Logo at 50

Cynthia Solomon

Author of *Computer Environments for Children* and Co-author of Twenty Things to Do with a Computer

NCWIT Pioneer in Tech Award, 2016, National Center for Women and Information Technology, Las Vegas

Constructionism Lifetime Achievement Recognition, Constructionism 2016 Thailand

It’s July 1967 and Seymour Papert is teaching a brand new programming language called *Logo* to a group of children at the Hanscom School, Lincoln MA.  Wally Feurzeig and I are observers. Before and after each class Seymour and I hold debugging sessions. The result was a totally revised Logo ready for implementation on a time-shared Digital Equipment Corporation (DEC) PDP-1 at Bolt Beranek and Newman (BBN), Cambridge, MA. This system was ready for the school year 1968-69 when Seymour and I team-taught fourteen 7th graders at Muzzey Junior High School, Lexington, MA.

How Come Logo?

In 1966 Seymour consulted for Wally Feurzeig’s education group at BBN. Wally had a school project involving 5 different schools. The students were learning *Telcomp*, a programming language similar to *Basic*, as an aide to learning math. Seymour visited the classrooms and was struck by the absurdity of students learning *Telcomp*, an algebraic programming language, to help in learning algebra. Seymour declared what was needed was a programming language designed for children.  By the summer with discussions with Wally and others, Seymour spec’d out this new language and presented it to a small group of us. At that time I was a member of Wally’s group at *BBN*.

We all had a Lisp background so procedural thinking was a natural component. In fact, we initially thought of it as *Baby Lisp* or *Lisp without parentheses*. The group included Wally, Dan Bobrow, Dick Grant and me. Dan immediately started implementing Logo in *Lisp* on *BBN's SDS 940*. He gave it over to me and I gave it to Dick. The guiding idea was that Logo would be a language to play with words and sentences. I can hear Seymour say: “What is one of the chief activities of children? Why, playing with words and sentences.”  By summer 1967 we had a working Logo.

The Muzzey Experience

In September of 1968, fourteen 7th graders who were clustered in the average range started a year-long computer math course in lieu of their regular math. We had teletype terminals in a classroom devoted to this enterprise and connected from the school in Lexington, MA to the dedicated Logo time-shared *DEC PDP-1* at *BBN* in Cambridge, MA. I mention the computer was a dedicated Logo system because the user interface made it quite easy to log on and to save files. This class predated turtles. In fact, Seymour learned many things from this experience; and foremost, was the need for a concrete object to play with. The result was a *tethered turtle* on wheels that could be navigated from Logo. It also had a pen in its center. The name came from Grey Walter, a neurophysiologist and cyberneticist in England. He had made automatons, he called tortoises, Elmer and Elsie.

The unpublished and unfinished paper that follows was first written in 1970. The paper was intended to be a book on learning Logo based on our year-long teaching experience with 7th graders between 1968-69. It is shared in 2017 for the first time in a published form to shed light on the historical work we did that are the theoretical underpinnings for decades of work that became *computational objects to think with*, and *the theories behind Constructionism*,

Conceptual Advances Derived from the Muzzey Experiment

Seymour Papert and Cynthia Solomon

|  |  |
| --- | --- |
|  |  Seymour and I in the spring of 1969 looking at Cathy’s project printed on her teletype. |

1. Empirical vs. Conceptual Advances

It is necessary to clarify a methodological point about the relation of the expensive teaching experiment, to the ideas and teaching materials we developed during the year. The general pattern of evolution of the material is to pass through several stages from a clumsy initial idea to a much smoother integrated and elaborated form. In practice, we were not able to generate material of the same quality without the help of children. Subjectively, contact with the children seemed to produce a deeper sensitivity to their needs and reactions.

There is an analogy with the methodological problem of experiment in Piaget’s work on cognitive development. It seems obvious in retrospect that one could have known without specific experiments that children are not born for example, with the conservations (in Piaget’s sense). However, such phenomena are in fact never recognized in this a priori way, but rather emerge from contact between sensitive, imaginative investigators and children. Thus, the role of the “experiments” is not merely to provide evidence and answer previously well defined questions. They also provide, in a less well understood way, a basis for the development of intuitions. (The exact age distributions are, of course, more simply related to the empirical evidence. But the deepest insights in Piagetian psychology do not depend on such numbers.)

Be all this as it may, the fact is that our major output is rather literary in character consisting of exposition of ideas about teaching rather than of empirical truths that can be listed and verified statistically. There are, nevertheless, relatively minor points that can be so listed.

For example:

1. We did, of course, establish the possibility of teaching certain material to average seventh grade children. But this is an extremely weak assertion without either objective measures of the children’s performance or any formal way of taking into account special characteristics of the teachers.
2. It is interesting to observe that the class did not fall behind the control group in arithmetical performance as measured by the Iowa Tests.
3. We have a list of specific conceptual deficiencies we detected in the children and a (possibly more significant) list of abilities that might be surprising to some teachers. Among the latter we were particularly impressed (though, as usual, in retrospect we wonder why) by the ease with which the children assimilated numerous new mathematical functions. This might seem surprising when one considers the fuss traditional mathematics makes about teaching a small set of functions (arithmetical operations, trig functions, actors, log … and little else). It seems less surprising if one focuses on the fact that children learn new English words every day – and that the meanings of many of these are at least as complex in structure as simple mathematical functions.

2. The Evolution of Our Ideas on Teaching Method

2.1 Computer Culture

Before the Muzzey experiment we had some experience in teaching children to program in Logo and other languages. The operational status of the Muzzey class, however, forced us rather brutally to recognize misleading facets of  previous experience. The most important was that our personal teaching methods depended heavily on what we shall call our computer culture. The meaning we attach to the term will become clearer as we go along: it includes the effects of long experience seeing the world in terms of programs, so that we have at our fingertips many examples, analogies, ways of looking at programming, jokes, turns of phrase and other useful aids for developing a lively, stimulating and supple interaction with children. The Muzzey experience brought out this facet first by making us face the problems encountered by a teacher with very limited “computer culture” and then by putting a strain even on our own accumulated resources by the need to confront a class every day.

[*Note: A wonderful math teacher started as the teacher. She was not adept at thinking in terms of projects. Instead she emphasized the Logo language and syntax. After a month I took over. The class was in place of the children’s regular math and met 4 days a week. Seymour taught with me two days a week.*]

Our reaction to the problem was to recognize the need for an extensive formulation of teaching ideas, and an even more extensive formulation of relevant elements of “computer culture”. The following pages represent a first stab at this dual purpose To provide a consistent basis for choices of material and level of sophistication we have written them with an imaginary audience in mind consisting of a class of student teachers with no previous experience with computers. We discuss the programs and topics we have taught to children; but we present them in the form we now think, in the wisdom of hindsight, they ought to have been presented. We also include more comments, alternative forms, systematic classifications, and so on, than can be presented in one pass to an elementary class of children; teachers can and should know a little more than they teach.

Our discussion, especially side remarks, gives substance to the phrase “computer culture” and shows that it is misleading in its apparent restrictiveness. They are more concerned with the description and manipulation of concepts like “process”, “project”, “procedure”, and of course, “description”, “manipulation”, and “concept”.

2.2 The First Steps

The initial parts of the Muzzey course concentrated on teaching the meaning and syntax of Logo operations. Children worked with problems like:

 LAST OF FIRST OF BUTFIRST OF “THE GOOD CAT”

Answer: “D”

We think now that this was a mistake. It is possible to write exciting active programs with a small subset of Logo commands and without any idea that operations can be composed (or chained). Our present theory is that early teaching should ignore syntax and variety of operations and should concentrate on concepts that have an immediate pay-off in programming and in excitement for children.

Programs can be written on the very first day without prior knowledge of anything more than PRINT, the definition syntax, and the use of quotation. (Quotation can either be slipped in by saying we use it as in ‘He said, “Hi”’ or be prepared by the lesson on Quote and Slash described in the next section.)

LOGO commands can be introduced one at a time when they are needed. The chained form is best conceptualized as an abbreviated description of a program (in which each line performs one operation and each intermediate result is properly named.)

A suitable program for the first session is:

 TO GREET

 PRINT “HELLO”

 PRINT “GOODBYE”

 GREET

 END

The effect of this is shown by:

 HELLO

 GOODBYE

 HELLO

 GOODBYE

 HELLO

 GOODBYE

  .

  .

  .

And so on until the BREAK key is pressed.

With GREET as a model, children can be asked to make programs that will print:

 HI

 BYE

 HI

 BYE

  .

  .

  .

and another to print:

 JOHN

 JOHN

 JOHN

  .

  .

  .

and

 HI MOM

 HOW ARE YOU?

 LOVE, JOHN

 HI MOM

 HOW ARE YOU?

 LOVE, JOHN

  .

  .

  .

and then anything they wish.

Our experience, based on a very large number of first sessions with people of all ages, is that giving models to copy is much better at this stage than verbal explanation.

*We do not share the prevalent general prejudice against rules and verbal statements in teaching mathematics. But we find we can develop language best by first establishing models or experiences to be used in defining terms.*

We also feel firmly that introducing the infinite-looping recursive program is best done at the very beginning. This creates a first image of the program as a “living” active agent. It avoids initial confusions between definition and “run” by making their effects qualitatively different – running goes on by itself indefinitely. Later we will face the problem of making a run stop on its own. Loop-free programs can come later still and appear as a curious case rather than as the natural starting point.

As a matter of fact, children will soon discover the loop-free program by carelessly leaving out (or wrongly stating) the recursion line. Perhaps, this is the best reason for not teaching it!

It is worth a great effort to establish a good initial image of programming and good habits of work. Many habits develop much faster than they can be changed. Some anecdotes will illustrate the point, perhaps, in a sense slightly deeper than its obvious one.

Example 1

Child X is bored at the teletype during a very early session. Not understanding enough to do anything constructive he toys idly with the keys. He accidentally finds that the repeat key can bring the machine to life. This has a positive effect – boredom has gone, the teletype has turned into a fascinating living creature. But it has a negative effect too: the next week is dominated by non-constructive games with the repeat key. By contrast child Y made a discovery just a little later in his development and was able to integrate the spectacular effect of repetition into the fun of programming. For example, a very simple procedure to count the number of repetitions led to games depending on guessing the number, aiming at particular numbers, and so on.

The first child lost more than the time spent on his game: during this period programming appeared to him not as fun but as a chore that interfered with his fun.

Example 2

A very bad habit that transfers too easily from the math class is to throw away incorrect programs – rather than understanding and debugging them. By starting children on programming with a small subset of Logo and with no concept of composing operations we increase the intelligibility of the programs they write, and so, increase our chance of being believed when we insist that the “bugged” program can be understood.

2.3 Naming

2.3.1 Quote and Slash

There is some justification for presenting the material in this section before talking to the children about Logo or introducing them to the terminals. The material is enjoyable in itself and if covered in advance will accelerate the transition to interesting programs.

We begin by playing a game with the class along these lines:

Teacher: Will someone please say your name.

A: John

Teacher: I said say your name

A: I did say my name

Teacher: Did you really say my name etc., etc.

Eventually someone will get the idea and reply:

B: Your name

Teacher: How do you spell it?

A: J-O-H-N

B: I-T

Teacher: How do you spell your name?

B: Y-O-U-R-N-A-M-E

Children are familiar with this kind of game and enter easily into the spirit. The point of the trick is that we can introduce quotation as a helpful device, a way out of a real difficulty, rather than as a formalistic restrictive convention.

In other words, instead of saying that quotes are compulsory by edict in

 He said, “Hi”

We show that they are valuable when we actually need to distinguish between

He said his name

and

 He said, “His name”

The example illustrates the fact which can be presented in a very informal, relaxed way that sometimes we talk about a word itself and sometimes we talk about the ***thing***of the ***word***. Most often everyone knows which we mean. But people sometimes get confused and computers do very easily. To make the distinction we introduce the notation of “quote” and “slash” used at this stage informally in English. To facilitate speaking aloud we read “A” as QUOTE A and /A/ as SLASH A.[[1]](#footnote-1) Now we develop dialogs like:

A: John say “your name”

J: John

A: No say “your name”

J: Your name

A: Right. Say /your name/

J: John

A: Say “your age”

J: your age

A: Say /your age/

J: 10

and so on.

2.3.2 Names of Things and Things of Names

Another useful conversation game in class is directed at heightening awareness of the process of naming.

Teacher: Tell me some things you can do with bricks.

Class: Build houses, throw them, break them …

Teacher: Tell me some things we can do with numbers.

Class: Add, count, …

Teacher: What can be done with bricks or numbers?

With some persistence and hinting, but never easily, we eventually get the class to say “talk about them” and, even more reluctantly, “give them names”.

The exercise is useful for later reference. For reasons that might be worth a thorough psycho-social study, children are shy about naming things – especially unusual things like bricks, words, numbers. The brick story can be used to keep children sufficiently aware of this reluctance – this bug in them, some might say – for them to work at removing it or at least be ready to recognize it when it operates.

To fight the shyness at this stage we introduce some terms for talking about the naming situation. Can we call the number 3 “Joe”? If so “Joe” is the name (or a name) of the number. It is strange that English has no common word to fill into:

3 is the \_\_\_\_\_\_\_\_\_\_\_ of “Joe”.

We shall use the word “thing”.

“Joe is the name of 3.

3 is the thing of “Joe”.

As an exercise we make each member of the class give their name a name. Why should words not have names?

Since the following dialog is meant to be spoken we have not written quotes:

Teacher: Joe, what is your name

Joe: Joe

Teacher: What is your name’s name

Joe: Bill

Teacher: What is the thing of Bill

Joe: Joe

Teacher: Slash Joe or Quote Joe

Joe: Quote Joe

Teacher: Are you Slash Joe or Quote Joe

Joe: Slash Joe

Teacher: What is the thing of the thing of Bill

Joe: I am

2.3.3. Naming is for Talking about Things

It pays to set up in advance of its need in programming a model for the important need to name things. A good example for the class is the following class exercise about building airplanes.

We find a volunteer who knows how to make the dart kind of model airplane.



We ask him to tell us how to make one. The point to bring out is that confusion will arise if he does not either resort to pointing or naming the new corners and edges, as they come into being. We invent some excuse for not pointing (e.g., we are telling someone by telephone). A conversation like the following might ensue:

V: Fold the paper in half, and unfold it.

T: Like this?



V: Yes. Now fold a corner into the middle.

T: O.K.



V: Now do the same with the corner on the other side.

T: O.K.



V: No, not that opposite side. The other.

T: Which? This one? (bottom right)

V: No, that one.

T: I see, like this.



V: Yes. Next fold the new corner.

T: So …

V: No, no. The new one on the edge.

T: What do you mean?

The point is made by creating confusion and then salvaging the situation by giving everything names so that instructions are now like:

Fold the corner A onto the center line L. This makes two new corners, one on L and one on the edge R. Call these D and E.

Now fold E onto the center line L. Etc., etc.

Another example is a set of instructions working a complicated machine: turn dial A until light C goes on, push switch D, etc.

2.4. Program Forms

An important component of the initial theory of teaching Logo was the scope it provides for approaching problems with well formulated strategies. One of our early favorite examples was the project of writing a program to translate English into Pig Latin. Many heuristic precepts can be applied to the problem in a simple and transparent fashion. For example, it is natural to plan the solution by subdividing the task into components. Thus one can think about how to obtain a sentence-translating procedure, say PIG, from a word-translating procedure, say 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. What is special about the programming problem (compared with, say an algebra word problem or a real-life problem) is that the plan is particularly simple and the subdivision into component parts is particularly clear. The pedagogical purpose is for the child to obtain from these simple cases good habits of planning and clear paradigms for his strategies. The hope is that they will later be able to apply them in more intricate circumstances.

As we have worked with Logo we have greatly extended this way of thinking, particularly in the direction of very simple programs. So much so that it now appears strange that anything *as untidy* as Pig Latin could ever have presented itself as an attractive example – though the point that it makes remains acutely important in our thinking. We have now come to recognize a number of very simple paradigm forms for programs. In teaching each new form, particular examples are presented to the children to be used as models for other programs of increasing complexity. As we do so, we also introduce the children to a descriptive language appropriate to the form and develop their understanding of where it can be used, of the typical bugs associated with it, and how it relates to informally familiar procedures. To show what we mean by this we shall develop in some detail these various aspects of a particular program form. A similar discussion of the other forms will be found elsewhere. Perhaps we should mention again that our discussion includes more material than can reasonably be given to most children at one time.

2.5. A Particular Program form: Straight Recursion

We begin with a very simple program that serves as a model for the most elementary version of this program form. It is made up of three lines:

TO HI TITLE line

PRINT “HI” ACTION line

HI RECURSION line

END

Notice the terminology:  ACTION and RECURSION. It will be used repeatedly to give structure to the process of writing and talking about writing a procedure. Interspersed with examples, we develop numerous analogies with recursion, e.g., many cyclic processes like automobile engines and sewing machines are like this model of recursion.

A slightly more complex version is obtained by allowing an input:

 TO HI /NAME/ TITLE line It tells us that the procedure has one

 Input.

 which will be called “NAME” in the

 description of the procedure.

PRINT /NAME/ ACTION line

HI /NAME/ RECURSION line

END

Example of Use

HI “JOHN”

JOHN

JOHN

JOHN

 .

 .

 .

Recursion with MODIFICATION

TO COUNTUP /NUMBER/ TITLE line

PRINT /NUMBER/ ACTION line

MAKE MODIFICATION line

 NAME:  “NEWNUMBER”

 THING: SUM /NUMBER/ AND 1

COUNTUP /NEWNUMBER/ RECURSION line

END

The introduction of a STOP TEST is easy at this point, yielding the most general version of the Straight Recursion program form:

Recursion with STOP TEST

TO COUNTUP /NUMBER/ /TOP/ TITLE line This procedure has two inputs

TEST IS /NUMBER/ /TOP/ TEST line

IF TRUE STOP

MAKE MODIFICATION line

 NAME:  “NEW”

 THING: SUM /NUMBER/ AND 1

COUNTUP /NEW/ /TOP/ RECURSION line

END

This program form is far from universal – the reader should peek at the next section[[2]](#footnote-2) where some very different forms are listed. But recursion is extremely powerful and we try to give children enough examples to sense this power. In addition to developing this sense of where it can be used, teaching a program form includes achieving a number of objectives, which are illustrated by the following list. All the examples in the list are couched in terms of the simple recursive program form. Similar formulations of objectives for other program forms will be found in later  sections (not available January 1970).

* Fluency in the use of a descriptive language illustrated by the names TEST, ACTION, MODIFICATION, etc. This ensures articulate communication between the children themselves and their teacher. It also helps the child formulate and perceive more precisely what they are about – particularly when in trouble.
* Fluency in changing a program into equivalent forms like:

TO COUNTUP /N/ /T/

TEST IS  /N/ /T/

IF TRUE STOP

COUNTUP (SUM /N/ AND 1) AND /T/

END

* Familiarity with common bugs. For example, the class can collect and name the ones it has seen often or finds tricky. Among the commonest are:

The SLIP-BY Bug

E.g., a procedure to count by 2s from 21 to 50 might slip by the STOP TEST with …, 47, 49, 51, … The remedy is to use an *inequality* instead of an equality for the STOP TEST; stopping when the number exceeds some limit. (Thereby, incidentally, the children meet inequalities as a life-saving device rather than as a formal horror.)

The Muddled Reduction Bug

You can think, loosely, of many recursive programs as successively reducing some LOGOTHING until there is nothing left. But the loose concept of reducing covers several different processes such as decreasing numbers until zero is reached, et alia. The Muddled Reduction Bug is the result of muddling these reductive processes.

The following STOP TEST and MODIFICATION lines are wrongly paired:

STOP TEST: TEST IS /IT/ /EMPTY/

MODIFICATION: IF TRUE MAKE “IT” DIFFERENCE /IT/ 1

STOP TEST: TEST IS /IT/ “0”

MODIFICATION: IF TRUE MAKE “IT” BUTFIRST /IT/

The Fence-Post Bug

This very common bug gets its name from the confusing fact that a fence with ten posts has nine spaces. The bug could also be called the Inclusive-Exclusive Bug since it is related to frequent ambiguity about whether to count in the end objects (e.g., how many days from the 4th to the 9th of March?).

Example

TO GREET /TIMES/

PRINT “HI”

TEST IS /TIMES/ 0

IF TRUE STOP

GREET SUM /TIMES/ AND -1

END

The command, GREET 4, will produce five HI’s.

The Inconsistent Name Bug

This extremely common bug can be introduced into a good program when editing. One might want to change a certain name but inadvertently leave one occurrence in its old form. Or a spelling mistake might produce the same effect. It is related to the important heuristic principle that anything can be called anything but always the same anything.

Recursion, Recursion Everywhere: Relating the Program Form to Familiar Situations

The relation of the program form to informally familiar processes is a less precise topic, but an integral part of the teaching. We think it is very important for the teacher to be able to see the RECURSION aspect of a large variety of processes and, thus, to seize the opportunity of making a point to a child in terms of examples he knows and cares about. Here are some examples.

Mechanical Recursion

We have known a boy who obtained his first real insight into recursion by seeing the recursion in the operation of an internal combustion engine. And his insight was not merely the superficial fact that the I.C.E. works cyclically.. What caught his mind was the fact that each cycle starts the next cycle – so “when it goes it keeps itself going.” A bouncing ball is another simple, relevant phenomenon, with the added feature that the cycles are modified by loss of energy.

Heuristic Analogs

The structure of the recursive program can be elegantly related to a number of heuristic principles of problem strategy. The point is illustrated by the following sketchy outline of a development of a program in class. The problem is to write a procedure called MEMBER with 2 inputs, a number and a sentence. The intended operation of MEMBER is shown by:

MEMBER 3 “A B C D” = “C”

MEMBER 1 “CAT DOG X” = “CAT”

So we begin by writing

TO MEMBER /NUM/ /LIST/

How do we go on? High on our list of good heuristics is

HEURISTIC:  Look for easy special cases.

The obvious one is /NUM/ = 1. In this case the problem is trivial since we can use the Logo operation FIRST to get the answer. So we have the first two lines of our procedure:

TEST IS /NUM/ 1

IF TRUE OUTPUT FIRST /LIST/

We note that the STOP TEST part of the recursion form often corresponds to the “easiest case” of the problem. Next we apply the

HEURISTIC:  Reduce the hard cases to the easy ones.

For MEMBER this is done by observing:

The problem

MEMBER 3 “A B C D”

can be reduced to the problem

MEMBER 2 “B C D”

and this can be reduced to the problem

MEMBER 1 “C D”

which is our easy case. So we solve the general problem by successive reductions. This is done by the “double MODIFICATION” lines:

MAKE

 NAME: “NEWNUM”

 THING: DIFFERENCE /NUM/ 1

MAKE

 NAME: “NEWLIST”

 THING: BUTFIRST OF /LIST/

OUTPUT MEMBER /NEWNUM/ /NEWLIST/[[3]](#footnote-3)

Pictures

Possession of pictorial representations of the operation of a program form is an important part of fluency – especially in the early stages. Manuals on programming usually present the “flow chart” as the universal diagrammatic aid. We find (a priori, and empirically) that this is too general and rigid. So we have developed a number of picture forms. The one most appropriate to the recursion program-form is our “little men” metaphor – which has great merit in other respects also and will be discussed later. At this point we illustrate it by the following “comic strip” illustrating how the computer reacts to

PRINT MEMBER 2 “A B C”

The key point is that it gives us a model for the push-down mechanism that is vivid, simple and entirely adequate for these simple programs.

 MEMBER 2 “A B C” MEMBER 1 “B C”

PRINT Man MEMBER Man NEXT MEMBER MAN

          

 your answer is your answer is

              “B”           “B”

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| The PRINT MAN sees that his input is MEMBER 2 “A B C”So he calls up a friend gives him the procedure MEMBER and the appropriate inputs. He says: get the job done; don’t bother me until you are through.[Later]The reply to my request is “B”. Now I can PRINT it. | –><–  | The MEMBER MAN says:  /NUM/ is 2. If it were 1, I’d reply the answer is FIRST “A B C”. But it is not 1, so I see I have to call up another guy to get MEMBER 1 “B C”[Later]The reply to my request is “B”. My instructions say I must OUTPUT the answer. That means I must pass it back. | –––>   |––   |   |     |<–  | This man has it easy. He replies “B”. Since the program says OUTPUT FIRST “B C” |

THAT’S ALL FOLKS!

Afterword About This Paper

Details of the Nitty Gritty Sort

We never wrote more chapters about this work and what you just read stops abruptly because a lot got lost. In the last part of chapter 2 were more names for various programming forms such as Set-Up forms, Super-Sub-Procedure forms, Try-Again forms and a non-recursive procedure form to find list membership that Seymour called the Test-Test-Test form.

We did write another paper involving this group of students, NIM: A Game Playing Program, January 1970[[4]](#footnote-4). It detailed the one-pile NIM game that these children worked on.

What we had planned was to continue writing about activities and reflections on our work with the 7th graders at Muzzey Junior High in 1968-69. Their content included Pig Latin programs, sentence generators, math quiz programs, games like one-pile Nim, concrete poetry and haiku, stories and so on.

Things for Reflection

We shared our view that different computer environments can give rise to different computer cultures. The idea that a culture forms around computing was novel then and unusual even now. In our computer culture expressivity, creativity, and curiosity were shared elements. We talked about projects, procedures, processes and debugging. We talked about computational thinking, about being a computer, giving things names, Polya-esque dividing problems into sub-problems, being willing to try things out, re-thinking and debugging. Having procedures do unexpected or unwanted things was not shameful. It was considered part of the process and through debugging students gained a deeper understanding of themselves, their bugs and their features. Each student seemed to find their own short cuts, aids, or heuristics. And then it was like there was an underground communications system as students shared their findings with their classmates.[[5]](#footnote-5)

.

Here are some ideas we emphasized.

* Give things names so that we can talk about them

This included names we give to data like numbers, words, and sentences as well as names of procedures. We also developed a meta-language, a way of talking about different parts of a procedure and different parts of the whole project.

* A good heuristic: break things into procedures and sub-procedures

We emphasized different roles procedures play. And used a little man model to illustrate this. A procedure had a certain job to do and could call on other procedures to help. Most subprocedures only had to do a task and didn’t have to know what the superprocedure would do.

* Debugging as a major programming activity

Perhaps the major activity in programming or coding is debugging. Debugging is the process of removing bugs or declaring what you thought was a bug to be a feature instead. Our emphasis was that most bugs were interesting objects. Sometimes they related to bugs in ourselves, sometimes they were typos, and sometimes they were due to cloudy thinking. One thing for sure, you would encounter bugs.

Playing computer, being a computer, stepping through a procedure were all part of debugging strategies.

Logo Development

This work set the stage for what was to become a powerful, flexible and useable programming language for children and highlighted the ease with which learners quickly became expert in using this tool. This work is also an important early example of involving children in the design of new technology and how the technology can benefit from the design expertise and feedback from children. Each time Seymour and I worked with children Logo was radically redesigned incorporating their feedback

Our next collaborative writing project was in 1971 after we worked for a year with 5th graders. By the start of the 1970-71 school year I had joined Seymour at MIT (and so there were two Logo efforts as the BBN folks continued work with Logo). Thanks to the talent and cooperation of Marvin Minsky and other AI folks we had a floor turtle, a display turtle, a music box and new ideas for expanding procedural thinking and debugging as we taught children to walk on stilts, juggle, and more. This work spawned lots of ideas that became part of the Twenty Things to Do with a Computer.

Our work foreshadows what has become the participatory design movement, the interaction design and children field, and the computer science advances in programming languages for novices.

-Cynthia Solomon, 2017

1. In this version of Logo the name of a thing had to be one word. To get at the thing the word had to be enclosed in slashes. Later versions of Logo used a prefix mark we called “dots” : . [↑](#footnote-ref-1)
2. Unfortunately there is no next section. [↑](#footnote-ref-2)
3. In Microworlds the procedure looks like the following.

to member :num :list

if :num = 1 [output first :list]

make “newnum :num – 1

make “newlist butfirst :list

output member :newnum :newlist

end [↑](#footnote-ref-3)
4. (MIT AIM 254) https://dspace.mit.edu/handle/1721.1/6199 [↑](#footnote-ref-4)
5. See my papers at https://logocomputercultures.wikispaces.com/home [↑](#footnote-ref-5)