The History of Computing Education Research

M. Guzdial, B. du Boulay,

1 The Scope of Computing Education Research

Teachers have been educating students about computing for many years. For almost as many years, computing education researchers have been studying, in particular, how students learn programming and how to improve that process. Programming languages such as Fortran (1957) and COBOL (1959) were originally invented to be easier than assembler and other early notations so that programming could be made available to a wider range of programmers. Programming languages such as BASIC (1964) and Pascal (1970) were invented explicitly to ease learning how to program. In the late 1960's, researchers started gathering data, studying how learners were learning programming, when they did not, and how they experienced programming.

We are limiting the scope of this chapter in three ways. The *content focus* of this chapter on the history of computing education research is very specifically on research on how students come to understand programming rather than on other aspects of computing such as databases, networks, theory of computation, and so on. The *time focus* of this chapter is from the first efforts to observe students learning programming (1967) up to the first offering of the *International Computing Education Research* (ICER) Conference in 2005. After 2005 we consider to be the "modern era" of computing education research.

Finally, we have filtered the historical events to focus on those that inform today's current work in computing education research. Computing education researchers have explored many paths over the last 50 years, and not all have been fruitful. We focus on the historical events that have the clearest connection to today's computing education research. As we review these events, we use three lenses:

- **Tools**: Educators in science, engineering, and mathematics may use computing technology, but computing education is necessarily tied to technology. Just as chemists cannot directly touch individual atoms and physicists cannot touch velocity, the bits and processes of programs cannot be directly sensed and manipulated by learners. Instead, we create tools that provide views on the program and its execution at different levels of granularity and from different viewpoints so making computing more malleable. What tools do we use, what have we used, and how do we design tools to serve our educational needs?
- **Objectives**: Why are we teaching students about computing? The answer has varied over the previous five decades, from preparing future programmers, to influencing how learners view their world, to a necessary part of general education like mathematics or history.
- **Research Methods**: How do we evaluate the effectiveness of computing education research, or answer our questions about how students learn? The earliest researchers applied the empirical methods they knew, but as our tools, objectives, and strategies have changed, we have also changed our research methods. We have even invented methods unique to computing education research.

The 1970's saw the emergence of many of the tools, objectives and research methods that underpinned work in later decades. Section 2 of this chapter looks at the computing education research conducted in the 1970s. It was an era when the programming technology available was fairly primitive, notably in terms of limited input and output devices. Section 3 explores the rapid developments in computing technology in the 1980's, particularly in interface capabilities, and how these affected computing education research.

Various claims had been made for the benefits of learning programming as a vehicle for developing various thinking skills, notably by Papert. While there had been some experimental work in the 1970's to test these claims, it was in the 1980's when educational researchers brought longitudinal research methods to bear on these issues. This was also the time when Cognitive Science research started to burgeon and this had an effect on the way that programming was conceptualised via cognitive models of the processes of programming, such as debugging. Section 4 explores the impact of these two disciplines on computing education research.

We then shift our focus in Section 5 to the organisation of computing education research activity over these decades, and in Section 6 we draw the threads together in anticipation of the next chapter in this book, looking at present and future of computing education research.

Given the given the breadth of coverage in this Handbook, there is a certain amount of overlap between this chapter and others. In addition to the next chapter on computing education today and tomorrow, we draw the reader's attention particularly to Chapter 3.4 on introductory programming and Chapter 3.6 on programming paradigms. This latter chapter extends the discussion in this chapter on notional machines, on the problems faced by novices and on programming paradigms that go beyond the procedural that is presented below. This chapter says a certain amount about teaching methods, but Chapters 2.2 and 3.2 extend the discussion much further. For a review of research on teaching methods for introductory programming, see Pears et al. (2007).

2 Early Studies in Computing Education Research

The late 1960's and early 1970's was already a period of dynamic activity in computing education research. There were two partially interacting streams of activity based on different objectives. The objective of the first stream was to understand the psychology around the activity of programming. The first stream was centred on the theoretical and empirical study of programming as a human skill, including issues of learning, as exemplified by the work of Green and his colleagues (see for example, Sime, Arblaster, & Green, 1977b, Page 52) and the publication of the first book on the psychology of computer programming (Weinberg, 1971). One driver for this stream was industry's need for programmers and their use of aptitude tests to identify them (see Chapter 3.4, Section 2.1 of this book). The second stream was centred more specifically on the learning of programming in educational settings as exemplified by the work of Papert and his colleagues (Papert, 1980a) and the publication of the first paper on the Logo programming language (Feurzeig, Papert, Bloom, Grant, & Solomon, 1969). The objective of this second stream was to understand the cognitive benefits of programming to the student programmer.

2.1 Learning Programming in General

Weinberg's book on the psychology of computer programming was quite short on the empirical results of both learning of programming and exercising the skill of programming but heralded an energetic decade of such work. Many of the research methods that survive into today's research emerged then, such as comparing novices and experts across different aspects of programming skill and analysing the ease of use and ease of learning of different representations of code. A review of the early work on empirical studies of programming can be found in du Boulay and O'Shea (1981). This section of the chapter draws heavily from that review, characterising the kinds of empirical work that were undertaken at that time. Note that the research predated personal computers, mice and much of the screen-based interaction we take for granted today. We return to this issue in Section 3 and show how the changes in hardware and software changed the focus of computing education research.

Two related areas of work were of special interest in this period: novices' difficulties with programming and language design for novices. Learning to program was already known to be difficult, so attempts were made to understand the specific nature of those difficulties as well as to explore how the design of programming languages and programming environments might mitigate those difficulties.

Novices' difficulties with programming were reported using observational research methods and small-scale laboratory-based studies of the skills and understanding of planning, coding and debugging. Two kinds of planning difficulty emerged: one concerned with the translation of a problem expressed in everyday terms into a formulation suitable for coding. The second concerned the degree of generality with which a problem was tackled, i.e. the requirement that a program would normally have to work with a range of possible data values, rather than a specific set. For example, concerning the issue of translation, Miller (1974) studied how novices approached the problem of developing algorithms for simple sorting tasks and found that disjunctions of properties caused more problems than conjunctions. He also reported that novices found it hard to turn everyday qualification statements (such as "PUT RED THINGS IN BOX 1") into the kind of conditional format (such as "IF THING IS RED PUT IN BOX 1") required by many programming languages (Miller, 1975). Concerning generality, Hoc (1977) studied programmers developing an algorithm for a change-giving machine and found that beginners tended to deal with specific instances of the problem rather than the general case.

In terms of coding, several researchers analysed the errors in novices' programs in various languages. For example, Youngs (1974) observed both novices and experts programming in a variety of languages (ALGOL, BASIC, COBOL, FORTRAN, PL/1). He categorised their errors into "syntactic," "semantic," "logical," or "clerical" and found that, for novices, semantic errors predominated and that they found them the hardest to debug. By contrast, experts' errors were more evenly distributed across the categories. Gannon (1978) found that novices' errors were clustered around specific language constructs. When the overall relative frequency of use of different constructs was taken into account, Youngs (1974) and Friend (1975), working on AID, a language similar to BASIC, found that conditional constructs were a common cause of errors. Friend also found that many syntax errors arose from novices overgeneralising the syntactic rules of the language.

There were several studies of debugging, mostly among experts but some among novices (Gould & Drongowski, 1974; Miller, 1974; Youngs, 1974). One conclusion that has stood the test of time is that there was great variability even amongst experts in terms of their debugging skill and speed (Gould, 1975). Eason (1976) found that novices' attitude to learning programming affected the degree to which they were willing to engage in the hard graft of learning debugging as well as the mastery of the tools needed (e.g. an interactive development environment) and their command languages.

The work cited above mostly used observational methods to characterise different aspects of novice and expert programming behaviour. A newer research methodology also started to emerge in terms of building models of programmer cognition. For example, Brooks (1977) developed a model of the cognitive processes in computer programming based on the (Newell & Simon, 1972) theory of problem-solving and the role of short-term memory. Brooks used think aloud methods and protocol analysis to uncover some of the implicit planning rules that programmers used to develop code. These issues are explored later in this chapter in Sections 4 and 6.

Language Design for Novices. In terms of high-level language design considerations, Weinberg (1971) argued for uniformity, compactness, and locality, while Barron (1977) for economy of concepts, uniformity, and orthogonality – each of them anticipating later work on Cognitive Dimensions (Green & Petre, 1996). One important line of experimental work concentrated on comparing, in laboratory settings, different ways of representing the code for flow of control and specifically conditionals. For example, Sime, Green and others conducted several studies on both the coding and comprehensibility of different ways of specifying conditional flow of control in tiny programs consisting only of conditionals (Sime, Arblaster, & Green, 1977a; Sime, Arblaster, et al., 1977b; Sime, Green, & Guest, 1977). They introduced the important distinction between "sequence information", which describes what a program will do under a given set of conditions, and "taxon information", which describes what set of conditions would cause a program to reach a given point. The experiments showed that a certain amount of redundancy in the notation reduced novices' errors and improved debugging success. In one notable study, the introduction of an **else** clause led to an enormous decrease in novice ability to solve taxon information problems.

Again using laboratory-based research methodologies, both Love (1977) and Shneiderman (1977) conducted experiments on the ability of novices and experts to memorise short programs, either in different forms (Love), or scrambled (not working) vs working (Shneiderman). Both sets of experiments showed that novices attended to the surface form of the code whereas more expert programmers paid attention to its deeper level structure, just as cognitive scientists had noted among novices and experts solving physics problems (Larkin et al., 1980).

2.2 Learning Programming in Educational Settings

One of the issues emerging in the 1970s centred on the design of the tools to be used by novices. A central issue was the choice of the novice's programming language and the programming environment in terms of their capability to make the workings of the underlying *notional machine* (the user-understandable semantics of the language) implied by the language both more explicit and more visible (du Boulay, O'Shea, & Monk, 1981). For example, Logo and its use of a Turtle to trace out a drawing were a good way of reifying the flow of control as a program executed and of relating procedures to sub-procedures. Similar notions were also applied in BASIC where, for pedagogical purposes, Mayer (1979) characterised the behaviour of a BASIC program in terms of a sequence of "transactions", and Barr, Beard, and Atkinson (1976) built the BIP system for teaching BASIC that highlighted each BASIC statement on the display as it was executed. Similar execution tracing systems were built for other languages such as Pascal (Nievergelt et al., 1978) and FORTRAN (Shapiro & Witmer, 1974). Note the further discussion of notional machines in Chapter 3.6 on programming paradigms.

The next two subsections explore work undertaken in two areas. The first is on learning topics through learning programming, such as thinking skills and cognitive science. The second concentrates on learning programming for its own sake.

Learning through Programming. Logo (Feurzeig et al., 1969) was a language derived from LISP (1958) and championed by Papert with the objective to teach mathematics and problem-solving through programming rather than to teach programming per se. (Papert, 1972, 1980a). On the back of attempts to see whether Papert's goals could be achieved (see Section 4 of this chapter) there were several detailed investigations of novices (both children and adults) learning Logo. For example, Cannara (1976) documented the difficulties that children had in learning Logo. These difficulties turned out to be both in terms of the language itself, such as the recursion/iteration distinction and the binding of argument values, but also in terms of more general aspects of programming such as the need for complete precision in coding (rather than assuming that the computer would figure out what they had meant to say) and the ability to break down complex problems into simpler parts. Similar difficulties were observed by Statz (1973) in her work with children, by Austin (1976) in work teaching student teachers, and by du Boulay (1978) also working with student teachers. So irrespective of the bigger issue of teaching mathematics through Logo programming, Logo as a first programming language produced its own share of difficulties for the learner. These arose for three reasons. First, the focus was much more on what programs could do rather than on the form of the programming language itself. Second, the language was developed by experts without input from learners. Third, the language was designed at the end of the 1960s when the psychology of programming was in its infancy.

Towards the end of the decade, Eisenstadt (1979) developed the declarative language Solo at the Open University (UK), designed with the objective to teach topics in cognitive science. This language was semantically similar to Prolog but was designed to have an easier syntax and a specialised editor which helped reduce syntax errors by including a predictive text facility. For example, if the user typed the "if" part of a conditional the "then" and "else" branches were automatically provided to be filled in. This kind of facility was crucial as students at the Open University often worked remotely without the benefit of a tutor nearby to help sort out programming difficulties. Students learning Solo were studied by Kahney (1982) who explored their different, and often incorrect, understandings of the notion of recursion, and by Hasemer (1983) who looked at their debugging behaviour and built a tool to support debugging based on those observations.

Learning programming in its own right. While the developers of Logo and Solo had objectives beyond simply teaching programming, other languages were studied whose aims were more to teach computing. We have already referred to the work of Friend (1975) on AID (similar to BASIC). Like Youngs (1974), she catalogued the errors that students made and also found that the required number of conditions, loops and subroutines was a good measure of how difficult beginners would find a problem.

Pascal (1970) was designed for novices and widely used in computer science departments as a vehicle for undergraduates to learn programming. Pascal was the language of the first

Advanced Placement CS exam in 1984. There were various critiques of its problems, which had little impact on its popularity (Habermann, 1973; Lecarme & Desjardins, 1975; Welsh, Sneeringer, & Hoare, 1977). Ripley and Druseikis (1978) studied computer science students' errors and found that about 60% of the errors concerned punctuation, mostly the use of the semi-colon, and that variable declarations were also another problematic area. A similar study by Pugh and Simpson (1979) came to similar conclusions about the use of the semi-colon. We see similar results today in analyses of Java error messages, and with the worldwide scale of current data collection efforts (Altadmri & Brown, 2015), we know that these problems are common and not just local to a particular study.

A pervading problem for learners of Pascal as well as other languages was misunderstanding the capability of the computer. A study by Sleeman, Putnam, Baxter, and Kuspa (1986) of Pascal learners noted that students attributed "to the computer the reasoning power of an average person," which Pea named the "superbug" problem (Pea, 1986). This was an issue already identified by Cannara (1976) in relation to Logo.

2.3 Research on teaching methods

Chapters 2.2 and 3.2 in this Handbook focus on pedagogic methods in the broad. Here we concentrate on issues specifically associated with early computing education research. One of the questions emerging in this period was "what is the best programming language for novices" (Tagg, 1974), though we might now disagree that such a question is useful in its most general form. An interesting variation on this question was whether one should start with a low-level language (e.g. assembler) or a high-level language (e.g. BASIC). Weyer and Cannara (1975) compared teaching Logo and then SIMPER (a low-level language), SIMPER and then Logo, with teaching Logo and SIMPER at the same time. They found that the joint approach worked best in that the differences and similarities between the languages helped understanding despite some novices exhibiting a certain amount of muddling up of commands between the languages.

Various experiments were conducted on characterising the notional machine, as we have indicated above. For example, Mayer (1975, 1976) showed that providing a simplified model of the FORTRAN notional machine was helpful in both coding and comprehension, and this was most pronounced where the programming problems were more difficult. Mayer (1975) found only a limited positive effect in learning FORTRAN. Shneiderman, Mayer, McKay, and Heller (1977) found no effect in learning FORTRAN. Researchers also examined the utility of using flowcharts as a learning aid. Brooks (1978) found that a variable dictionary (an annotated listing of variables in a program) provided more effective assistance than a flowchart for students were working in FORTRAN.

The issue of learning in pairs or groups was also explored. Lemos (1978, 1979) found that students who debugged COBOL programs in small groups were more favourably disposed towards programming and came to understand the language better. Likewise Cheney (1977) found that students who worked on their programming assignments in pairs learned more effectively – an interesting foretaste of much later research on pair-programming (Bryant, Romero, & du Boulay, 2008). Bork (1971) explored the best order for introducing programming concepts, contrasting top-down ("whole program") with bottom-up ("grammatical"). Lemos (1975) found no difference between the two approaches for FORTRAN but Shneiderman (1977) argued for a "spiral approach" that amalgamated both methods.

The main outcomes arising from the work in the 1970's was greater clarity about the difficulties that novices faced in learning to program. These difficulties included (i) learners understanding what programming was for, (ii) how they should reconceptualise a problem in terms of the kinds of structures and mechanisms available in the programming language being used, and (iii) how those structures and mechanisms functioned when a program was executed. There were also advances in the tools, notably in language design and in programming environment design that helped to mitigate these issues.

3 Re-designing the Learner’s Interface for Computing Education

As mentioned earlier, the nature of computing education research was strongly influenced by the hardware and systems software available at the time. Papert’s Logo and the rest of the programming languages discussed above from the 1960’s and 1970’s were designed to be a predominantly text-based experience. Logo was originally programmed using a teletype. The first uses of Logo by children were to manipulate text language, like a program to make poems or play games (Papert, 1971; Papert & Solomon, 1971). The turtle came shortly thereafter, but was still controlled from a text-based interface. Starting in the 1970’s, researchers explored how the interface might be changed so that the tool might better fit the objectives.

3.1 Smalltalk

Alan Kay visited Seymour Papert during the early Logo experiments and saw the potential learning benefits of computers (Kay, 1972). He reconceived the challenge of building a programming language for students as the challenge of building computational media for students, where programming was part of an authorial or creative process. Where Papert had an objective of providing the computer as an object to think with (Papert, 1980b), Kay saw the computer as a tool for expression and communication, as well as reflection and problem-solving (Kay, 1993).

Kay designed Smalltalk, the first language explicitly called “object-oriented.” The earliest version of Smalltalk, Smalltalk-72, had a syntax similar to Logo (Goldberg & Kay, 1976). The later versions of Smalltalk (through the middle of the 1970’s) were often used by children (Kay & Goldberg, 1977), but the focus of Smalltalk development shifted to support professional developers. By the time Smalltalk was released in 1981, it had evolved into a tool that was much more challenging for children to use (Goldberg & Robson, 1983).

When Kay and his group were designing Smalltalk, they wanted learners to be able to use the environment as a creative medium. They wanted students to be able to draw and use their drawings as part of animations that were programmed by computer. They wanted students to be able to play music and write programs that would make music. They developed the desktop user interface, with overlapping windows containing multiple fonts and styles, pop-up menus, icons, and a mouse pointer (Kay & Goldberg, 1977). Computer icons were first invented in Smalltalk-72 (Smith, 1975). The user interface that we use daily today was designed in order to achieve Kay’s vision of the computer as a creative medium, where programming was one of the ways in which learners would express and communicate in this medium.

The research that Kay and his group did with Smalltalk was observation-based. Students would visit their lab at the Xerox PARC to test the feasibility of students working with the medium they were inventing. Smalltalk was used in a Palo Alto school for a while as a classroom-based experiment (Kay, 1993). In those studies, the team documented some of the challenges that students had with object-oriented programming, such as class-instance differences and finding functionality in a large class hierarchy.

3.2 Boxer and Programmable Toys

Of those first programming languages designed for learners (e.g., Logo, SOLO, Pascal), Smalltalk has arguably had the largest impact on computing today, because of the user interface inventions it advanced. Other early computer education research groups recognized the importance of using the advancing user interface technologies in order to go beyond simple text to provide a rich computational environment for learning. There are two threads of this work that has had impact on today's work. One is Boxer, which has given us theory for how students come to understand computing. The second are the interactions between toys and programming that have had an influence on today's blocks-based programming languages.

diSessa and Abelson developed Boxer as a successor to Logo (diSessa 1985; diSessa & Abelson, 1986). Like Kay, their goal was to use more advanced user interface techniques to improve the learning experience. Unlike any previous effort, Boxer gave semantic meaning to these user interface elements. Everything in Boxer was in a graphical box on the screen, both data and procedures. References to variables and binding to arguments (mentioned earlier as a challenge in Logo) become references to concrete named boxes with visible values on the screen. Boxer was developed with a similar objective to Smalltalk, which was to serve as a medium for computational literacy (diSessa, 2001). The idea of using graphical elements as semantically meaningful parts of a beginner's programming language started in Boxer and has led us to the blocks-based languages that are developed and studied today.

The earliest Logo turtle was a physical robotic device, which made the user's experience of programming as more than just text from the beginning. In the 1970's, physical programming of the turtle was made possible through Radia Perlman's buttons and slots, which moved away from text or graphics for the learner's programming interface (McNerney, 2004). In McNerney's (2004) review of physical programming environments, he draws a direct line from Perlman's buttons and slots, to the development of a variety of connections between Logo and Lego building sets (Resnick, Martin, Sargent, & Silverman, 1996), including MultiLogo (Resnick, 1990). Allison Druin built physical programming environments where students built physical, interactive spaces for story-telling, a different objective than previous programming activities (Druin et al., 2011; Sherman et al., 2001). Resnick's work on developing programming languages that were more accessible for children programming intelligent Lego bricks led some of the early blocks-based languages, including Scratch, the most well-known blocks-based programming language today (Maloney et al., 2004; Maloney, Peppler, Kafai, Resnick, & Rusk, 2008).

The main outcome of the work described in this section was the emergence of the components of computer interfaces that we now take for granted including windows, icons, pointers and mice. This outcome has certainly influenced the design of development environments. Specifically for computing education research, a second but important theme is consideration of non-textual elements (including physical devices) in supporting student learning and programming.

4 Enter the Education Researchers

Two related disciplines started to impact on computing education research. One was Education research, bringing with it an emphasis on research using large cohorts over extended periods of time. The other was cognitive science which emerged in the 1970/1980s and its research methodology based on modelling human cognition. The involvement of these researchers broadened computer education research. Cognitive researchers modelled the user, not just the language or the interface. Both in general and in educational settings, the cognitively-informed researchers who then started exploring computing considered cognitive outcomes and transfer as well as the learning process.

4.1 The Impact of Education Researchers

Earlier work had already shown that Logo as a programming language had its own share of difficulties for novices (see for example, Cannara, 1976). It was a series of studies by Pea, Kurland and their colleagues that now examined whether learning Logo had the benefits claimed for it in terms of increased problem-solving ability and other thinking skills (Papert, 1980a). Taking a developmental and cognitive perspective, Pea and Kurland (1984) undertook a theoretical analysis and review of how learning to program might, in principle, impact on children's ability "to plan effectively, to think procedurally, or to view their flawed problem solutions as 'fixable' rather than 'wrong'". Now the research methodology shifted away from small-scale laboratory-based methods to longer-term evaluative studies in authentic educational settings. In an empirical study of 8-10 year old children learning Logo over a year, Pea, Kurland and their colleagues found that these children were no better at a planning task at the end of the year than children who had not learned Logo (Pea, Kurland, & Hawkins, 1985). In a much larger study involving 15-17 year old students learning a range of languages over a year, including Logo over 9 weeks within that year, they found similar results as well as misunderstandings of the Logo language and how it worked (Kurland, Pea, Clement, & Mawby, 1986). Likewise, Kurland and Pea (1985) found in a small study with children that they developed various incorrect mental models of how recursion worked, similar to those reported in other studies of the understanding of recursion.

Papert and Pea famously debated these studies in 1987. Papert argued that Pea was exhibiting "technocentric thinking," by studying the technology as opposed to studying the educational culture that could be created with tools like Logo (Papert, 1987). Papert argued that we were too early in the development of new kinds of educational culture that computers might facilitate to evaluate it in a treatment model. Pea argued in response that we must study what we build, and that we can too easily fool ourselves into believing that our interventions work without careful studies (Pea, 1987). The Pea and Papert position papers highlight the sharp contrast between computing and educational research in terms of both their objectives and their research methods. Is the goal of teaching programming to improve the classroom or re-make it into something new which cannot be studied in the same way? Do we use the traditional methods of educational research, or do we need new methods?

These rather negative findings for Logo's influence on the development of thinking skills was underlined in a wide-ranging review of the literature covering Logo, BASIC and Pascal (Palumbo, 1990). However, Palumbo criticised much of the research he reviewed for not paying proper attention to "five critical issues concerning this area of research: (a) sufficient attention to problem solving theory, (b) issues related to the programming treatment, (c) the programming language selected and the method of instruction, (d) system-related issues, and (e) the selection of an appropriate sample."

Following this, Palumbo and Reed (1991) attempted to deal with these critical issues and compared a group of students learning BASIC with a group learning computer literacy, which focused on basic skills involved in computer operations, and did find evidence of improved problem-solving in the group who had learned BASIC. In a similar fashion, Carver found that the skill of debugging could be learned by the 8-11 year olds through experience with Logo programming (Carver, 1986). Carver's critical insight was that transfer occurred only when that transferable skill was made an explicit part of the Logo curriculum, rather than hoping it might be learned simply in passing (Carver, 1986).

4.2 The Impact of Cognitive Science

Brooks' work, mentioned earlier, continued into the 1980's with a theory of the comprehension of programs that tried to explain the great variability in skill amongst both experts and novices (Brooks, 1983). In a parallel vein, Soloway and Ehrlich (1984) empirically explored questions around programmers' knowledge of "programming plans – stereotypic action sequences", and of the "rules of programming discourse . . . which govern the composition of plans into programs." They used both fill-in-the-blank methods and recall methods to demonstrate the effects of these two kinds of programming knowledge. Likewise Wiedenbeck (1986) introduced the notion of beacons, "lines of code which serve as typical indicators of a particular structure or operation", that assisted experts better than novices to recall programs that they had studied. This work on plans and beacons reflected the interest in mental models and schemata in the cognitive science of the time (see for example, Johnson-Laird, 1983).

With the emphasis on problem-solving emerging from cognitive science, it was not surprising that work on understanding the nature of debugging was undertaken. Lukey (1980) built a system, PUDSY, to embody and model his theory of how programmers debug (Pascal) programs. His system made use of rules to segment programs, a description of the flow of control, the recognition of debugging clues and simple data flow analysis, but as far as we know there was no empirical work based on the system. In a series of empirical studies, Katz and Anderson (1987) observed and analysed students' debugging strategies in detail as applied to LISP programs. Among other results they identified a range of students' bugs in LISP programs, in a similar fashion to Soloway and his colleagues' analysis of bugs in student Pascal programs (Johnson, Soloway, Cutler, & Draper, 1983; Spohrer et al., 1985). These kinds of analysis provided evidence about which aspects of these languages required extra tutorial support as well as providing the basis for progress on automated tutors and debugging systems.

There was also interest in novices learning Prolog (Taylor & du Boulay, 1987). This is a declarative language with a complex internal reasoning mechanism that novices find difficult to comprehend, even when (and sometimes especially when) the "trace mechanism" is switched on showing the internal reasoning steps. In effect, the work on trace mechanisms was an attempt to provide the programmer with a Prolog notional machine (Eisenstadt & Brayshaw, 1990), very different of course from the Logo notional machine mentioned earlier. The work on programming plans and beacons applicable to procedural languages morphed into understanding Prolog programmers' knowledge in terms of Prolog schemata. These were stereotypical, slightly abstracted chunks of Prolog code (Gegg-Harrison, 1991). A useful guide to this work on Prolog can be found in Brna, du Boulay, and Pain (1999).

New kinds of tool for novices were also emerging in the form of tutors, error diagnosis systems and support environments for learning programming, including tutors for learning Prolog (see du Boulay & Sothcott, 1987, for a review). A notable example emerged from cognitive science: Anderson and Reiser (1985) developed a tutor for LISP, arising both out Anderson's cognitive science learning theory "Adaptive Character of Thought" (ACT) and his analysis of students learning to program, mentioned above. The strategy in this work was to exploit a cognitive science view of how a tutor might support a student's declarative understanding of programming ("what it is") as it developed into procedural skill ("how to do it"). This tutor was the forerunner a large family of tutors that modelled the skills of a domain, in this case programming in LISP, based on a production-rule representation and were able to guide students step by step through problem-solving (Anderson, Corbett, Koedinger, & Pelletier, 1995). Empirical evaluations of the tutor largely showed that it helped learners to master simple programming more quickly than other methods (e.g. lectures and text book) but that it did not necessarily provide deeper understanding than the other methods. This was not the only tool developed at that time to assist computer science students that took a cognitive science stance towards learning programming. We have already referred to Soloway and his colleagues' analysis of programming knowledge in terms of "plans" (Soloway & Ehrlich, 1984). In addition, Johnson and Soloway (1987) developed an error diagnosis system for student Pascal programs, PROUST, based on these ideas.

4.3 Phenomenographic research

Cognitive science and its notion of mental models were not the only perspective on computing education research. By contrast, the phenomenographic perspective studied students in authentic settings to capture a sense of their personal experience of learning computer science. Phenomenography entered computing education research through educational psychologist Ference Marton and his student, Shirley Booth. For a brief guide to several approaches for research in computer science education including cognitive science and phenomenography, see Ben-Ari, Berglund, Booth, and Holmboe (2004).

Phenomenography emphasised the individual quality of the learning experience of each student in relation to the context in which it was learned.

> "Fundamental to an understanding of the phenomenographic approach is to realise that its epistemological stance is *not* "psychological", treating man and his behaviour as separable from the world in which he moves and lives. *Nor* is it "mentalist", treating cognition and cognitive acts as isolated to the mind and separable from the one who lives through them. The phenomenographic epistemological stance is that man is in relation to his world, and that cognition is such a relation . . ." (Booth, 1992, Page 52)

An interesting example of the approach was the thesis work of Booth on undergraduates learning computer science. In addition to teasing out the learner's ways of understanding concepts such as recursion, she used an interview technique asking oblique questions to gain a sense of what they thought it meant and what they believed it took to learn to program (Booth, 1993). The phenomenographic approach was later linked with Activity Theory to study how the learning situation for students doing a distributed course in computer systems influenced their learning (Berglund, 2002). The phenomenographic approach was also used to explore the learning of advanced network concepts in a distributed course (Berglund & Pears, 2003).

There were three main outcomes of the work described in this section. First was the use of large cohorts and longitudinal studies to study learning programming in classrooms rather than in laboratories. The second was the emergence of cognitive science to enable more detailed understanding of novice and expert programming skills, runnable models of human programming activity, and tools to support that activity, to be built. The third were the use of phenomenographic methods to capture the individuality of the experience of students learning programming and other computing concepts.

5 Computing Education Emerges as a Research Discipline

This section looks at the emergence of organizations for computing education research rather than at the content of the research itself as we have done in the previous sections.

Two international groups were formed in the 1980's to promote and support computer education research. One was the Psychology of Programming Interest Group (PPIG), formed in the UK in the 1987 and holding its first workshop in 1989. The other was the Empirical Studies of Programmers (ESP) formed in the USA and held its first workshop in in

1986 (Soloway & Iyengar, 1986). Both groups ran a series of workshops and conferences: ESP until 1997 but with PPIG still doing so to the present day. A brief history of ESP can be found on the PPIG website as follows:

> "The ESP series was managed by the USA-based Empirical Studies of Programmers Foundation. The last published list of the Board of Directors of that Foundation (in 1997) was Deborah Boehm-Davis, Wayne Gray, Thomas Moher, Jean Scholtz and James Spohrer.
>
> There were seven ESP conferences, all held in the USA. The research coverage of the series was very similar to the European (UK-based) PPIG series, which is the host organisation for this newsletter. Many people considered ESP and PPIG to be sister organisations. All ESP conferences except ESP 3 published formal proceedings volumes. Until ESP 6, the publisher of those proceedings was Ablex. The proceedings of ESP 7 in 1997 was published by the ACM Press. An attempt was made to convene an ESP 8 meeting that would have been held in 1999, although insufficient submissions were received for the meeting to be viable. The papers received were instead published as a special issue of the International Journal of Human-Computer Studies (Volume 54, Number 2, published February 2001)." (Alan Blackwell, PPIG Website, http://www.ppig.org/news/2006-06-01/whatever-happened-empirical-studies-programmers)

Over the years both groups were concerned with general issues in computer education research as well as studies of expert programmers. To give an idea of the flavour of ESP work, their first workshop included papers on novice/expert differences in debugging and in specifying procedures, cognitive processes in program comprehension, novice debugging in LISP, bugs in novice programs in Pascal, novice problems in coding BASIC and a plea that expert professional programmers should also be studied. This range of papers suggest differences in objective among these researchers. For the early Logo researchers, the goal of teaching programming was to change thinking. For some of the first ESP researchers, the goal of teaching programming was to get students to exhibit expert behaviour, so it was important to study expert professional programmers and to contrast novice/expert differences. A useful account of what was known about the psychology of programming at this time can be found in (Hoc, Green, Samurcay, & Gilmore, 1990).

The ACM Special Interest Group in Computer Science Education was one of the first organizations focused on Computer Science Education. Their annual symposium was started in 1970 and continues to attract over 1000 attendees annually. The SIGCSE Symposium was initially focused on providing a forum for teachers of computing education to share their best practices, but research results were often presented at the symposia. ACM SIGCSE's non-US conference, Innovation and Technology in Computer Science Education (ITiCSE), started in 1996. ITiCSE has been particularly important for its "working groups" that served as fertile ground for developing computing education research. Other conferences, such as IEEE Symposium on Visual Languages and Human-Centric Computing (started in 1984) also often included computing education research results.

In 2001, the McCracken Working Group invented a research method for computing education research, the Multi-Institutional Multi-National (MIMN) study (McCracken et al., 2001). McCracken and his colleagues recognized that the validity of any study at one institution was subject to critique because of experimental variables that might be unique to that institution or in common with only a subset of institutions. These threats to validity made it difficult to make progress in computing education research. The McCracken Working

Group used a common task across five different institutions in four different countries, in order to avoid the limitations of single institution studies. The results were convincing to the computing education research community, and the surprisingly poor performance was a clarion call to make change in how we teach (Lister, 2011). Soon, other MIMN studies were conducted (e.g., Lister, Box, Morrison, Tenenberg, & Westbrook, 2004), and MIMN studies were generalized and defined as a research method (Fincher et al., 2005).

Some years later, the McCracken Working Group study was replicated. The results were not better, but were no longer surprising (Utting et al., 2013). It is a measure of progress in computing education research that we now better understand computing education and the challenges in that field.

In the United States, there was growing recognition that more computing education research was needed. Tenenberg, Fincher, and Petre began the Bootstrapping project which helped develop computer science educators who wanted to become researchers. The success of Bootstrapping led to another project in the United States (Scaffolding), and more capacity-building projects around the world (Fincher & Tenenberg, 2006). Today, most computing education researchers in the United States were part of one of the capacity-building projects in the early 2000's or are a student of someone who was.

Because of the capacity-building efforts, there were more active computing education researchers than ever before. The community needed its own conference. While the ACM SIGCSE (Special Interest Group on CS Education) technical symposium had been around since the late 1960's, the focus there was on supporting practitioners, not on advancing research. Fincher, Anderson, and Guzdial were the first organizers of the ACM SIGCSE International Computing Education Research (ICER) conference in 2005 (Anderson, Fincher, & Guzdial, 2005). ICER became the best known and most respected computing education research venue globally. ICER continues to grow, attracting 150 participants in this last year. ICER papers are wide-ranging and cover tools, objectives, and research methods.

6 Research Questions in Computing Education Research

The list of research questions that have been explored in the approximately five decades of computing education research could already fill a book. For an excellent overview of the first four decades of this field (see Robins, Rountree, & Rountree, 2003). In this section, we consider three that are often revisited and connect across these decades.

6.1 Developing a Notion of Programming

One of the most significant problems in learning to program is developing a mental model of what the computer is doing when it executes a program. The problem was first identified by du Boulay in 1986, wherein he coined the term notional machine to describe a model of how the computer interprets and executes a program (du Boulay, 1986). The first step in the process is recognizing that the computer does not contain a homunculus which is trying to understand the program. The belief that the computer contains a kind of human inside it is called "the superbug" by Pea (1986). Resnick identified the challenge that the computer is an external agent, and a robot being programmed is yet another agent (Resnick, 1990). Children growing up rarely face the challenge of giving detailed process instructions to another agent, let alone the complexity of instructing a non-human agent who does not share a common language.

Some researchers in the 1960's believed that eventually computers would understand humans and natural language so that the task of programming would go away (Greenberger, 1962). Perlis argued in 1961 that we would never reach that stage, that there would always be "friction" because of the mismatch between humans and computers. "Procedural literacy" is what Mateas (2008) called the knowledge and skills needed to overcome that mismatch. The challenge of developing a mental model of the notional machine is one that researchers have revisited every decade since the question was first defined (Sorva, 2013).

A similar problem to developing a mental model of the notional machine is developing a mental model of the programming process (Garner, Haden, & Robins, 2005; Joni & Soloway, 1986). Setting aside the complexity of syntax and semantics, students struggle with the notion of the interpreter or compiler, the act of debugging, and how the output or result of a program might be found (Sleeman, 1986). A decade after Sleeman first identified these issues, Clancy and Linn reported that students struggled to understand how all the different aspects of programming fit together (Clancy & Linn, 1992). A further decade later, researchers described the puzzling debugging strategies of students (Murphy et al., 2008). Graphical user interfaces and physical programming may have increased the complexity for students to try to understand the process of creating software and answering questions like, "Where is the program that's causing this behaviour?" For example, students do not understand what their programs did when they were run (Hundhausen & Brown, 2007), nor where their programs are stored (Resnick, 1990). More positively, the advent of the Internet may allow us to explore recording student process information while programming, and studying that may give us new insights into the development of models of how students develop software and what they think is going on (Hundhausen, Olivares, & Carter, 2017).

6.2 Programming as a Notation for Thinking

Human languages, especially literacy in written language, has had a dramatic impact on society (McLuhan, 1962, 1964) and even on individual readers' brains (Wolf, 2007). The early computing education researchers expected that programming and computational literacy would have a similar impact (Kay, 1993), but they recognized that the design of the language would be critical for broad social and individual impact (diSessa 1985).

Like Papert, diSessa and his students argued that programming would lead to different kinds of understanding in mathematics than traditional pen-and-paper based forms. Turtles allow students to explore issues as complicated as differentials and general relativity through programs that can be more accessible than the equivalent equations (Abelson & DiSessa, 1986). Programming can be used by students to invent new kinds of graphical notations to represent variables of interest and their relations (diSessa , Hammer, Sherin, & Kolpakowski, 1991). In physics, equations better represent balance, but programs can be better for representing causal and temporal relationships (Sherin, 2001).

The form of the programming language has been studied since the development of graphical user interfaces. Green and Petre and colleagues studied a variety of graphical notations (such as LabView and petri nets) and found that textual programming interfaces led to better performance by programmers (Green & Petre, 1992; Green & Petre, 1996; Green, Petre, & Bellamy, 1991). They suggest that the superiority of textual languages was likely not inherent, but learned. We have much more experience with textual notations than with visual notations (Petre, 1995).

The trade-off between graphical and textual languages may be different for beginners. Hundhausen, Farley, & Brown (2006) found a visual, direct-manipulation language led to students writing programs sooner than another group of students using a textual language.

Weintrop and Wilensky have found that some of the conditionals and iteration errors that students make with text-based languages are much less common when students use graphical blocks-based languages (Weintrop & Wilensky, 2015). Several studies have shown that students learning blocks-based languages can transfer their knowledge to more traditional text-based languages (e.g., Weintrop & Wilensky, 2015). The future of computational literacy will likely be of mixed modality. Students will use different kinds of programming languages at different stages, e.g., blocks-based languages as beginners, text-based languages if they become computing professionals, and perhaps domain-specific languages with graphical or textual forms for end-user programmers.

6.3 Representing Execution

In 1987, Brown introduced the idea of animating algorithms in order to make them accessible by students and professional programmers (Brown, 1987). For decades, we have been asking if animated representations of program execution do help with understanding, when they might, and how best should they be designed. A challenge in this research is deciding the objective of the animation. Is the objective to improve learning of the algorithm, to provide new insights when the programmer already understands the program, or to support debugging? Sorva's (2012) dissertation is an excellent starting place for understanding program visualization.

In general, there is little evidence that viewing algorithm animations leads to improved learning about the animations (Hundhausen, Douglas, & Stasko, 2002). However, we can use algorithm animations as part of other learning goals and develop successful learning activities beyond just viewing. For example, Stasko found that students learned from building the animations, rather than just viewing them (Stasko, 1997a, 1997b) and from answering questions about static representations after viewing a dynamic animation (Byrne, Catrambone, & Stasko, 1999). While animations themselves may have limited impact on learning, they are motivating and can engage students for greater time-on-task and thus greater learning (Kehoe, Stasko, & Taylor, 2001). Sorva has suggested a different role for program execution visualizations – to teach a mental model of the notional machine, rather than to teach the particular algorithm being taught (Sorva, 2012; Sorva, Sirkå, 2010).

7 Conclusion: Future Research Questions in a Historical Context

We started this chapter by mentioning the tools, objectives and research methods of CER.

The computational power and interface capability of the tools available to learners have increased greatly since the early days of CER (Good, 2011), as has the ubiquity of devices to learn with and on, including smartphones, tablets, tangible computing and a range of cheap hardware kits such as the Arduino. The kinds of problem that a beginner programmer can tackle are richer and more varied, and no longer restricted to printing "Hello World" or printing out the Fibonacci sequence, not least because the range of input and output devices has also developed dramatically since Turtles first crawled the floor. Indeed, programming novices may even use their own bodies as input devices for coding up dance and other movement sequences (Romero, du Boulay, Robertson, Good, & Howland, 2009). We see that the hardware and systems software available to students and teachers influence what students were taught and how they were taught. The advent of more computational materials suggests the need to explore how these new media influence student learning.

The objectives argued about in the Papert vs. Pea debate mentioned earlier have now re-emerged in two new ways. First there is interest around the world in introducing school pupils to programming at an early age (see e.g. http://www.computingatschool.org.uk/ and http://www.csforall.org/). This has happened in response to the increased importance of computing in our everyday lives and the expectation that informed citizens should have at least some understanding of programming to be able to function effectively, or possibly choose a career in computing. This has reawakened many of the questions around how novices can come to understand programs and programming as a process that we have sketched earlier. Even the word “algorithm” has now entered everyday vocabulary, though perhaps with different meanings than was meant when Perlis and Snow talked about teaching algorithms to all undergraduates in (Greenberger, 1962). However, children are now so surrounded by computers of many kinds that the issue of understanding in principle what a computer and a program might be used for may be less problematic than in the early days (du Boulay, 1986). The ubiquity and invisibility of programs might make developing understanding of how programs work even harder. It’s hard to learn about something one cannot see.

International interest in making computing education accessible to all puts the computing education literature in a new light. The studies we have reviewed here do not always tell us who was doing the learning. Papert’s studies were mostly with children, but we do not have data about class or socioeconomic status. Most of the studies in computing education that we have reviewed here have been undertaken with computer science students in higher education, which is a privileged subset of students (Margolis & Fisher, 2002). Many studies published in venues like ACM SIGCSE do not tell us the gender of the participants. There are very few studies of students with learning disabilities or below-average intelligence (Ladner & Israel, 2016). Indeed, we now have a whole new cohort of the population who need to learn to program their smart devices and their homes (see for example, Blackwell, Rode, & Toye, 2009). We need to revisit past studies and consider if the results might have been different with a broader sample of study participants.

The second echo of the Papert vs. Pea debate centres on the notion of computational thinking (Wing, 2006). This develops some of Papert’s ideas about how programming offers a model of how to think effectively, solve problems and manage complex situations that can be applied in other areas of life. So school pupils and college students are offered computational thinking courses that help them practice some of the thinking skills that programmers apply, without necessarily learning to program. The echo of Pea’s paper still can be heard, as the field struggles to measure computational thinking (Roman-Gonzalez, Perez-Gonzalez, & Jimenez-Fernandez, 2016).

In terms of research methods, we now see a wide variety of methods derived from educational research including long and short-term evaluations, data-driven and at-scale methods (Moreno-León, Robles, & Román-González, 2017), design-based research from learning sciences (Ericson, Guzdial, & Morrison, 2015), action-based research by teachers (Ni, Tew, Guzdial, & McKlin, 2011), and learner-centred design methods for building programming environments for novices (Good, 2011; Guzdial, 2015; Howland, Good, & du Boulay, 2013). From psychology and cognitive science we see both qualitative and quantitative analyses of programming processes involving a range of data-capture technologies such as eye-tracking (Bednarik & Tukiainen, 2006), think aloud protocols (Bryant et al., 2008), as well as modeling. We also see measures of cognitive load applied to learning programming, from validated instruments (Morrison, Dorn, & Guzdial, 2014) to setting a background task to be undertaken in parallel (Abdul-Rahman & du Boulay, 2014).

We are limited in our research methods by who comes to our research enterprise today. In the early years, computing education research occurred across campus, e.g., Perlis described research in business and economics departments in (Greenberger, 1962). Section 4 described the education researchers entering into computing education research. Today, most authors publishing at ICER are computer scientists or have a strong computing background. We have too few education researchers (or learning scientists, or psychologists), which means that we have too few people bringing new research methods into the community. We need that rich interdisciplinary background that our field used to enjoy in the past.

Back in the 1960's, the two most prominent objectives for starting computing education were (1) to prepare future programmers, see Sackman (1968) cited in Ensmenger (2010), and (2) to use computing as a tool for thinking and problem-solving (Greenberger, 1962). The former focused on industry-standard programming tools, while the latter encouraged the development of new learner-focused programming tools. Computing education research started when scientists began asking whether these efforts worked. The research methods selected have always been inextricably tangled with the objectives. As we define new roles for computing in people's lives, we will be defining new objectives, creating new tools, and applying a variety of research methods as we try to understand what happens when humans learn to control machines and to measure how successful the humans are at the task.

References

Abdul-Rahman, S. S., & du Boulay, B. (2014). Learning programming via worked-examples: Relation of learning styles to cognitive load. *Computers in Human Behavior, 30*, 286-298. doi:http://dx.doi.org/10.1016/j.chb.2013.09.007

Abelson, H., & DiSessa, A. (1986). *Turtle geometry: The computer as a medium for exploring mathematics*: MIT press.

Altadmri, A., & Brown, N. C. C. (2015). *37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data*. Paper presented at the Proceedings of the 46th ACM Technical Symposium on Computer Science Education, Kansas City, Missouri, USA.

Anderson, J. R., Corbett, A. T., Koedinger, K. R., & Pelletier, R. (1995). Cognitive Tutors: Lessons Learned. *The Journal of the Learning Sciences, 4*(2), 167-207.

Anderson, J. R., & Reiser, B. J. (1985). The LISP Tutor. *BYTE, 10*(4), 159-175.

Anderson, R., Fincher, S. A., & Guzdial, M. (2005). Proceedings of the 1st International Computing Education Research Workshop, ICER 2005. *Unknown Journal*.

Austin, H. (1976). *Teaching Teachers LOGO: The Lesley Experiments*. Retrieved from http://hdl.handle.net/1721.1/6237

Barr, A., Beard, M., & Atkinson, R. C. (1976). The computer as a tutorial laboratory: the Stanford BIP project. *International Journal of Man-Machine Studies, 8*(5), 567-582. doi:https://doi.org/10.1016/S0020-7373(76)80021-1

Barron, D. W. (1977). *An Introduction to the Study of Programming Languages*. Cambridge: Cambridge University Press.

Bednarik, R., & Tukiainen, M. (2006). *An eye-tracking methodology for characterizing program comprehension processes*. Paper presented at the Proceedings of the 2006 symposium on Eye tracking research & applications (ETRA '06), San Diego, California.

Ben-Ari, M., Berglund, A., Booth, S., & Holmboe, C. (2004). What do we mean by theoretically sound research in computer science education? *ACM SIGCSE Bulletin*, 230-231. doi:http://dx.doi.org/10.1145/1026487.1008059

Berglund, A. (2002). *Learning computer systems in a distributed course: Problematizing content and context*. Paper presented at the EARLI, SIG 10.

Berglund, A., & Pears, A. (2003). *Students' Understanding of Computer Networks in an Internationally Distributed Course*. Paper presented at the The 3rd IEEE International Conference on Advanced Learning Technologies (ICALT'03), Athens, Greece.

Blackwell, A. F., Rode, J. A., & Toye, E. F. (2009). How do we program the home? Gender, attention investment, and the psychology of programming at home. *International Journal of Human-Computer Studies, 67*(4), 324-341. doi:https://doi.org/10.1016/j.ijhcs.2008.09.011

Booth, S. (1992). *Learning to Program: A phenomonographic perspective.* (PhD), University of Gothenburg.

Booth, S. (1993). A Study of Learning to Program From an Experiential Perspective. *Computers in Human Behavior, 9*(2-3), 185-202. doi:https://doi.org/10.1016/0747-5632(93)90006-E

Brna, P., du Boulay, B., & Pain, H. (Eds.). (1999). *Learning to Build and Comprehend Complex Information Structures: Prolog as a Case Study*. Stamford, Connecticut: Ablex Publishing Corporation.

Brooks, R. (1977). Towards a theory of the cognitive processes in computer programming. *International Journal of Man-Machine Studies, 9*(6), 737-751. doi:https://doi.org/10.1016/S0020-7373(77)80039-4

Brooks, R. (1978). *Using a behavioral theory of program comprehension in software engineering*. Paper presented at the ICSE '78 Proceedings of the 3rd international conference on Software engineering Atlanta, Georgia, USA.

Brooks, R. (1983). Towards a theory of the comprehension of computer programs. *International Journal Man-Machine Studies, 18*(6), 543-554. doi:https://doi.org/10.1016/S0020-7373(83)80031-5

Brown, M. H. (1987). *Algorithm animation.* Brown University.

Bryant, S., Romero, P., & du Boulay, B. (2008). Pair Programming and the Mysterious Role of the Navigator. *International Journal of Human-Computer Studies, 66*(7), 519-529.

Byrne, M. D., Catrambone, R., & Stasko, J. T. (1999). Evaluating animations as student aids in learning computer algorithms. *Comput. Educ., 33*(4), 253-278. doi:10.1016/s0360-1315(99)00023-8

Cannara, A. B. (1976). *Experiments In Teaching Children Computer Programming* (271). Retrieved from

Carver, S. M. (1986). *Transfer of LOGO Debugging Skill: Analysis, Instruction, and Assessment.* (PhD), Carnegie-Mellon University, Pittsburgh, USA.   (ERIC:ED284678)

Cheney, P. H. (1977). Teaching Computer Programming in an Environment Where Collaboration is Required. *Journal of the Association for Educational Data Systems, 11*(1), 1-5.

Clancy, M. J., & Linn, M. (1992). *Designing Pascal Solutions: A Case Study Approach*: W.H. Freeman & Company.

diSessa, A. (2001). *Changing Minds*: MIT Press.

diSessa , A. A. (1985). A principled design for an integrated computational environment. *Human-Computer Interaction, 1*(1), 1-47.

diSessa , A. A., & Abelson, H. (1986). Boxer: A reconstructible computational medium. *Communications of the ACM, 29*(9), 859-868.

diSessa , A. A., Hammer, D., Sherin, B. L., & Kolpakowski, T. (1991). Inventing graphing: Meta-representational expertise in children. *Journal of Mathematical Behavior, 2*(117-160).

Druin, A., Knell, G., Soloway, E., Russell, D., Mynatt, E., & Rogers, Y. (2011). The future of child-computer interaction *CHI EA '11: CHI '11 Extended Abstracts on Human Factors in Computing Systems* (pp. 693-696). Vancouver, BC, Canada: ACM.

du Boulay, B. (1986). Some difficulties of learning to program. *Journal of Educational Computing Research, 2*(1), 57-73.

du Boulay, B., & O'Shea, T. (1981). Teaching Novices Programming. In M. J. Coombs & J. L. Alty (Eds.), *Computing skills and the user interface* (pp. 147-200): Academic Press.

du Boulay, B., O'Shea, T., & Monk, J. (1981). The black box inside the glass box: presenting computing concepts to novices. *International Journal of Man-Machine Studies, 14*(3), 237-249.

du Boulay, B., & Sothcott, C. (1987). Computers teaching programming: an introductory survey of the field. In R. W. Lawler & M. Yazdani (Eds.), *Artificial Intelligence and Education* (Vol. 1, pp. 345-372). Norwood, New Jersey: Ablex.

du Boulay, J. B. H. (1978). *Learning Primary Mathematics through Computer Programming.* (PhD), University of Edinburgh.

du Boulay, J. B. H. (1986). Some Difficulties of Learning to Program. *Journal of Educational Computing Research, 2*, 57-73.

Eason, K. D. (1976). Understanding the naive computer user. *The Computer Journal, 19*(1), 3-7. doi:https://doi.org/10.1093/comjnl/19.1.3

Eisenstadt, M. (1979). A friendly software environment for psychology students. *AISB Quarterly*.

Eisenstadt, M., & Brayshaw, M. (1990). A fine-grained account of Prolog execution for teaching and debugging. *Instructional Science, 19*(4-5), 407-436. doi:https://doi-org.ezproxy.sussex.ac.uk/10.1007/BF00116447

Ensmenger, N. L. (2010). *The computer boys take over: Computers, programmers, and the politics of technical expertise*. Cambridge, MA: MIT Press.

Ericson, B. J., Guzdial, M. J., & Morrison, B. B. (2015). *Analysis of Interactive Features Designed to Enhance Learning in an Ebook*. Paper presented at the Proceedings of the eleventh annual International Conference on International Computing Education Research, Omaha, Nebraska, USA. http://delivery.acm.org/10.1145/2790000/2787731/p169-ericson.pdf?ip=128.61.71.130&id=2787731&acc=ACTIVE%20SERVICE&key=A79D83B43E50B5B8%2E5E2401E94B5C98E0%2E4D4702B0C3E38B35%2E4D4702B0C3E38B35&CFID=542853285&CFTOKEN=14252009&__acm__=1441306284_79ae35835ba068e72b3a9a581ce4a31e

Feurzeig, W., Papert, S., Bloom, M., Grant, R., & Solomon, C. (1969). *Programming-Languages as a Conceptual Framework for Teaching Mathematics. Final Report on the First Fifteen Months of the LOGO Project*. Retrieved from https://eric.ed.gov/?id=ED038034

Fincher, S., Lister, R., Clear, T., Robins, A., Tenenberg, J., & Petre, M. (2005). Multi-institutional, multi-national studies in CSEd Research: some design considerations and trade-offs *ICER '05: Proceedings of the first international workshop on Computing education research* (pp. 111-121). Seattle, WA, USA: ACM.

Fincher, S., & Tenenberg, J. (2006). Using Theory to Inform Capacity-Building: Bootstrapping Communities of Practice in Computer Science Education Research. *Journal of Engineering Education, 95*(4), 265-277.

Friend, J. (1975). *Programs Students Write*. Retrieved from https://eric.ed.gov/?id=ED112861

Gannon, J. D. (1978). *Characteristic errors in programming languages*. Paper presented at the ACM '78 Proceedings of the 1978 annual conference.

Garner, S., Haden, P., & Robins, A. (2005). *My program is correct but it doesn't run: a preliminary investigation of novice programmers' problems*. Paper presented at the Proceedings of the 7th Australasian conference on Computing education - Volume 42, Newcastle, New South Wales, Australia.

Gegg-Harrison, T. S. (1991). Learning Prolog in a schema-based environment. *Instructional Science, 20*(2-3), 173-192. doi:https://doi.org/10.1007/BF00120881

Goldberg, A., & Kay, A. (1976). *Smalltalk-72: Instruction Manual*: Xerox Corporation.

Goldberg, A., & Robson, D. (1983). *Smalltalk-80: the language and its implementation*: Addison-Wesley Longman Publishing Co., Inc.

Good, J. (2011). Learners at the Wheel: Novice Programming Environments Come of Age. *International Journal of People-Oriented Programming (IJPOP), 1*(1), 1-24. doi:http://dx.doi.org/10.4018/ijpop.2011010101

Gould, J. D. (1975). Some Psychological Evidence On How People Debug Computer Programs. *International Journal Man-Machine Studies, 7*(2), 171-182. doi:https://doi.org/10.1016/S0020-7373(75)80005-8

Gould, J. D., & Drongowski, P. (1974). An Exploratory Study of Computer Program Debugging. *Human Factors, 16*(3), 258-277. doi:https://doi.org/10.1177/001872087401600308

Green, T. R. G., & Petre, M. (1992). When visual programs are harder to read than textual programs. In G. C. v. d. Veer, M. J. Tauber, S. Bagnarola, & M. Antavolits (Eds.), *Human-Computer Itneraction: Tasks and Organisation, Proceedings EECE-6 (6th European Conference on Cognitive Ergonomics)*.

Green, T. R. G., & Petre, M. (1996). Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages and Computing, 7*(2), 131-174.

Green, T. R. G., & Petre, M. (1996). Usability Analysis of Visual Programming Environments: A ‘Cognitive Dimensions’ Framework. *Journal of Visual Languages & Computing, 7*(2), 131-174. doi:https://doi.org/10.1006/jvlc.1996.0009

Green, T. R. G., Petre, M., & Bellamy, R. K. E. (1991). Comprehensibility of visual and textual programs: a test of 'superlativism' against the 'match-mismatch' conjecture. In J. Koenemann-Belliveau, T. Moher, & S. Robertson (Eds.), *Empirical Studies of Programmers: Fourth Workshop* (pp. 121-146). Norwood, NJ: Ablex.

Greenberger, M. (1962). *Computers and the World of the Future*: MIT Press.

Guzdial, M. (2015). *Learner-Centered Design of Computing Education: Research on Computing for Everyone*: Morgan & Claypool Publishers.

Habermann, A. N. (1973). Critical comments on the programming language Pascal. *Acta Informatica, 3*(1), 47-57. doi:https://doi.org/10.1007/BF00288652

Hasemer, T. (1983). *An Empirically-Based Debugging System for Novice Programmers* (Technical Report No. 6). Retrieved from

Hoc, J.-M. (1977). Role of mental representation in learning a programming language. *International Journal Man-Machine Studies, 9*(1), 87-105. doi:https://doi.org/10.1016/S0020-7373(77)80044-8

Hoc, J.-M., Green, T. R. G., Samurcay, R., & Gilmore, D. J. (1990). *Psychology of Programming*: European Association of Cognitive Ergonomics and Academic Press.

Howland, K., Good, J., & du Boulay, B. (2013). Narrative Threads: A Tool to Support Young People in Creating Their Own Narrative-Based Computer Games. In Z. Pan, A. D. Cheok, W. Müller, I. Iurgel, P. Petta, & B. Urban (Eds.), *Transactions on Edutainment X* (pp. 122-145). Berlin, Heidelberg: Springer Berlin Heidelberg.

Hundhausen, C. D., & Brown, J. L. (2007). An experimental study of the impact of visual semantic feedback on novice programming. *J. Vis. Lang. Comput., 18*(6), 537-559. doi:10.1016/j.jvlc.2006.09.001

Hundhausen, C. D., Douglas, S. H., & Stasko, J. T. (2002). A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing, 13*, 259-290.

Hundhausen, C. D., Farley, S., & Brown, J. L. (2006). *Can Direct Manipulation Lower the Barriers to Programming and Promote Positive Transfer to Textual Programming? An Experimental Study*. Paper presented at the Proceedings of the Visual Languages and Human-Centric Computing.

Hundhausen, C. D., Olivares, D. M., & Carter, A. S. (2017). IDE-Based Learning Analytics for Computing Education: A Process Model, Critical Review, and Research Agenda. *Trans. Comput. Educ., 17*(3), 1-26. doi:10.1145/3105759

Johnson, W. L., & Soloway, E. (1987). Proust: an automatic debugger for Pascal programs. In G. P. Kearsley (Ed.), *Artificial Intelligence \& Instruction: applications and methods*. Reading, Massachusetts: Addison-Wesley Publishing.

Johnson, W. L., Soloway, E., Cutler, B., & Draper, S. (1983). *Bug Catalogue 1* (286). Retrieved from

Johnson-Laird, P. N. (1983). *Mental Models: Towards a Cognitive Sceince of Language, Inference and Consciousness*. Cambridge, Massachusetts: Harbard University Press.

Joni, S.-N. A., & Soloway, E. (1986). But My Program Runs! Discourse Rules for Novice Programmers. *Journal of Educational Computing Research, 2*(1), 95-125. doi:10.2190/6e5w-ar7c-nx76-hut2

Kahney, H. (1982). *An In-Depth Study of the Cognitive Behaviour of Novice Programmers* (Technical Report No. 5). Retrieved from

Katz, I. R., & Anderson, J. R. (1987). Debugging: an analysis of bug-location strategies. *Human-Computer Interaction, 3*(4), 351-399. doi:http://dx.doi.org/10.1207/s15327051hci0304_2

Kay, A., & Goldberg, A. (1977). Personal dynamic media. *IEEE Computer*, 31-41.

Kay, A. C. (1972). *A Personal Computer for Children of All Ages*. Paper presented at the Proceedings of the ACM annual conference - Volume 1, Boston, Massachusetts, USA.

Kay, A. C. (1993). The Early History of Smalltalk *The Second ACM SIGPLAN Conference on History of Programming Languages* (pp. 69-95). Cambridge, Massachusetts, USA: ACM.

Kehoe, C., Stasko, J., & Taylor, A. (2001). Rethinking the evaluation of algorithm animations as learning aids. *Int. J. Hum.-Comput. Stud., 54*(2), 265-284. doi:10.1006/ijhc.2000.0409

Kurland, D. M., & Pea, R. D. (1985). Children's Mental Models Of Recursive Logo Programs. *Journal of Educational Computing Research, 1*(2), 235-243. doi:http://dx.doi.org/10.2190/JV9Y-5PD0-MX22-9J4Y

Kurland, D. M., Pea, R. D., Clement, C., & Mawby, R. (1986). A Study Of The Development Of Programming Ability And Thinking Skills In High School Students. *Journal of*

Educational Computing Research, 2(4), 429-458. doi:http://dx.doi.org/10.2190/BKML-B1QV-KDN4-8ULH

Ladner, R. E., & Israel, M. (2016). "For all" in "computer science for all". *Commun. ACM, 59*(9), 26-28. doi:10.1145/2971329

Larkin, J., McDermott, J., Simon, D.P. and Simon, H.A., 1980. Expert and novice performance in solving physics problems. *Science*, *208*(4450), pp.1335-1342.

Lecarme, O., & Desjardins, P. (1975). More comments on the programming language Pascal. *Acta Informatica, 4*(3), 231-243. doi:https://doi.org/10.1007/BF00288728

Lemos, R. S. (1975). FORTRAN Programming: an analysis of pedagogical alternatives. *Journal of Educational Data Processing, 12*(3), 21-29.

Lemos, R. S. (1978). Students' attitudes towards programming: The effects of structured walk-throughs. *Computers & Education, 2*(4), 301-306. doi:https://doi.org/10.1016/0360-1315(78)90005-2

Lemos, R. S. (1979). Teaching programming languages: A survey of approaches. *ACM SIGCSE Bulletin - Proceedings of the 10th SIGCSE symposium on Computer science, 11*(1), 174-181. doi:https://doi.org/10.1145/953030.809578

Lister, R. (2011). Ten Years After the McCracken Working Group. *ACM Inroads, 2*(4), 18-19.

Lister, R., Box, I., Morrison, B., Tenenberg, J., & Westbrook, D. S. (2004). The Dimensions of Variation in the Teaching of Data Structures. *SIGCSE Bull., 36*(3), 92--96. doi:10.1145/1026487.1008023

Love, T. (1977). *An experimental investigation of the effect of program structure on program understanding*. Paper presented at the Proceedings of an ACM conference on Language design for reliable software, Raleigh, North Carolina.

Lukey, F. J. (1980). Understanding and Debugging Programs. *International Journal of Man-Machine Studies, 12*(2), 189-202. doi:https://doi.org/10.1016/S0020-7373(80)80017-4

Maloney, J., Burd, L., Kafai, Y., Rusk, N., Silverman, B., & Resnick, M. (2004). Scratch: A Sneak Preview *C5 '04: Proceedings of the Second International Conference on Creating, Connecting and Collaborating through Computing* (pp. 104-109). Washington, DC, USA: IEEE Computer Society.

Maloney, J. H., Peppler, K., Kafai, Y., Resnick, M., & Rusk, N. (2008). *Programming by choice: urban youth learning programming with scratch*. Paper presented at the SIGCSE '08: Proceedings of the 39th SIGCSE technical symposium on Computer science education, New York, NY, USA. http://doi.acm.org/10.1145/1352135.1352260

Margolis, J., & Fisher, A. (2002). *Unlocking the Clubhouse: Women in Computing*: MIT Press.

Marton, F., 1981. Phenomenography—describing conceptions of the world around us. *Instructional science*, *10*(2), pp.177-200.

Mateas, M. (2008). Procedural literacy: educating the new media practitioner. In D. Drew (Ed.), *Beyond Fun* (pp. 67-83): ETC Press.

Mayer, R. E. (1975). Different problem-solving competencies established in learning computer programming with and without meaningful models. *Journal of educational psychology, 67*(6), 725-734. doi:http://dx.doi.org/10.1037/0022-0663.67.6.725

Mayer, R. E. (1976). Comprehension as Affected by Structure of Problem Representation. *Memory and Cognition, 4*(3), 249-255. doi:https://doi.org/10.3758/BF03213171

Mayer, R. E. (1979). A psychology of learning BASIC. *Communications of the ACM, 22*(11), 589-593. doi:https://doi.org/10.1145/359168.359171

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B.-D., . . . Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin, 33*(4), 125-140.

McLuhan, M. H. (1962). *The Gutenberg Galaxy: The Making of Typographic Man*: University of Toronto Press.

McLuhan, M. H. (1964). *Understanding Media: The Extensions of Man*. Cambridge, MA: MIT Press.

McNerney, T. S. (2004). From turtles to Tangible Programming Bricks: explorations in physical language design. *Personal and Ubiquitous Computing, 8*(5), 326-337. doi:10.1007/s00779-004-0295-6

Miller, L. A. (1974). Programming by non-programmers. *International Journal of Man-Machine Studies, 6*(2), 237-260. doi:https://doi.org/10.1016/S0020-7373(74)80004-0

Miller, L. A. (1975). *Naive programmer problems with specification of transfer-of-control*. Paper presented at the AFIPS '75 Proceedings national computer conference, Anaheim, California.

Moreno-León, J., Robles, G., & Román-González, M. (2017). Towards Data-Driven Learning Paths to Develop Computational Thinking with Scratch. *IEEE Transactions on Emerging Topics in Computing, PP*(99), 1-1. doi:http://dx.doi.org/10.1109/TETC.2017.2734818

Morrison, B. B., Dorn, B., & Guzdial, M. (2014). *Measuring cognitive load in introductory CS: adaptation of an instrument*. Paper presented at the Proceedings of the tenth annual conference on International computing education research, Glasgow, Scotland, United Kingdom. http://delivery.acm.org/10.1145/2640000/2632348/p131-morrison.pdf?ip=143.215.63.175&id=2632348&acc=ACTIVE%20SERVICE&key=A79D83B43E50B5B8%2E5E2401E94B5C98E0%2E4D4702B0C3E38B35%2E4D4702B0C3E38B35&CFID=463892605&CFTOKEN=52470917&__acm__=1418761875_da92130665614ac939c9f0d8fbbd41e9

Murphy, L., Lewandowski, G., Ren, #233, McCauley, e., Simon, B., . . . Zander, C. (2008). *Debugging: the good, the bad, and the quirky -- a qualitative analysis of novices' strategies*. Paper presented at the Proceedings of the 39th SIGCSE technical symposium on Computer science education, Portland, OR, USA.

Newell, A., & Simon, H. A. (1972). *Human problem solving*: Prentice-Hall.

Ni, L., Tew, A. E., Guzdial, M. J., & McKlin, T. (2011). *A regional professional development program for computing teachers: The disciplinary commons for computing educators*. Paper presented at the 2011 Annual Meeting of the American Educational Research Association, New Orleans, LA.

Nievergelt, J., Frei, H. P., Burkhart, H., Jacobi, C., Pattner, B., Sugaya, H., . . . Weydert, J. (1978). XS-0: a self-explanatory school computer. *ACM SIGCSE Bulletin, 10*(4), 66-69. doi:https://doi.org/10.1145/988906.988921

Palumbo, D. J. (1990). Programming Language/Problem-Solving Research: A Review of Relevant Issues. *Review of Educational Research, 60*(1), 65-89. doi:https://doi-org.ezproxy.sussex.ac.uk/10.3102/00346543060001065

Palumbo, D. J., & Reed, M. W. (1991). The Effect Of Basic Programming Language Instruction On High School Students' Problem Solving And Computer Anxiety. *Journal of Research on Computing in Education, 23*(3), 343-372. doi:http://dx.doi.org/10.1080/08886504.1991.10781967

Papert, S. (1971). *Teaching children to be mathematicians versus teaching about mathematics*. Retrieved from

Papert, S. (1972). Teaching Children to be Mathematicians Versus Teaching About Mathematics. *International Journal of Mathematical Education in Science and Technology, 3*(3), 249-262. doi:http://dx.doi.org/10.1080/0020739700030306

Papert, S. (1980a). *Mindstorms: Children, Computers and Powerful Ideas*. Brighton, Sussex: Harvester Press.

Papert, S. (1980b). *Mindstorms: Children, computers, and powerful ideas*: Basic Books.

Papert, S. (1987). Information Technology and Education: Computer Criticism vs. Technocentric Thinking. *Educational Researcher, 16*(1), 22-30. doi:10.3102/0013189x016001022

Papert, S. A., & Solomon, C. (1971). *Twenty Things to Do with A Computer*. Retrieved from Cambridge, MA:

Pea, R. D. (1986). Language-Independent Conceptual ``bugs" in novice programming. *Journal of Educational Computing Research, 1*(1986).

Pea, R. D. (1987). The Aims of Software Criticism: Reply to Professor Papert. *Educational Researcher, 16*(5), 4-8. doi:10.3102/0013189x016005004

Pea, R. D., & Kurland, D. M. (1984). On the cognitive effects of learning computer programming. *New Ideas in Psychology, 2*(2), 137-168. doi:https://doi.org/10.1016/0732-118X(84)90018-7

Pea, R. D., Kurland, D. M., & Hawkins, J. (1985). LOGO and the Development of Thinking Skills. In M. Chen & W. Paisley (Eds.), *Children and Microcomputers: Research on the Newest Medium* (pp. 193-317): Sage.

Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., . . . Paterson, J. (2007). A Survey of Literature on the Teaching of Introductory Programming. *ACM SIGCSE Bulletin, 39*(4), 204-223. doi:http://dx.doi.org/10.1145/1345375.1345441

Petre, M. (1995). Why looking isn't always seeing: readership skills and graphical programming. *Commun. ACM, 38*(6), 33-44. doi:10.1145/203241.203251

Pugh, J., & Simpson, D. (1979). Pascal errors - empirical evidence. *Computer Bulletin, 2*, 26-28.

Resnick, M. (1990). Multilogo: A study of children and concurrent programming. *Interactive Learning Environments, 1*(3), 153-170.

Resnick, M., Martin, F., Sargent, R., & Silverman, B. (1996). Programmable bricks: toys to think with. *IBM Syst. J., 35*(3-4), 443-452.

Ripley, G. D., & Druseikis, F. C. (1978). A statistical analysis of syntax errors. *Computer Languages, 3*(4), 227-240. doi:https://doi.org/10.1016/0096-0551(78)90041-3

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and Teaching Programming: A Review and Discussion. *Computer Science Education, 13*(2), 137-172. doi:http://dx.doi.org/10.1076/csed.13.2.137.14200

Roman-Gonzalez, M., Perez-Gonzalez, J.-C., & Jimenez-Fernandez, C. (2016). Which cognitive abilities underlie computational thinking? Criterion validity of the Computational Thinking Test. *Computers in Human Behavior, 72*, 678-691. doi:https://doi.org/10.1016/j.chb.2016.08.047

Romero, P., du Boulay, B., Robertson, J., Good, J., & Howland, K. (2009). Is Embodied Interaction Beneficial When Learning Programming? *VMR '09 Proceedings of the 3rd International Conference on Virtual and Mixed Reality: Held as Part of HCI International 2009* (pp. 97-105). Berlin: Springer-Verlag.

Sackman, H. (1968). Conference on Personnel Research. *Datamation, 14*(7), 74-76.

Shapiro, S. C., & Witmer, D. P. (1974). Interactive visual simulators for beginning programming students. *ACM SIGCSE Bulletin - Proceedings of the 4th SIGCSE symposium on Computer science education, 6*(1), 11-14. doi:https://doi.org/10.1145/953057.810431

Sherin, B. L. (2001). A comparison of programming languages and algebraic notation as expressive langauges for physics. *International Journal of Computers for Mathematical Learning, 6*, 1-61.

Sherman, L., Druin, A., Montemayor, J., Farber, A., Platner, M., Simms, S., . . . Lal, A. (2001). *StoryKit: tools for children to build room-sized interactive experiences*. Paper presented at the CHI '01 Extended Abstracts on Human Factors in Computing Systems, Seattle, Washington.

Shneiderman, B. (1977). Measuring computer program quality and comprehension. *International Journal Man-Machine Studies, 9*(4), 465-478. doi:https://doi.org/10.1016/S0020-7373(77)80014-X

Shneiderman, B. (1977). Teaching Programming: a spiral approach to syntax and semantics. *Computers and Education, 1*(4), 193-197. doi:https://doi.org/10.1016/0360-1315(77)90008-2

Shneiderman, B., Mayer, R. E., McKay, D., & Heller, P. (1977). Experimental investigations of the utility of detailed flowcharts in programming. *Communications of the ACM, 20*(6), 373-381. doi:http://dx.doi.org/10.1145/359605.359610

Sime, M. E., Arblaster, A. T., & Green, T. R. G. (1977a). Reducing programming errors in nested conditionals by prescribing a writing procedure. *International Journal Man-Machine Studies, 9*(1), 119-126. doi:https://doi.org/10.1016/S0020-7373(77)80046-1

Sime, M. E., Arblaster, A. T., & Green, T. R. G. (1977b). Structuring the Programmer's Task. *Journal of Occupational Psychology, 50*(3), 205-216. doi:http://dx.doi.org/10.1111/j.2044-8325.1977.tb00376.x

Sime, M. E., Green, T. R. G., & Guest, D. J. (1977). Scope marking in computer conditionals—a psychological evaluation. *International Journal Man-Machine Studies, 9*(1), 107-118. doi:https://doi.org/10.1016/S0020-7373(77)80045-X

Sleeman, D. (1986). The challenges of teaching computer programming. *Commun. ACM, 29*(9), 840-841. doi:10.1145/6592.214913

Sleeman, D., Putnam, R. T., Baxter, J., & Kuspa, L. (1986). Pascal and High School Students: A Study of Errors. *2*(1), 5-23. doi:https://doi.org/10.2190/2XPP-LTYH-98NQ-BU77

Smith, D. C. (1975). *PYGMALION: A creative programming environment.* (PhD), Stanford University, Stanford, CA.

Soloway, E., & Ehrlich, K. (1984). Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering, SE-10:5*(5), 595-609. doi:http://dx.doi.org/10.1109/TSE.1984.5010283

Soloway, E., & Iyengar, S. (Eds.). (1986). *Emprical Studies of Programmers: Papers presented at the first Workshop on Empirical Studies of Programmers*. Norwood, New Jersey, USA: Ablex.

Sorva, J. (2012). *Visual Program Simulation in Introductory Programming Education.* (Doctor of Science in Technology), Aalto University School of Science.

Sorva, J. (2013). Notional Machines and Introductory Programming Education. *Trans. Comput. Educ., 13*(2), 8:1-8:31.

Sorva, J., Sirki\, T., & \#228. (2010). *UUhistle: a software tool for visual program simulation*. Paper presented at the Proceedings of the 10th Koli Calling International Conference on Computing Education Research, Berlin, Germany.

Spohrer, J. C., Pope, E., Lipman, M., Sack, W., Freiman, S., Littman, D., . . . Soloway, E. (1985). *Bug Catalogue: II, III, IV* (386). Retrieved from

Stasko, J. T. (1997a). *Supporting student-built algorithm animation as a pedagogical tool*. Paper presented at the CHI '97 Extended Abstracts on Human Factors in Computing Systems, Atlanta, Georgia.

Stasko, J. T. (1997b). *Using student-built algorithm animations as learning aids*. Paper presented at the Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education, San Jose, California, USA.

Statz, J. A. (1973). *The Development Of Computer Programming Concepts And Problem-Solving Abilities Among Ten-Year-Olds Learning Logo.* (PhD), Syracuse University.

Tagg, W. (1974). Programming languages for school use. *Computer Education, 16*, 11-22.

Taylor, J., & du Boulay, B. (1987). Why novices may find programming in Prolog hard. In J. C. Rutkowska & C. Crook (Eds.), *Computers, Cognition and Development: Issues for Psychology and Education*. Chichester, Sussex: John Wiley \& Sons.

Utting, I., Tew, A. E., McCracken, M., Thomas, L., Bouvier, D., Frye, R., . . . Wilusz, T. (2013). A Fresh Look at Novice Programmers' Performance and Their Teachers' Expectations *Proceedings of the ITiCSE Working Group Reports Conference on Innovation and Technology in Computer Science Education-working Group Reports* (pp. 15-32). Canterbury, England, United Kingdom: ACM.

Weinberg, G. M. (1971). *The Psychology of Computer Programming*. New York: Van Nostrand / Reinhold.

Weintrop, D., & Wilensky, U. (2015). *Using Commutative Assessments to Compare Conceptual Understanding in Blocks-based and Text-based Programs*. Paper presented at the Proceedings of the eleventh annual International Conference on International Computing Education Research, Omaha, Nebraska, USA.

Welsh, J., Sneeringer, W. J., & Hoare, C. A. R. (1977). Ambiguities and insecurities in pascal. *Software: Practice and Experience*. doi:http://dx.doi.org/10.1002/spe.4380070604

Weyer, S. A., & Cannara, A. B. (1975). *Children Learning Computer Programming: Experiments with Languages, Curricula and Programmable Devices* (250). Retrieved from

Wiedenbeck, S. (1986). Beacons in computer program comprehension. *International Journal of Man-Machine Studies, 25*(6), 697-709. doi:http://dx.doi.org/10.1016/S0020-7373(86)80083-9

Wing, J. (2006). Computational Thinking. *Communications of the ACM, 49*(3), 33-35. doi:http://dx.doi.org/10.1145/1118178.1118215

Wolf, M. (2007). *Proust and the Squid: The Story and Science of the Reading Brain*: Harper Colins.

Youngs, E. A. (1974). Human errors in programming. *International Journal of Man-Machine Studies, 6*(3), 361-376. doi:https://doi.org/10.1016/S0020-7373(74)80027-1