# And they were thinking? Basic, Logo, Personality and Pedagogy

John S. Murnane



# And they were thinking? Basic, Logo, Personality and Pedagogy

John S. Murnane

The ICT in Education and Research Group
The University of Melbourne, Australia
jmurnane@unimelb.edu.au

**Abstract.** This paper is concerned with some limited aspects of the history of two programming languages purpose-designed for students learning to program digital computers: Basic and Logo. The focus is the very different educational aims and philosophies of the originators of these languages. They are compared and their early use in schools sketched. While the reasons for teaching students to write programs were initially based on experience in programming digital computers for non-educational use, despite extensive research and publications, it would seem that the teacher of today is not in a much better position to justify teaching programming than the original pioneers.

**Keywords:** Computer education; introductory programming languages; history of computing.

## 1 Introduction

> *The introduction of yet another language clearly deserves critical examination.* Feurzeig, Papert, Bollm, Grant, and Solomon [1 p12]

This paper is concerned with the history of programming languages purpose-designed for students learning to program digital computers. A proper treatment of this topic would run to a small library and deal with the ideas and intent behind the form of the language, research on its success in educational use, examples of classroom use and modifications made as a result of experience, so stringent selection was necessary. My main interest is in the intent of the creators the languages, so I began with the idea of examining the educational concepts behind the development of several of them, but in the end found space for only two, Basic and Logo, and then in a very constrained form.

The creation of a programming language of any sort is a complex business and the province of a special elite in the world of programming. Yet the difficulty of the creation of a language for fields such as business or mathematics for which there is an existing set of well tried models, pales into insignificance compared to the task of creating one for educational purposes: a space where a choice must first be made between various pedagogies, all with their own built-in advantages and disadvantages,

advocates and detractors, before even the form of the language is decided on. Nor will the educational aim be simple. Is it to be a language to introduce the forms and disciplines of programming itself, or is it to facilitate a more general development in problem solving and analytical and logical thinking? Are we teaching computing and its applications, or are we using programming for some wider educational purpose? Ham [2 p34] believes that, when it comes to the use of IT in education, "even within the educational policy and research communities, people do not necessarily agree on the questions which are worth asking."

I wish to examine the published thoughts of some of the earliest pioneers who dared to swim in this very complex sea.

## 2 The Promise

> *We have learned how to work with the computer in solving a problem, rather than submitting a problem for machine solution.* Kemeney and Kurtz [3 p22]

Even in the new centaury it is easy to find material critical of the use of computers in schools, from Cuban's 2001 [4] characterisation as being "oversold and underused," to Cox (2010) [5 p16] who found "the actual integrated use of IT by the teachers is much lower than might have been expected from so many sustained national and international programs." But even allowing that not all teachers use IT brilliantly, it is hard understand Munro's [6 p47] criticism that "microcomputers were introduced into educational institutions with no prior research and with no educational rationale for their use." True, they were. When something is new its introduction cannot be based on research, and the educational rationale must be the belief of the teacher in the promise the new idea holds. It was the promise the new world of programming held for all sorts of educational and cognitive advances that attracted the pioneers of educational programming languages. "Programming," declared Ershov in 1981 [7 p1] is "the second literacy."

Looking back at their a-priori positions on the benefits that writing programs could bring to their students, one finds a remarkable unity of spirit. Cynically, it could be argued that in the 1960's, apart from some rather inflexible Computer Assisted Instruction (CAI), and some (mostly non-interactive) simulations, writing a program was about the only educational thing you could do with one. But the pioneers, and particularly those with a hand in writing specialised educational languages, such as Kemeny, Kurtz, Feurzeig and Papert, were all convinced that great educational advantages would come from programming a digital computer, although often for different educational and cognitive reasons. Weyer and Cannara [8 p3] put it this way: "If, by a free interpretation of Church's thesis, any ideas which can be formalised may be studied concretely via a computer program, then, by learning programming in full generality, students can learn to construct laboratories to study any ideas they wish to think about."

Basic and Logo were both written in the 1960's, were specifically designed for educational purposes, and are still in widespread use. A fascinating speculation, now difficult to resolve from material published of the time, is how much the designers

were influenced by their educational philosophy, and how much by the educational environment in which they happened to be, and the available tools. Feurzeig and Papert, operating in the Artificial Intelligence Laboratory at Stanford, the home of Lisp, not only had an example of a language congruent with their educational ideas, but one in which their new language could be written. Kemeny and Kurtz had two very small computers from which they hoped to assemble a useful system, neither of which had so much as an operating system.

The decade that produced the first educational computer languages is now 50 years in the past. With the singular exception of anything written by Seymour Papert, looking back through the papers and reports leaves a distinct impression of teachers striving: striving towards goals imperfectly grasped, using computing equipment barely up to the task and hampered by primitive translators, operating systems and input/output devices. (Papert knew what he was doing from the outset.) Yet the *overall* feeling is of high optimism, more positive than one finds across the literature in 2010. The pioneers *knew* that programming a computer had educational benefits, and were going to set about proving it to the world.

## 3 Basic

> *Our goal was to provide our user community with friendly access to the computer*. Kemeny and Kurtz [9 p534]

Seymour Papert always complained that besides being a poor language, Basic had been left for the academic community "to pick up, like cast-off clothing" [10]. There is some weight to this, for Kemeny and Kurtz, Basic's authors, do concentrate in their publications on Basic as Computer Science and do not say much on its pedagogical or curricular aspects. Their Final Report to the Course Content Improvement Program of the National Science Foundation, who financed it, is titled "The Dartmouth Time Sharing System" [3] rather than something suggesting *educational* advance. It gives the reasons for teaching programming to college and secondary students as:

- The need for more people to learn to program because of the key roles computers play in "'business, industry, government and all forms of research."
- To change the attitude "of the typically intelligent person towards computers," which they characterised as "a mixture of fear and superstitious awe." (One wonders how much has changed!)
- To put "the computer at the fingertips of the Faculty." [3 p1]

There is little here to explain just why writing a program might be educationally advantageous. They simply state "the hard question was not 'whether' but 'how'" [9 p518]. They do give many examples of programs written by students across a range of disciplines, but leave it largely to the reader to decide on the cognitive benefits that accrue. Significantly, they did find that in a programming environment, students were more likely to share ideas [3 p16]. They also characterise computers as "a magnificent means of recreation," (p. 8) something which in 2010 threatens to overshadow their significance for learning.

Kemeny and Kurtz strongly distinguish between using computers for instruction and having students write their own programs. While they saw the possibilities of CAI, and even thought "they ought to do more," they saw much more potential when "the student is the teacher and the machine learns" [3 p11], noting that "by being able to program certain processes, the student necessarily shows a through understanding of the process" (p. 27). Both these themes run through computer education to the present day.

Understandably, Kemeny and Kurtz devote most of their various publications to Basic itself and the ground-breaking time-sharing system shared between two computers that went with it. Written by sophomore Michael Busch and junior John McGeechie [3 p5], this system was the element which made the project economically possible and educationally viable, an example of what can be achieved by enthusiasts too young to know that what they were doing wasn't supposed to be possible.

Basic was an acronym for 'Beginners All-purpose Symbolic Instruction Code.' Dartmouth, where its originators taught, attracted students who were "not generally interested in the Sciences," so it was designed for those studying the liberal arts [9 p518 & 522] "as an extremely simple language that can be quickly mastered by a novice" [3 p3]. They considered that Fortran had "many disadvantages for the novice and occasional user," meaning largely there was too much fussy detail to remember, and "decided that we would improve it" (p3). Considering other languages available or planned, they thought the Algol compound statement introduced too many complexities for beginners [9 p538] and wanted something much smaller and more general then Cobol. As long as the user is content with real numbers (which Kemeny and Kurtz considered removed the need for typed variables), and happy with two-character variable names (forced by the exigencies of the computers), Basic can be said to express mathematical ideas generally as well as Fortran. Indeed, it has the added power of matrix operations, and with the extension to string handling it came much closer to being truly 'general purpose' as well. It has also shown remarkable longevity, widespread use in commercial applications and a capacity to accept extensions gracefully.

Perhaps the best expression of the benefits of teaching students to program comes from Thomas Dwyer [11-14]. Dwyer extended Dartmouth Basic to make it a better language to teach *with*. This may sound contradictory and clash with the educational ideas of Basic's creators, but actually it was designed to reinforce them.

Kemeny and Kurtz discuss the advantages of having a student teach the computer. Papert's *Mindstorms* [15] can be seen as an extended plea for learner control. Dwyer considered that students learn best when they are teaching other *students*, so he wanted his students to write programs for others to learn *from*. Hence his characterisation of educational programming as having two modes: Dual and Solo. Dual mode consisted of using a computer to learn with something programmed by someone else—what today would be covered by courseware and information retrieval and processing. Solo mode was, initially, writing programs for your own use, but then going on to write programs to teach different parts of the curriculum to others. Here he was picking up Kemeny and Kurtz's ideas on needing to understand a process if you are to write a program for it. Dwyer [12 p220] explained this as:

> *a learning situation which develops advanced cognitive and motor skills for students of quite varied backgrounds, and which also involves many affective*

> *elements, but which relies heavily on technology for achieving these ends. While it is clear that dual instruction is essential (one does not recommend that a student immediately go out in an airplane and 'do his own thing') it is equally clear that the student will optimise his benefits from the dual mode if he knows he is preparing for a solo flight. He knows, in fact, that he can eventually exert more influence on his learning than his instructor … computer technology in education should invite similar control at all levels. It should, in particular, invite the student to 'go behind the scenes' (possibly acting in concert with teachers) at any time they elect. There should be no secrets, on one-upmanship of the adult world over the student world.*

To go behind the scenes and do your own thing you have to be able to change the system. In 1971 it meant you had to be able to program. These days with the advent of Web 2.0 and all sorts of multi-media vehicles it's a bit different, but the principle still holds, more strongly if anything. diSessa [16 p164] advocates a similar hierarchy. Papert considered the "proper use …of drill-and-practice programs" was something for other students to write [17 p4–1]. It's also interesting that Dwyer emphasised, as his first principle, the essential "social character of human learning," though I doubt that in the USA in the 1960's he'd ever heard of Vygotsky.

Despite its attractions, Dartmouth Basic has always been heavily criticized for its simplicity and the lack of structure and formal rigor of programs written in it. The first definition of Algol, with its ground-breaking ideas and structure was published in 1958, and an Algol compiler was available at Dartmouth by 1967 [3 p8]. It would seem that Kemeny and Kurtz's desire for simplicity overruled thoughts of enhanced Scientific (or Mathematical) precision.

## 4 Logo

In original conception, Logo was conceived as a form of Lisp suitable for beginners to write programs in. The 'Lo' in the title suggested 'Logic,' and the earliest versions of the Logo system were written in Lisp. It is curious then that early papers [1 p1, 18 p1] make it clear that it was "expressly designed" for the teaching of Mathematics. At that time there was no Turtle Geometry, and indeed no arithmetic functionality beyond addition and subtraction. Brown and Rubinstein [19 p3] flatly describe it as "non-numeric."

Early Logo was very simple. Like Basic, it came with built-in editing and file manipulation commands. If these are ignored, the 1971 version consisted of just 26 'operations,' five of which accessed the calendar and clock, and 15 'commands.' Most operations were concerned with program logic or list manipulation. An essential part of the design was to produce a language of such simplicity that it forced users to write their own library of commonly used routines, such as multiplication and division. From such a library, complex programs could be built. "Ideally, by the end of the course, each student would have created his own extended version of Logo" [19 p10]. If you exclude the rich set of mathematical functions, contemporary Basic was even smaller, but for a different reason: to make the language as easy as possible to learn. (And remember Basic was designed for the 'non-mathematical.')

There is no mention of the degree of difficulty inherent in learning to program generally, and certainly not in learning Logo, in Feurzeig's papers. The entire emphasis in *The Final Report* [1] is on the difficulty of learning *Mathematics*, and how Logo was developed to make that easier. The programming language followed as a result of a specific educational need. The designers of Logo intended it not only as a vehicle to express Mathematical ideas and make Mathematical concepts concrete, they saw it as a meta-language in which to express Mathematical thought [1 p5]. They wanted a "standard, teachable terminology to discuss the *heuristic aspects* of mathematical activity concerned with the art of solving problems" (p. 5, their emphasis). Here is the origin of Papert's often expressed need to teach 'thinking about thinking' [17 p2] and the decision to write a computer language whose primitives and predicates inherently contained and *expressed* the mechanisms of logic and Mathematics. "Do we give children the instruction 'think!' without even telling them how to think?" (p4, Papert's emphasis). The mathematical purpose expressed by Feurzeig is actually at odds with Papert who is at pains to stress the general problem solving capability of the language [17, 20]. Lisp's origins in Artificial Intelligence were supposed to support this [21 p14, 22 p16], but no author I have read ever explained how it was to happen.

## 5 Basic and Logo

Feurzeig's Logo group began with *education* and worked *back* to the form of their language. To create Basic Kemeny and Kurtz began with Computer Science and found educational uses for it. As someone who has spent forty years shouting at his education students to always begin with education and bring in computers if they could be useful I find it painful that Basic was an almost instant educational success and Logo wasn't.

By 1967 Basic was in use in eighteen secondary schools, eighteen colleges and universities, government agencies and "some local business concerns" [23 p23, 24 p2]. School use in particular was only limited by the number of available telephone lines. The reports are full of interesting and advanced programs written by students at all levels and the enthusiasm of the authors is obvious. (That said, Putnam, Sleeman, Baxter and Kuspa [25 p22] state "Errors were found with virtually every construct in all tests and interviews. … students with a semester or more of experience with Basic had a very fuzzy knowledge of how various constructs operate.")

By contrast, many of the programs in the *Final Report on the Logo Project* [1], seem forced, elementary and repetitive. Many from the primary level, ages 7 to 9, are examples of programs to reverse the letters in a word, print a set of consecutive numbers or simply print strings. The first lessons did not involve writing code at all. This did not happen until lesson Seven (p. 67). In the secondary curriculum, many essential elementary functions such as divide and multiply were written by the teachers and given to the students to try and understand (p. 215), the inference being that they could not be expected to write these routines themselves. Johnson [26 p201] found "The position that the programming environments themselves, e.g., Logo

microworlds, would become the school mathematics curriculum has clearly failed to gain the support of the educational system." (See also Mayer [27].)

None of this suggests a language easily taken up by beginners and used for their own purposes. Part of the reason has to be the use, initially, of recursion for all loops, definite or indefinite. Recursion is, as Papert has said repeatedly, a powerful problem solving tool [15, 17, 20, 28] and indeed it is. But then, so is calculus. Papert in particular has always insisted that Logo is designed to encourage experimentation, with students writing and testing their own creations. Papert worked with Jean Piaget for many years and passionately believes in the idea of 'learning by doing,' something he later extended to what might be termed 'learning by *making things*.'

Given this emphasis, it is difficult to understand the reliance on recursion at the expense of a general iterative statement. Not once in all my reading have I come across an assertion that students can be expected to discover a recursive solution to a problem on their own. All I can find are examples provided *to* students to explain, understand, and adapt. In his seminal book, *Mindstorms*, [15 p71] Papert states that "recursion stands out as the one idea that is particularly able to evoke an excited response." That might be so, but he devotes less than two pages to it, mentioning it once more in the Appendix in the context of 'circular logic' (p109). Brown and Rubinstein suggest that with suitable prior experience, students can write their own recursive routine to traverse a tree, but they give no clue to their success rate. They did find that "if a student couldn't figure out how to write a function, we could not slowly lead him down the path to discovery" [19: 43]. Once acquainted with WHILE—DO in Basic or Pascal, or even the primitive Dartmouth-Basic GOTO, students have no trouble in writing their own indefinite loops. (Murnane [29, 30]. See also Vitale [31 p272 & 272], and Murnane and Warner [32] for examples of experiments where children could have, but failed to use, recursion.)

A further reason which can be advanced is Logo's Lisp inheritance, essentially lambda calculus, whereas Basic was deliberately designed to be "as close to ordinary English combined with elementary algebra as possible" [23 p4]. Logo statements do not always accord with English, although in its earlier versions it approached it more closely than in later ones. Indeed, anyone coming to Logo after 1970 would be hard-put to recognise the original language. For example, Multiply [1 p215] is defined as:

```
TO MULTIPLY /X/ AND /Y/
10 IS /Y/ "0"
20 IF YES RETURN "0"
30 RETURN SUM OF /X/ AND MULTIPLY OF /X/ AND (DIFF OF
/Y/ AND 1)
END
```

Even allowing for the difficulty of conceptualising the recursion, it is not English, and in its early iterations, Logo struggled to make progress. The cure for many of these problems was provided by Seymour Papert.

Logo is often associated specifically with Papert, and particularly with his Turtle Graphics. He joined the project in January 1969 as a consultant [1 p1, 33] and his invention of the Turtle and its commands transformed the language.

A Turtle is a small robot which, when connected to a computer, can move and turn on the floor. At a stroke this eliminated the gap between entering a program and

observing its outcome, since the Turtle could execute a command as soon as it was entered. It also solved the problem of the student understanding what the command did: they could walk around behind the Turtle following its movements with their own. While they might need to be taught the meaning of "TEST IS COUNT /SENTENCE/ 1 [18 p45] they could easily appreciate what FORWARD 100 meant because it accorded to their own body actions and their natural language. Turtle Geometry provides an immediate and meaningful environment for the beginner, relating body movement to the effect of a Logo statement. Papert [28 p24] describes this as the "idea of 'body-syntonic' representations of knowledge." (For a discussion of designing computer languages to correspond to natural language see Murnane [34]).

Along with Turtle commands came definite iteration: REPEAT :N, relieving the programmer of the need to write all loops recursively. A recursive loop can only be executed by writing a procedure and then executing it. In keeping with the idea of observing actions as the commands were entered, you could now type REPEAT 4 [FORWARD 100 RIGHT 90] and *watch* a square being drawn. Note also the close correspondence to English syntax.

Once the Turtle migrated from the floor to the screen, Logo became accessible and viable in any classroom.

Papert's other outstanding attribute was his ability as a teacher, educational theorist and writer. Probably no one has matched his output, or perhaps, his influence, on educational programming. His argument is that students "do not understand the kind of thing a mathematical structure is: they do not see the point of the whole enterprise" [15 p23]. By using Logo and the Turtle, these concepts can be made visible and concrete. Nevertheless, his pronouncements on the promise of Logo to help teach mathematical concepts [35 p3] such as angle, length, variables and differential geometry, as well as "epistemological primitives, such as the notion of a mathematical system itself" (p. 23) have not been supported by hard research. Ross and Howe [36 p147] found that "the research of the last decade into 'mathematics through programming' have been more encouraging than discouraging, but only mildly so."

Rather sadly, the weight of subsequent research suggests that programming in Logo, by itself, does not teach Mathematics. Students, unless specifically taught about these points, keep Logo and Mathematics entirely separate in their minds, and few teachers, perhaps persuaded by Papert, seem to do this, or do so with much success. An experiment with Year 8 students, all of whom had used Logo (in the form of LogoWriter), showed almost no traces of Logo when asked to perform tasks in which it could be expected to appear if Papert's theories are correct [32]. There were almost no signs of Logo functioning as a meta-language. Even Abelson, Barnberger, Goldstein and Papert [22 p10] rather sadly remark that "Logo did not succeed in displacing Basic as the almost universal computer language for schools."

## 6 Beyond the 60s

> *At a minimum … the teacher must be absolutely fluent in at thinking in Logo.* Brown and Rubinstein [19: 4]
>
> *Anybody who is at all serious about writing programs must avoid the temptation of thinking in a programming language.* Juliff [37: 38]

These two quotations really summarise the different educational and practical orientations of Logo and Basic. Logo was intended to be used in the closed environment of education as a *language to think with*, and Basic was intended to introduce students to the world of programming. Kurtz [9 p3] insisted that "our one mistake was to include the word 'Beginners' in the name" but the *significant* letter in the title stood for *'All-purpose.'* Basic has been used for professional applications almost from its appearance. Logo has not, and was never intended to be.

At their inception, Basic and Logo were much simpler. Partly this was a result of choice by the designers: Basic was to be as easy as possible to learn and Logo was to encourage students to write most of their tools themselves, but there was also an element of 'the possible.' Kemeny and Kurtz were undoubtedly constrained by the small and slow machines they were working with and the need to build the system and compiler from scratch, in assembly language, with only themselves and their students as analyst/programmers. (They do characterise the GE-235 as "reasonably fast … with a 6 micro-second cycle time" [23 p5].) Logo, being developed slightly later, within a larger institution and on a single machine, probably suffered less in this respect, and certainly had the advantage of the system being written in Lisp, an existing, highly-adaptable, high-level language. I think it would be possible to make a strong argument that the simplicity of both should have been preserved, but the temptation to take advantage of developing Computer Science theory and faster machines probably made extension inevitable.

Both Basic and Logo have been extended far beyond the limits any of their creators could have imagined: the computer science of the 60's gave no inkling of the possibilities the personal computer and object-oriented programming would bring. Microsoft Visual Basic for instance, is to be found more in industrial applications that it is in a school. It is a graceful expansion, the language itself growing to include most of the trappings considered essential for complex and safe applications: declaration and strong typing of variables, procedures, explicit indefinite loops, implicit blocking and the incorporation of objects, but without losing its original form and flavour. Its syntax is still relatively straight-forward and is as suitable for education as the original.

Logo, in the form of MicroWorlds, is part of a full-blown multi-media/robotics environment and in 2010 is probably the only language that makes Cobol look small or offers the same invitation to write the same thing in so many different ways. Feurezig's successors seem to invent a new command every time they have a new idea, even when existing commands would seem to be perfectly suitable to the purpose. For instance, the Robotics version adds a completely new, quite separate set of commands to talk to the Logo RCX 'brick.' This leaves the existing, and quite adequate, 'Talkto' protocol in the main body of the language and separates Lego Robotics from the use of its rich array of logic. Redundancy in the language is

therefore rife, while it is axiomatic in computer language design that there should only be one way to do something [38 p527]. (See Lindsey and van der Meulen [39 p174] for a particularly salutatory example of the consequences of failing to observe this principle.) On the other hand, the MicroWorlds Backpack is a brilliant model of an object, thought the language itself cannot really be said to be object-oriented.

## 7 Conclusion

> *We have found that the transition from Logo to Basic is fairly easy for most students, whereas the transition from Basic to Logo is often incredibly difficult.* Brown and Rubinstein [19: 42]

Basic and Logo have both born out their creator's intention. Basic is easy to learn and has indeed become All-purpose. Logo has enriched countless classrooms and introduced students to new ways of thinking about, and expressing problem solutions.

Therefore Brown and Rubinstein, above, present a neat summation of the teacher's dilemma. Basic *has* proven to be easy to learn and get along with, but is not as likely to foster new ways of thinking and expression as Logo. This is Weyer and Cannara's point about "learning programming in full generality" [8 p3]. Does introducing students to Basic actually prevent, or mitigate against, as Brown and Rubinstein suggest, the development of a *full set* of programming tools? Logo's exponents insist it can do this, but it seems to be at the expense of leaving many students cold. Logo's advocates have not demonstrated the gains exposure to these ideas are supposed to bring. That said, given the enormous number of contributing factors, research demonstrating this is inherently extremely difficult, and the advocates of Basic have not done much better.

Essentially the teacher of today is in no better position than the pioneers, and is essentially dependent on *their own belief in the promise* that having students write programs will bring educational and other advantages. And because I am passionately in this school myself, I would be perfectly happy to do so in Logo or in Basic because I see the same promise in both, (though since I value simplicity, I would have a hankering after LogoWriter rather than the over-elaborated MicroWorlds). The pioneers of educational computing *knew* that programming a computer had educational benefits and set out to prove it, but I don't think the World listened.

## References


1. Feurzeig, W., Papert, S., Bollm, M., Grant, R., Solomon, C.: Programming-Languages as a Conceptual Framework for Teaching Mathematics. Final report of the first fifteen months of the Logo project. Washington, D.C, Bolt, Beranek and Newman. R–1889. 329 pp. (1969). ERIC ED 007 932
2. Ham, V.: Technology as Trojan horse. In: McDougall, A., Murnane, J.S., Jones, A., Reynolds, N. (eds.): Researching IT in Education Theory, practice and directions. pp. 25–38. Routledge, London (2010)

3. Kemeny, J., Kurtz, T.: The Dartmouth Time-Sharing System. Washington, D.C., National Science Foundation. 76 pp. (1967)
4. Cubin, L.: Oversold and underused: computers in the classroom. Harvard University Press, Cambridge M.A. (2001)
5. Cox, M.: The changing nature of researching IT in education. In: McDougall, A., Murnane, J.S., Jones, A., Reynolds, N. (eds.): Researching IT in Education Theory, practice and directions. pp. 11–24. Routledge, London (2010)
6. Munro, R.K.: Setting a new course for research on information technology in education. In: McDougall, A., Murnane, J.S., Jones, A., Reynolds, N. (eds.): Researching IT in Education Theory, practice and directions. pp. 46–53. Routledge, London (2010)
7. Ershov, A.P.: Programming: the second literacy. In: Lewis, R., Tagg, D. (eds.): Computers in Education, Vol. 1. pp. 1–8.. North-Holland, Amsterdam (1981)
8. Weyer, S.A., Cannara, A.B.: Children learning computer programming: experiences with languages, curricula and programming devices. Stanford, Calf., Stanford University. 228 pp. Technical Report No. 250. (1975). ERIC ED 111 347
9. Kurtz, T.: Basic. In: Wexelblat, R.L. (ed.): History of Programming Languages. Academic Press, New York (1981)
10. Papert, S.: Talking Turtle. [Videorecording]. Open University Educational Enterprises, England, Holt-Saunders (1993)
11. Dwyer, T.A.: Teacher-Student Authored CAI Using the NEWBASIC/CATALYST System. 22 pp. Pittsburgh Univ., Pa., National Science Foundation, Washington, D.C. (1970). ERIC ED 043 235
12. Dwyer, T.A.: Some principles for the human use of computers in education. International Journal of Man-Machine Studies 3**,** 219–239 (1971)
13. Dwyer, T.A.: Teacher/Student authored CAI using the NEWBASIC system. Communications of the ACM 15, 21–27 (1972)
14. Dwyer, T.A.: Heuristic strategies for using computers to enrich education. International Journal of Man-Machine Studies 6, 137–154 (1974)
15. Papert, S.: Mindstorms: children, computers, and powerful ideas. Basic Books, New York (1980)
16. diSessa, A.A.: Reflections on component computing from the Boxer project's perspective. Interactive Learning Environments 12, 161–165 (2004)
17. Papert, S.: Teaching Children Thinking. Cambridge, Massachusetts, Massachusetts Institute of Technology. 241 pp. (1971). ERIC ED 077 241
18. Feurzeig, W., Kukas, G., Faflick, P., Grant, R., Lukas, J.D., Morgan, C.R., Weiner, W.B., Wexelblat, P.M.: An Introductory LOGO Teaching Sequence: LOGO Teaching Sequence on Logic. Cambridge, Mass., Bolt, Beranek and Newman. 329 pp. (1971). ERIC ED 057 579
19. Brown, J.S., Rubinstein, R.: Recursive functional programming for the student in the humanities and social sciences. Irvine, California, University of California. 53 pp. UCI–ICS–TR–27a (1974). ERIC ED 108 664
20. Papert, S., Solomon, C.: Twenty things to do with a computer. Massachusetts, Massachusetts Institute of Technology. 240 pp. (1971). ERIC ED 077 240
21. Evens, P.: What is Logo? Deakin University Press, Geelong, Australia (1992)
22. Abelson, H., Barnberger, J., Goldstein, I., Papert, S.: Logo progress report 1973–1975. Cambridge, Mass., Massachusetts Institute of Technology. AI Memo 356, 22 pp. (1976) ERIC ED 128 181
23. Kemeny, J., Kurtz, T.: The Dartmouth Time-Sharing System. 76 pp. Washington, D.C., National Science Foundation (1967)
24. Kurtz, T.: Demonstration and Experimentation in Computer Training and Use in Secondary Schools, Activities and Accomplishments of the first year. Hanover, Dartmouth College, Hanover, N.H. 81 pp. (1968) ERIC ED 027 225

25. Putnam, R., Sleeman, D., Baxter, J.A., Kuspa, L.K.: A summary of misconceptions of high school Basic programmers. Stanford, CA., Stanford University School of Education. Occasional Report #010 (1984). ERIC ED 258 556
26. Johnson, D.C.: Algorithmics and programming in the school mathematics curriculum: support is waning—is there still a case to be made? Education and Information Technologies 5, 201–214 (2000)
27. Mayer, R.E.: Introduction to research on teaching and learning computer programming. In: Mayer, R.E. (ed.): Teaching and Learning Computer Programming. pp. 1–12. Lawrence Erlbaum, Hillsdale, New Jersey (1988)
28. Papert, S.: The Turtle's long slow trip: Micro-educational perspectives on Microworlds. Journal of Educational Computing Research 27, 7–27 (2002)
29. Murnane, J.S.: Models of recursion. Computers & Education 16, 197–201 (1991)
30. Murnane, J.S.: To iterate or to recurse? Computers & Education 19, 387–394 (1992)
31. Vitale, B.: Elective Recursion: A Trip in Recursive Land. New Ideas in Psychology 7, 253–276 (1989)
32. Murnane, J.S., Warner, J.W.: An empirical study of secondary students expression of algorithms in natural language. In: McDougall, A., Murnane, J.S., Chambers, D. (eds.): 7th IFIP World Conference on Computers in Education, Vol. 8. Computers in Education 2001: Australian Topics. pp. 81–86. Bedford Park, South Australia: Australian Computer Society (2001)
33. Davis, R.B.: Editorial. The Journal of Mathematical Behavior 10**,** 4 (1991)
34. Murnane, J.S.: The psychology of computer languages for introductory programming courses. New Ideas in Psychology 11, 213–228 (1993)
35. Papert, S.: Teaching children to be mathematicians vs teaching mathematics. Cambridge, Mass., Massachusetts Institute of Technology Artificial Intelligence Laboratory. 26 pp. (1971). ERIC ED 077 243
36. Ross, P., Howe, J.: Teaching mathematics through programming: ten years on. In: Lewis, R., Tagg, D. (eds.): Computers in Education, Vol. 1. pp. 143–148. North-Holland, Amsterdam (1981)
37. Juliff, P.: Programming—should we enjoy it or do it properly? In: Welch, R. (ed.): Ninth Australian Computer Conference. pp. 38–43. Hobart: Mercury-Walch, Hobart, Tasmania (1982)
38. MacLennan, B.J.: Principles of Programming Languages. Holt, Rinehart & Winston, New York (1983)
39. Lindsey, C.H., van der Meulen, S.G.: An Informal Introduction to Algol 68. North Holland, Amsterdam (1973)