History of Computing* 28, 1 (2006), 18–31.

Logo Foundation. 1993-2001. *Logo Foundation.* Retrieved August 11, 2019 from https://web.archive.org/web/20200308051245/https://el.media.mit.edu/logo-foundation/index.html

Logo Foundation. 2020. *Logo History.* https://web.archive.org/web/20200103033813/https://el.media.mit.edu/logo-foundation/what_is_logo/history.html

Neil Fraser. 2015. Ten things we've learned from Blockly. In *2015 IEEE Blocks and Beyond Workshop.* 49–50. http://ieeexplore.ieee.org/document/7369000/

Free Software Foundation. 1985. Emacs. https://web.archive.org/web/20200228113711/https://www.gnu.org/software/emacs/

Michael Friendly. 1988. *Advanced Logo.* LEA.

Daniel D. Garcia, Brian Harvey, E. Paul Goldenberg, June Mark, Mary Fries, Albert Cuoco, Jane Kang, and Selim Tezel. 2019. *The Beauty and Joy of Computing.* University of California, Berkeley, and Education Development Center, Inc. https://web.archive.org/web/20190829095245/https://bjc.berkeley.edu/ and https://web.archive.org/web/20190711234424/https://bjc.edc.org/.

Daniel D. Garcia, Brian Harvey, and Luke Segars. 2012. CS principles pilot at University of California, Berkeley. *ACM Inroads* 3 (06 2012). https://doi.org/10.1145/2189835.2189853

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

Ira Goldstein. 1973. Germland. https://doi.org/10.5281/zenodo.3719335 MIT Logo Group Working Paper 7.

Ira Goldstein. 1975. Summary of MYCROFT: A System for Understanding Simple Picture Programs. *Artificial Intelligence* 6, 3 (September 1975), 249–288. https://doi.org/10.1016/0004-3702(75)90003-X

Idit Harel (Ed.). 1990. *Constructionist Learning.* Massachusetts Institute of Technology, Media Laboratory, Cambridge, MA.

Brian Harvey. 1985. *Computer Science Logo Style, v.1: Symbolic Computing.* MIT Press, Cambridge, MA. https://people.eecs.berkeley.edu/~bh/v1-toc2.html Second edition, 1997.

Brian Harvey. 1986. *Computer Science Logo Style, v.2: Advanced Techniques.* MIT Press, Cambridge, MA. https://people.eecs.berkeley.edu/~bh/v2-toc2.html Second edition, 1997.

Brian Harvey. 1987. *Computer Science Logo Style, v.3: Beyond Programming.* MIT Press, Cambridge, MA. https://people.eecs.berkeley.edu/~bh/v3-toc2.html Second edition, 1997.

Brian Harvey. 1988. *Berkeley Logo.* University of California, Berkeley. https://web.archive.org/web/20190927020220/https://people.eecs.berkeley.edu/~bh/logo.html

Brian Harvey. 1991. Symbolic Programming vs. the AP Curriculum. *The Computing Teacher* (1991). https://people.eecs.berkeley.edu/~bh/bridge.html

Brian Harvey. 2019. *"Why do we have to learn this baby language?".* https://web.archive.org/web/20200207073819/https://snap.berkeley.edu/baby3.pdf

Brian Harvey and Jens Mönig. 2010. Bringing "No Ceiling" to Scratch: Can One Language Serve Kids and Computer Scientists?. In *Constructionism 2010.* Paris.

Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proc. International Joint Conference on Artificial Intelligence.*

W. Daniel Hillis (Ed.). 1986. *The Connection Machine.* MIT Press, Cambridge, MA, USA.

Jim Howe and Tim O'Shea. 1978. Learning Mathematics Through LOGO. *SIGCUE Outlook* 12, 1 (Jan. 1978), 2–11. https://doi.org/10.1145/963847.963848

Celia Hoyles and Richard Noss. 1992. *Learning Mathematics and Logo.* MIT Press, Cambridge, MA.

INRP. 1981. *Pratique active de l'informatique par l'enfant.* Institut National de la Recherche Pédagogique, France. https://web.archive.org/web/20180712173526/http://lara.inist.fr/bitstream/handle/2332/1248/INRP_RP_81_111.pdf

Ken Kahn. 1975. A logo Natural Language System. MIT Logo Group Working Paper 46.

Kenneth Kahn. 1976. An Actor-Based Computer Animation Language. In *Proceedings of the ACM/SIGGRAPH Workshop on User-oriented Design of Interactive Graphics Systems (UODIGS '76)*. ACM, New York, NY, USA. https://doi.org/10.1145/1024273.1024278 Revision of: LOGO Working Paper 48, AI Working Paper 120, MIT, February 1976. Also published in: Creative Computing, Vol. 6, No. 11, Nov. 1980, pp. 75-84.

Kenneth Kahn. 1977. Three Interactions between Al and Education. *Machine Intelligence* 8 (1977).

Ken Kahn. 1979a. Director Guide. MIT AI Lab memo 482b. https://dspace.mit.edu/handle/1721.1/6302

Kenneth Kahn. 2001. ToonTalk and Logo: Is ToonTalk a Colleague, Successor, Sibling, or Child of Logo?. In *Eurologo 2001*. Oesterreichische Computer Gesellschaft, Linz. http://www.toontalk.com/Papers/logott.pdf

Ken Kahn. 2007. Should LOGO keep going forward 1? *Informatics in Education-An International Journal* 6, 2 (2007), 307–321.

Ken Kahn, Evgenia Sendova, Ana Isabel Sacristán, and Richard Noss. 2011. Young students exploring cardinality by constructing infinite processes. *Technology, Knowledge and Learning* 16, 1 (2011), 3–34.

Kenneth M Kahn. 1979b. *Creation of computer animation from story descriptions*. Ph.D. Dissertation. Massachusetts Institute of Technology.

Kenneth M Kahn and Vijay A Saraswat. 1990. Complete visualizations of concurrent programs and their executions. In *Proceedings of the 1990 IEEE Workshop on Visual Languages*. IEEE, 7–15.

Ivan Kalas and Andrej Blaho. 1994. Not All Birds Are Turtles: Developing Teaching / Learning Environments in Initial Teacher Training at Comenius University. In *Information Technologies in Teacher Education: Issues and Experiences for Countries in Transition, Proceedings of a European Workshop, University of Twente*. Enschede, Netherlands, 222–237. http://www.unesco.org/education/information/pdf/412_37.pdf

Alan C. Kay. 2013. Afterword: What Is A Dynabook? Commentary on "A Personal Computer For Children Of All Ages". https://web.archive.org/web/20190128191307/http://www.vpri.org/pdf/hc_what_Is_a_dynabook.pdf

Eric Klopfer, Hal Scheintaub, Wendy Huang, Daniel Wendel, and Ricarose Roque. 2009. The Simulation Cycle: Combining Games, Simulations, Engineering and Science Using StarLogo TNG. *E-Learning and Digital Media* 6, 1 (2009), 71–96. https://doi.org/10.2304/elea.2009.6.1.71

Duks Koschitz and Eric Rosenbaum. 2012. Exploring algorithmic geometry with 'Beetle Blocks:' A graphical programming language for generating 3D forms. In *Proceedings of the 15th International Conference on Geometry and Graphics*. 380–389.

Chronis Kynigos, Manolis Koutlis, Kriton Kyrimis, George Tsironis, Giorgos Vasiliou, and George Birbilis. 2000. *E-Slate*. https://web.archive.org/web/20190906073941/http://e-slate.cti.gr/

Robert W. Lawler and Masoud Yazdani (Eds.). 1987. *Artificial Intelligence and Education, vol. 1: Learning Environments and Tutoring Systems*. Ablex Publishing Corp., Norwood, NJ, USA.

Akos Ledeczi and Brian Broll. 2016. NetsBlox. https://web.archive.org/web/20191130183352/https://netsblox.org/

Steven Levy. 1984. *Hackers: Heroes of the Computer Revolution*. Anchor Doubleday.

Philip G. Lewis. 1990. *Approaching Precalculus Mathematics Discretely*. MIT Press, Cambridge, MA.

Henry Lieberman. 1986. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In *Conference proceedings on Object-oriented programming systems, languages and applications (OOPSLA 86)*, Norman Meyrowitz (Ed.). ACM, New York, NY, USA, 214–223. https://doi.org/10.1145/28697.28718

Henry Lieberman and Christopher Fry. 1995. Bridging the gulf between code and behavior in programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI 95)*, Irvin R. Katz, Robert Mack, Linn Marks, Mary Beth Rosson, and Jakob Nielsen (Eds.). ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 480–486. https://doi.org/10.1145/223904.223969

John Maloney, Leo Burd, Yasmin Kafai, Natalie Rusk, Brian Silverman, and Mitchel Resnick. 2004. Scratch: A sneak preview. Paper published in Creating, Connecting and Collaborating through Computing. In *Proceedings for the second International Conference of the Institute of Electrical and Electronics Engineers*.

Andrea Mayr-Stalder and Michael Aschauer. 2016. TurtleStitch. https://web.archive.org/web/20200103032255/https://www.turtlestitch.org/

Warren S. McCulloch and Walter Pitts. 1943. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics* 5 (1943), 115–133. Issue 4. https://doi.org/10.1007/BF02478259

Andreas Micheler. 2004. *aUCBLogo*. https://web.archive.org/web/20190811211744/http://www.aucblogo.org/en/Logo.html

Mark L. Miller. 1979. A Structured Planning and Debugging Environment for Elementary Programming. *International Journal of Man-Machine Studies* XI, 1 (1979), 79–95.

Mark L. Miller and Ira P. Goldstein. 1977. Structured Planning and Debugging. In *IJCAI'77, Proceedings of the 5th International Joint Conference on Artificial Intelligence*, Vol. 2. Morgan Kaufmann, Cambridge, MA, 773–779.

George Mills. 2016. *MSWLogo*. https://web.archive.org/web/20200224081900/http://www.softronix.com/logo.html

Marvin Minsky. 2019a. The Infinite Construction Kit. In *Inventive Minds*, Cynthia Solomon and Xiao Xiao (Eds.). MIT Press, Cambridge, MA. https://web.archive.org/web/20020209183902/https://web.media.mit.edu/~minsky/papers/Logoworks.html Also published as preface to LogoWorks: Challenging Programs in Logo, Cynthia Solomon, Margaret Minsky, and

Brian Harvey, eds. McGraw-Hill 1986.
Marvin Minsky. 2019b. *Inventive minds: Marvin Minsky on education.* MIT Press, Cambridge, MA. Cynthia Solomon and Xiao Xiao (Eds.).
Marvin Minsky and Seymour Papert. 1969. *Perceptrons: An Introduction to Computational Geometry.* The MIT Press, Cambridge, MA. Reissued as The 1988 Expanded Edition With A New Foreword By Léon Bottou in 2017.
MIT Scheller Teacher Education Program. 2019. StarLogo Nova. https://web.archive.org/web/20190716024204/https://www.slnova.org/
Jens Mönig and Brian Harvey. 2020. *Snap! 5.0.* University of California, Berkeley. https://web.archive.org/web/20191227022839/https://snap.berkeley.edu/
Yishay Mor, Richard Noss, Ken Kahn, Celia Hoyles, and Gordon Simpson. 2004. Thinking in progress. *Micromath* 20, 2 (2004).
Leonel Caseiro Morgado. 2015. *Framework for computer programming in preschool and kindergarten.* Ph.D. Dissertation. Universidade de Tras-os-Montes e Alto Douro (Portugal).
Allen Newell. 1972. *Human Problem Solving.* Prentice-Hall, Inc., USA.
Richard Noss, Lulu Healy, and Celia Hoyles. 1997. The Construction of Mathematical Meanings: Connecting the Visual with the Symbolic. *Educational Studies in Mathematics* 33, 2 (July 1997), 203–233. https://www.jstor.org/stable/3482643
Richard Noss and Celia Hoyles. 1996. *Windows on Mathematical Meanings: Learning Cultures and Computers.* Springer Netherlands, Heidelberg, Germany. https://books.google.com/books?id=VclPmvPRbSAC
Tim O'Shea and John Self. 1983. *Learning and Teaching with Computers: Artificial Intelligence in Education.* Harvester Press, Hemel Hempstead, UK. 307 pages. https://books.google.com/books?id=2EubQgAACAAJ
David O'Sullivan and George LW Perry. 2013. *Spatial simulation: exploring pattern and process.* John Wiley & Sons.
Papert, Fonseca, Almeida, Kozberg, Tempel, Soprunov, Yakovleva, Reggini, Richardson, Almeida, and Cavallo. 1999. *Logo Philosophy and Implementation.* Logo Computer Systems INC., Westmount, QC, Canada. http://www.microworlds.com/company/philosophy.pdf
Seymour Papert. 1972a. Teaching Children Thinking. *Programmed Learning and Educational Technology* 9, 5 (1972), 245–255. Previously published in the World Conference on Computer Education, Amsterdam, 1970, IFIPS.
Seymour Papert. 1972b. Teaching children to be mathematicians versus teaching about mathematics. *International journal of mathematical education in science and technology* 3, 3 (1972), 249–262.
Seymour Papert. 1980. *Mindstorms: Children, Computers, and Powerful Ideas.* Basic Books, Inc., New York, NY, USA.
Seymour Papert. 1988. The Conservation of Piaget: The Computer as Grist. *Constructivism in the computer age* (1988), 3–14.
Seymour Papert. 1994. *The Children's Machine: Rethinking School In The Age Of The Computer.* BasicBooks.
Seymour Papert. 1996. *The Connected Family: Bridging the Digital Generation Gap.* Longstreet Press.
Seymour Papert. 1999. The Century's Greatest Minds. *Time Magazine* (1999).
Seymour Papert et al. 1978. Interim Report of the LOGO Project in the Brookline Public Schools: An Assessment and Documentation of a Children's Computer Laboratory. Artificial Intelligence Memo No. 484. https://dspace.mit.edu/handle/1721.1/5744
Seymour Papert and MIT Epistemology & Learning Research Group. 1986. *Constructionism: A New Opportunity for Elementary Science Education.* Massachusetts Institute of Technology, Media Laboratory, Epistemology and Learning Group. https://books.google.com.sg/books?id=0N8-HAAACAAJ
Seymour Papert and Cynthia Solomon. 1971. Twenty Things To Do With A Computer. MIT AI Lab memo 248. https://dspace.mit.edu/handle/1721.1/5836
Seymour Papert, Daniel Watt, Andrea diSessa, and Sylvia Weir. 1979a. Final Report of the Brookline LOGO Project. Part II: Project Summary and Data. Artificial Intelligence Memo No. 545. https://dspace.mit.edu/handle/1721.1/6323
Seymour Papert, Daniel Watt, Andrea diSessa, and Sylvia Weir. 1979b. Final Report of the Brookline LOGO Project. Part III: Profiles of Individual Student's Work. Artificial Intelligence Memo No. 546. https://dspace.mit.edu/handle/1721.1/6323
Erik Pasternak, Rachel Fenichel, and Andrew N. Marshall. 2017. Tips for creating a block language with Blockly. In *2017 IEEE Blocks and Beyond Workshop.* 21–24. http://ieeexplore.ieee.org/document/8120404/
Roy D. Pea. 1983. Logo Programming and Problem solving. Bank Street College of Education, Technical Report 12.
Roy D. Pea, D. Midian Kurland, and Jan Hawkins. 1985. Logo and the Development of Thinking Skills. In *Children and Microcomputers: Research on the Newest Medium*, M. Chen and W. Paisley (Eds.). Sage, Chapter 9, 193–298.
Radia Perlman. 1974. TORTIS: Toddler's Own Recursive Turtle Interpreter System. MIT Logo memo 9, MIT AI Lab memo 311. https://dspace.mit.edu/handle/1721.1/6224
Radia Perlman. 1976. Using Computer Technology to Provide a Creative Learning Environment for Preschool Children. MIT AI Lab memo 360. https://dspace.mit.edu/handle/1721.1/5784
Jean Piaget. 1971. *Introduction to Genetic Epistemology.* New York: Norton.
Playful Invention Company. 2014. *PicoCricket (web page).* https://web.archive.org/web/20140812070053/http://www.playfulinvention.com/portfolio/the-picocricket-kit/

Steven F Railsback and Volker Grimm. 2019. *Agent-based and individual-based modeling: a practical introduction.* Princeton university press.

Eric Raymond. 1996. *The Hacker's Dictionary, Third Edition.* MIT Press, Cambridge, MA.

Mitchel Resnick. 1994. *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds.* MIT Press, Cambridge, MA. https://mitpress.mit.edu/books/turtles-termites-and-traffic-jams

Mitchel Resnick. 1996. Beyond the Centralized Mindset. *Journal of the Learning Sciences* 5, 1 (1996), 1–22. https://doi.org/10.1207/s15327809jls0501_1

Eric Rosenbaum, Duks Koschitz, and Bernat Romagosa. 2011. BeetleBlocks. https://web.archive.org/web/20110202110720/http://beetleblocks.com/ archived 2 February, 2011.

Jean-Jacques Rousseau. 1762. *Émile: ou De l'éducation.* Chez Jean Néaulme, libraire. https://books.google.com/books?id=Elu6g0qpyMMC

Vijay A. Saraswat. 1993. *Concurrent Constraint Programming.* MIT Press, Cambridge, MA.

Arthur Schopenhauer and Philipp Otto Runge. 2010. *On Vision and Colors.* Princeton Architectural Press, Princeton, NJ.

David F. Sewell. 1990. *New Tools for New Minds: A Cognitive Perspective on the Use of Computers with Young Children.* St. Martin's Press, Inc., New York.

Fatimata Seye Sylla. 1985. *Computers and literacy in Senegal.* Master's thesis. Massachusetts Institute of Technology. https://dspace.mit.edu/handle/1721.1/77676

Andrew Shalit, David Moon, and Orca Starbuck. 1996. *The Dylan Reference Manual.* Addison-Wesley.

Bruce Sherina, Andrea A. diSessa, David Hammer, and Bruce L Sherin. 1993. Dynaturtle revisited: Learning physics through the collaborative design of a computer model. *Interactive Learning Environments* 3 (1993), 91–118. https://doi.org/10.1080/1049482930030201

Wolfgang Slany. 2012. A mobile visual programming system for Android smartphones and tablets. In *VL/HCC*. IEEE, 265–266.

Wolfgang Slany, Kirshan Kumar Luhana, Matthias Mueller, Christian Schindler, and Bernadette Spieler. 2018. Rock Bottom, the World, the Sky: Catrobat, an Extremely Large-scale and Long-term Visual Coding Project Relying Purely on Smartphones. In *Constructionism 2018*. Vilnius, 104–119.

Cynthia Solomon. 1988. *Computer Environments for Children: A Reflection on Theories of Learning and Education.* MIT Press, Cambridge, MA.

Cynthia Solomon, Margaret Minsky, and Brian Harvey. 1986. *LogoWorks: Challenging Programs in Logo.* Byte/McGraw-Hill.

Cynthia J. Solomon. 1983. *Introduction to Computer Programming Through Turtle Graphics.* Atari/LCSI.

Guy Lewis Steele, Jr. 1977. Debunking the "Expensive Procedure Call" Myth or, Procedure Call Implementations Considered Harmful or, LAMBDA: The Ultimate GOTO. In *Proceedings of the 1977 Annual Conference* (Seattle, Washington) *(ACM '77)*. ACM, New York, NY, USA, 153–162. https://doi.org/10.1145/800179.810196

Gerald J Sussman. 1973. A Computational Model of Skill Acquisition. MIT AI Lab Technical Report 297. http://hdl.handle.net/1721.1/6894

Terrapin. 2020a. Terrapin Logo. https://web.archive.org/web/20200227075051/https://weblogo.terrapinlogo.com/ Also at (NON-ARCHIVAL) /https://weblogo.terrapinlogo.com.

Terrapin. 2020b. Terrapin Logo Manual. https://web.archive.org/web/20200228120021/https://doc.terrapinlogo.com/doku.php/weblogo:start Also at (NON-ARCHIVAL) https://weblogo.terrapinlogo.com/doc.

David D. Thornburg. 1986. *Beyond Turtle Graphics: Further Explorations of Logo.* Addison-Wesley.

Sherry Turkle and Seymour Papert. 1990. Epistemological Pluralism: Styles and Voices within the Computer Culture. *Signs: Journal of Women in Culture and Society* 16, 1 (1990), 128–157.

Jose Valente. 2020. personal communication, 1 Feb 2020.

Roger Wagner. 1989. HyperStudio. https://web.archive.org/web/20190717150236/https://www.mackiev.com/hyperstudio/index.html

David Walden, Raymond S Nickerson, and Nineteen Long-time BBN People. 2011. *A Culture of Innovation: Insider Accounts of Computing and Life at BBN.* Waterside Publishing.

Daniel Watt. 1983. *Learning with Logo.* Addison-Wesley.

Daniel Watt and Molly Watt. 1986. *Teaching with Logo.* Addison-Wesley.

Uri Wilensky and William Rand. 2015. *An introduction to agent-based modeling: modeling natural, social, and engineered complex systems with NetLogo.* Mit Press, Cambridge, MA.

David Wolber, Harold Abelson, and Mark Friedman. 2015. Democratizing Computing with App Inventor. *GetMobile: Mobile Comp. and Comm.* 18, 4 (Jan. 2015), 53–58. https://doi.org/10.1145/2721914.2721935

Masoud Yazdani (Ed.). 1984. *New Horizons in Educational Computing.* Ellis Horwood Ltd., Chichester, England.

Masoud Yazdani and Robert Lawler (Eds.). 1991. *Artificial Intelligence and Education, vol. 2: Principles and Case Studies.* Ablex Publishing Corp., Norwood, NJ, USA.