2016

The Long Quest for Computational Thinking

Tedre, Matti; Denning, Peter J.

ACM

Tedre, Matti, and Peter J. Denning. "The long quest for computational thinking." Proceedings of the 16th Koli Calling International Conference on Computing Education Research. ACM, 2016.

https://hdl.handle.net/10945/61006

Tedre, Matti; Denning, Peter J. (2016) The Long Quest for Computational Thinking. *Proceedings of the 16th Koli Calling Conference on Computing Education Research*, November 24–27, 2016, Koli, Finland: pp. 120–129.

The Long Quest for Computational Thinking

Matti Tedre
Stockholm University, DSV
Kista, Sweden
matti.tedre@dsv.su.se

Peter J. Denning
Naval Postgraduate School
Monterey, CA, USA
pjd@nps.edu

ABSTRACT

Computational thinking (CT) is a popular phrase that refers to a collection of computational ideas and habits of mind that people in computing disciplines acquire through their work in designing programs, software, simulations, and computations performed by machinery. Recently a computational thinking for K–12 movement has spawned initiatives across the education sector, and educational reforms are under way in many countries. However, modern CT initiatives should be well aware of the broad and deep history of computational thinking, or risk repeating already refuted claims, past mistakes, and already solved problems, or losing some of the richest and most ambitious ideas in CT. This paper presents an overview of three important historical currents from which CT has developed: evolution of computing's disciplinary ways of thinking and practicing, educational research and efforts in computing, and emergence of computational science and digitalization of society. The paper examines a number of threats to CT initiatives: lack of ambition, dogmatism, knowing versus doing, exaggerated claims, narrow views of computing, overemphasis on formulation, and lost sight of computational models.

CCS Concepts

•Social and professional topics → History of computing; Computational thinking;

Keywords

Computer science education; Computational thinking; CSER; Computational ideas; History of computational thinking; Disciplinary ways of thinking and practicing

1. INTRODUCTION

Computational thinking (CT) has become the subject of worldwide attention in recent years as part of multiple efforts to bring computer science into all K–12 schools. The leaders of the movement have proposed definitions, a body of knowledge, and assessment methods for CT [1, 2]. They have some impressive achievements, including the training of 10,000 CS teachers in the US, a new Advanced Placement (AP) program, and a set of "CS principles" courses at universities that receive graduates of the new AP program. CSTA (computer science teachers' association), code.org, and K12CS.org have been working on detailed curriculum guidelines and obtaining political support at the US federal and state level to push computer science into all K–12 schools. Similar initiatives have quickly emerged in other countries, too [3, 4].

The current movement began in 2006 with an essay by Jeannette Wing [5]. Her idea was that everyone would benefit from learning to think like a computer scientist [6]. However, the movement has been criticized for vagueness, ambiguous definitions and visions of CT, and arrogance [4], as well as for bold, unsubstantiated claims about the universal benefit of CT [3]. The computing education community has found it hard to find a consensus on definition of CT [4]. The multiple CT visions, although inspiring and ambitious, do not agree on what exactly should be taught about CT, how to assess whether students have learned CT, and who are the main beneficiaries of CT.

Our purpose here is to deepen and broaden discussions about CT by taking a careful look at the long and rich history of CT as well as its ambitious visions over the years. Aho's definition of CT stands out for its exceptional clarity: Computational thinking is the "thought processes involved in formulating problems so their solutions can be represented as computational steps and algorithms" [7]. Aho was reflecting an idea about thought processes in computing that had been in play since the 1950s. Many early pioneers regarded computational, algorithmic, or procedural thinking as an important skill set for those who design and implement computations. The new CT movement aimed to include also those who use computational tools and those who engage in step-by-step procedures [5]. The attempt to broaden the CT audience moved into uncharted territory, where there is less certainty that tool users and procedure followers need CT or benefit from it.

This survey describes the historical development of CT and the intellectual ideas that drove its development. In the process we will examine various claims about CT and will conclude some of them are unsubstantiated.

We see three reasons to review CT and its claims from a historical perspective. Firstly, understanding the long history of CT is a matter of academic rigor, "standing on the shoulders of giants." When researchers do their homework

Koli Calling 2016, November 24 - 27, 2016, Koli, Finland

ISBN 978-1-4503-4770-9/16/11…$15.00

DOI: http://dx.doi.org/10.1145/2999541.2999542

well, they know what previous generations of scientists have tried and done, and where they have succeeded and failed. They avoid "reinventing the wheel" by acknowledging predecessors who built the foundations on which the current generation of researchers is now working. In this paper we will suggest that a lack of knowledge about CT's long and rich history may lead to weaker and less ambitious versions of CT than we have seen in the past, causing CT to diminish, not progress.

Secondly, strong unsubstantiated claims create expectations about CT that cannot be fulfilled. We will see that some bold claims of CT have repeatedly been made and refuted over several decades. An example is the claim of automatic skill transfer from CT to different knowledge domains, which was debunked in the 1980s [8] but which was repeated so persistently that even a recent 2015 book had to explicitly repeat the extensive critique of that claim about CT [3, pp. 27–29, 39–40]. Exaggerated claims about CT serve no one—for eventually, when CT cannot deliver on such claims, there will be many disillusioned educators and consumers of education, and computer science will be seen as an over-seller of CT. CT is powerful enough without exaggerated claims.

Thirdly, the direct beneficiaries of CT—engineering, science, and design—are deeply historical. In contrast, the behavior of machines and information transforming processes studied in the computational sciences and engineering are by nature context free: the machine is supposed to behave in a certain well defined way irrespective of where or when it is used. This clean, abstract view of machines and processes is attractive to mathematicians and many scientists. But the full picture is not so tidy. Engineers and designers are always working to find better ways to harness effects in an uncertain world. They are explorers and builders, often doing their work before the science is available to support them [9]. They are keenly interested in what works and what does not. They must know their history to do their work.

2. DISCIPLINARY WAYS OF THINKING AND PRACTICING

Early descriptions of computational thinking arose at a time of great uncertainty about the nascent field of computing, and especially about how it differs from other, more established fields. In the early descriptions of computing's identity, Fein [10] characterized computing as an interdisciplinary field and Gorn [11] as a study of mechanical languages and their processors. Zadeh [12] insisted computing is more an engineering field than a mathematical field. Hamming [13, 14] distinguished computing from mathematics by its practical relevance and emphasis on the real world. Forsythe [15] thought design was central to computing. Hammer [16] emphasized the dramatic effects of computing on society and the human condition. When Newell, Perlis, and Simon published their famous 1967 defense of computer science [17], computing had already started to claim its rightful place in the academia, but a shared understanding of computing's intellectual identity was still missing. In this process of academic soul-seeking while honoring practical needs, descriptions of computing's unique disciplinary ways of thinking and practicing started to emerge.

2.1 Clarifying Computing's Unique Ways of Thinking and Practicing

In the late 1950s Alan Perlis was among the first computing pioneers to highlight the value of coding as a mental tool for understanding all kinds of problems [18]. In 1960 he argued that the value of computers is less about their use as instruments and more about their cultivating a certain style of reasoning about problems and designing solutions [19]. He gave the name *algorithmizing* to computing's quantitative analysis of the way one *does* things, and argued that it had become so ingrained in our culture that everyone should sooner or later learn it.

In the 1970s computing educators worked to justify computing as a unique field separate from mathematics yet still rigorous enough to warrant a place in the traditional research universities, which shunned technological subjects in preference to theoretically oriented subjects [20]. Over the years many people, including pioneers like Dijkstra [21] and Knuth [22], reaffirmed the idea that computing's disciplinary identity arises from its unique mental processes.

Dijkstra believed that the uniqueness of computing comes from *algorithmic thinking* that was characterized by 1) mastery of natural language in order to bridge the gap between informally expressed problems and formal solutions, 2) ability to invent one's own formalisms and concepts when solving problems, and 3) agility to switch back and forth between semantic levels—a sort of "mental zoom lens" [21].

Knuth searched for the essence of algorithmic thinking in mathematics texts that discussed "types of thinking" and contrasted those with reasoning patterns used by computer scientists [23]. He found that representation of reality, reduction to simpler problems, abstract reasoning, information structures, and attention to algorithms were common in algorithmic thinking but uncommon in mathematics. He also found two thinking patterns used by computer scientists that were not employed by mathematicians: complexity and causality—considering the complexity or economy of processing, and designing imperative procedures that create action in the world.

Others also emphasized the importance of learning algorithmic thinking [24]. Some even wished to name the field "algorithmics" [23, 25]. By the late 1970s, the idea that algorithms are the central subject matter of computer science, and programming or designing algorithms are the central practice, became very popular. But the computing field was populated with people with different interests ranging from programming and algorithms to systems and experimentation. Those divisions were not just rhetorical, they manifested in very tangible ways. For instance, in the late 1970s there was a significant "brain drain" from universities to industry of academics who could design and build computing systems. In the US the problem was severe enough that NSF funded a study published in 1979 and known as the Feldman report [26]. That report concluded that the brain drain was real and that NSF could help reverse the deterioration of CS departments by creating new programs specifically for systems experimentation. In 1980, the department chairs of most CS departments in the US drafted their own report on the problem and on what it would cost to equip the experimental labs of a CS department [27]. The ACM soon signed on and implored the NSF to help [28].

The NSF responded by funding the CSNET project (to transfer Internet technology from the ARPANET into a net-

work for CS research) in 1981 and soon thereafter the Coordinated Experimental Research (CER) program to support experimental systems research. By the mid 1980s, many people acknowledged that computer science has a broader mandate than designing and analyzing algorithms—it also builds systems and networks that serve as platforms and infrastructure to execute algorithms. This broader mandate was further refined in 1989 by another report "Computing as a Discipline" from ACM and IEEE, which argued that computing was a field encompassing theory, abstraction, and design—everything computational from algorithms to architectures, design, and networks [29].

The view of computing's disciplinary ways of thinking and practicing broadened over time. While some early pioneers had claimed algorithmic thinking was the core of the field, later pioneers claimed that computation was the core—systems, architectures, and design were essential but did not completely fall within "algorithmic thinking". That older debate about algorithms versus systems has resurrected in the CT claims today. Some modern CT initiatives emphasize the programming and algorithm side almost to the point of risking the exclusion of the experimental and system side of computer science.

2.2 General-Purpose Thinking Tools

The debate about algorithmic thinking was not limited to whether algorithms or systems characterized the field. From the earliest days it included claims that algorithmic thinking would train the brain to be a better problem solver in all fields [19]. In 1968, Forsythe [30] argued that computing's unique ways of thinking provide general-purpose mental tools which remain serviceable for a lifetime. Decade by decade the claims became increasingly ambitious. In 1970 Minsky made the claim that programming would gradually become more important than mathematics for early education [31]. In 1984 Bolter [32] argued that computing is the defining technology of the current era, and similar to "the classical man" and "the modern man" he described the computational image of the human, "Turing's man," as the quintessential image of humanity in the computer age. In 1996 Abelson and Sussman argued that computing's procedural epistemology revolutionizes the way people think and express what they think [33].

The phrase "general-purpose mental tool," already mentioned by Perlis in 1960 [19], appeared frequently in characterizations of computational thinking. For example, in 1974 Knuth [22] argued that thinking through algorithms is a useful aid in fields from chemistry to linguistics and music. He referred to the old adage "a person does not really understand something until he/she teaches it to someone else". In line with the idea of "programming to learn" instead of "learning to program" [34], Knuth repeated the idea that dates back to the 1950s [18]: that teaching something to a dumb computer—that is, expressing a process as an algorithm and a program—forces precision and leads to much deeper understanding than any traditional means of thinking does [22].

Many people advocated the view that learning programming, or procedural thinking, leads to other, related higher-order cognitive skills: you get several birds with one stone. In 1970 Minsky argued that the concept of procedure was "the secret educators have so long been seeking" [31]. Knuth wrote that his experiences have convinced him of "the pedagogic value of algorithmic approach; it aids in the understanding of concepts of all kinds" [22]. Feurzeig et al. argued that teaching programming also improves logical and rigorous thinking in general [35]. Kugel asked whether the role of computing might be like the role of logic in the Middle Ages, where it was supposed to "sharpen the mind" [36]. Some CT ideas captivated educators across different fields, and the early decades of many computing magazines and computer science education publications were replete with descriptions of computing programs in liberal arts colleges.

3. CT FOR K–12

Translating the high ideas about computing's general-purpose thinking tools into courses in K–12 schools was a major challenge from the beginning. In the 1980s few schools had a computer course of any kind and most lacked teachers with computer science knowledge. For these reasons, computer literacy was seen by many as the first step toward getting programming into grade-school education. Yet "literacy" in computing terms was seen in many competing ways. Literacy in programming was called a "modern survival skill" [37]. Other common descriptions were procedural literacy [37], computational literacy [38], literacy in algorithmic reasoning [39, p.112], "the second literacy" [40], proceduracy [41] and "the fourth literacy" (e.g., [42]). Knuth's book *Literate Programming* [43] viewed programming as a medium of logical thought. Recently Annette Vee reviewed and analyzed visions of understanding computer programming as a literacy and proposed a way to achieve it [42].

One of the most important contributions to CT in K–12 education arose from the empiricist side of computing's disciplinary debates. Crystallizing over a decade's worth of research by several teams of researchers, in his 1980 book *Mindstorms* Seymour Papert advocated an empirical approach to knowledge construction using computers and the LOGO language [44]. He described *procedural thinking* as a powerful intellectual tool. His approach was thoroughly empiricist: the tangible, physical nature of the machine "provides a more grounded reference than can any abstract work"—a view he and his colleagues advocated a decade before [35]. Papert appears to be the first to use the phrase *computational thinking* to describe all this [44, p.182].

With his colleagues Papert also argued that programming was a great tool for concretizing Pólya's classic text *How to Solve It* [45] on problem solving in mathematics [35]. The connection to mathematical thinking was strong at the beginning: Papert's early work did not explicitly discuss ways of thinking arising in computing, but in the decade after 1969 the views among his group of researchers evolved from mathematical "rigorous thinking" [35] to "procedural thinking" and "computational thinking" [44].

Papert's work on computers and education was seen by many as a breakthrough in education. The group's work influenced pioneering ideas, such as Kay's *Dynabook* [46], Solomon's work on computers and learning [47], diSessa and Abelson's *Turtle Geometry* [48], and the Boxer programming environment [49]. The culmination of Papert's work on CT in *Mindstorms* was not only groundbreaking but also comprehensive—whereas the earlier descriptions of computing's thinking patterns were abstract, Papert tailored his work to a deep understanding of how children learn: a feature that has played a major part in CT ever since. Papert's book and his follow-up essays became well known outside

computing circles for introducing constructionism: a vision of student-centered, project-based discovery learning using new technology. It was envisioned that computational ideas could serve learning in a broad variety of subjects, from Newton's laws to music, but more importantly, they "can change the way [children] learn everything else" [44, p.8]. Papert's work was followed by Russ and Beynon's empirical modeling (e.g., [50, 51, 52]), which used computers to found knowledge construction on purely empirical basis: learners can explore a phenomenon and build their own models through experimenting, trying out their own "what if" scenarios, observing, and measuring. Some authors anticipated that the very idea of programming would change when computational literacy becomes a common everyday activity for most people [38, 49].

The advocates of general computer literacy were eventually successful at getting schools to offer computer literacy courses—but those courses often focused on the use of basic computing applications such as word processors and spreadsheets, and not on computing concepts. An important milestone in the campaign to get better computing courses in K–12 education occurred in 1999, when the National Research Council published a report *Fluency with Information Technology*, which laid out an intellectual basis of a national education program that went beyond "computing literacy" by teaching capabilities, concepts, and skills [53]. Larry Snyder, chair of the NRC panel, published an influential textbook in 2003 for fluency courses in high schools and colleges [54].

While advocacy of computer literacy was a rather uncomplicated undertaking easily justified by know-how of productivity tools that were quickly gaining popularity, computational thinking was not an easy sell. One of the big and well-received claims of *Mindstorms* was that practice in programming developed cognitive skills that increased the students' problem solving abilities in many domains—a shift from "learning to program" to "programming to learn" [34]. This claim had been made repeatedly since the 1960s [31, 35, 44]. Large numbers of people argued—often without much empirical evidence—that programming prepares students' intellectual skills in other domains, too, or improves their metacognitive skills [55]. Many proponents of programming in K–12 education argued that learning how to program would have beneficial cognitive side-effects, such as rigorous thinking, understanding of general concepts, art of heuristics, generalized capability to "debug", problem-solving related metacognition, relativistic thinking, and epistemological commitment [8, 35].

Critics were quick to point out that numerous studies in developmental cognitive science and psychology of programming did not support these claims and, in some cases, reduced student problem solving ability in other domains [8, 55, 56, 57]. The critics argued that programming is not a unitary skill but a complex network of skills, that research results with adults as well as children spoke against spontaneous transfer of cognitive skills, and that the very idea of general domain-independent problem solving skills was problematic [8]. Moreover, they argued, learning to program was itself argued to be dependent on mathematical ability, analogical reasoning, conditional reasoning, memory capacity, procedural thinking, and temporal reasoning skills [8]. The general finding was that transfer happens only when computation is taught in the same environment it will be used in, and only then with significant practice and student reflection [3, 55, 57, 58].

Despite the contrary results from 1980s on, many modern promoters of CT have been criticized for continuing to claim that computational thinking enhances general cognitive skills in all knowledge domains [3]. Since the early research studies [8, 56], many education researchers have searched for evidence but have not found any. In 1997 Koschmann weighed in with more of the same doubts and debunked the claims referring to the analogy that learning programming is good for children's thinking skills just as learning Latin once was thought to be [58]. Guzdial reviewed again the evidence available by 2015 and reaffirmed there is no evidence to support the claim [3]. He reiterated the finding that CT skills may be useful in subjects like engineering or mathematics, and CT or programming may transform how the student sees a problem in those domains, but that is not *transfer*; it is direct application of computing in different domains [3]. There is no significant empirical support for the transfer claim.

Regardless, the march of computers into schools intensified through the 1990s and 2000s. In many schools today, children have personal tablets or workstations provided by the schools. The many reasons for this development have little to do with the CT debates above. Justifications for computers in schools include access to simulations and other teaching software, access to basic programming, participating in the Internet revolution, learning 21st century skills, preparation for employment in STEM fields, broadened social participation, allowing children to express individual creativity, and "crossing the digital divide" [3]. These developments in education mirrored accelerating changes in computerization of society rather than changes in general epistemology and the scientific method. Computers entered homes, the Internet changed the way people used computers to communicate, the Web introduced new ways to access information and to shop, mobile technology became commonplace, and as the prices plummeted, the user base grew dramatically. It has been argued that in some societies programming has worked its way into institutional and societal infrastructures the same way writing did earlier [42].

However, at the same time, there was a fundamental change in how computers were seen in science. Computing revolutionized the practices and principles of science and engineering, and that epistemological and methodological revolution is the very foundation of modern computational thinking. In terms of epistemology, computing changed some fundamental insights about scientific knowledge, and in terms of methodology, computing fundamentally changed how science is done.

4. NEW WAVE OF CT

4.1 Rise of Computational Science

Numerical analysis has always been important in science, engineering, and management from Newton's prolific calculations to the massive census processing tasks of the late 1800s [20, 59]. Easing the burden of calculation with machinery was a long time dream of many scientists and engineers. For example, Charles Babbage offered the Difference Engine to the British Government in 1820 as a way to make navigation tables more reliable and eliminate shipwrecks. The US Army sponsored research into analog computers in

the 1920s and electronic digital computers in the 1940s so that they could more reliably calculate the firing parameters of projectiles.

Scientists, familiar with numerical mathematics, have never been strangers to computational thinking [59]. Well before computer scientists came along many of them were already involved in numerical analysis and large-scale tabulating operations that entailed what we would today call CT [60]. After the birth of modern computing, both experimentally as well as theoretically oriented scientists saw something in computing to help them. In his writings about the first stored-program computers of the 1940s, John von Neumann described grid-oriented methods to solve differential equations found in the mathematical models of physical processes in many fields of science. He was interested in the possibility of using numerical simulation to evaluate mathematical models of physical process. Meanwhile, others looked to the new electronic computers as tools to analyze large data sets from scientific experiments.

Although a change was long bubbling under, the relationship between scientists and computing changed drastically in the 1980s. Prior to then, computing's value in science was primarily seen as a support for the traditions of experimenters as well as theoreticians. Experimenters had new ways to analyze large data sets. Theoreticians had numerical methods for solving their equations. Over the 1980s, computation became a third way of doing science, joining these traditions. That change was based on the insight that simulation could be a method in its own right. Scientists could explore phenomena by simulating them. Simulations could produce data for analysis, and they could allow tracking the behavior of systems for which no mathematical models are known. What is more, the idea of modeling a natural process as an information process and then using computation to explore the information process opened a horizon of new possibilities for understanding natural processes.

The supercomputer was the engine powering this revolution (see [61] for further discussion). NASA was using supercomputers in the early 1980s to evaluate air flows around aircraft instead of the traditional wind tunnel, and to discover heat shield materials that would allow a space probe to plunge deeply into Jupiter's atmosphere before burning up. In both these examples, the computations were part of the process of scientific discovery and understanding. Physicist Ken Wilson was awarded a Nobel Prize in 1982 for significant discoveries in the phase-change behaviors of materials under the influence of external magnetic or electrical fields—he conducted his studies with software systems he designed to carry out detailed and faithful simulations of materials using a supercomputer. Soon thereafter he became an advocate for computational science—the branches of various fields that conduct their scientific investigations using computer simulations. He and others described "grand challenge" problems in their fields that would yield to algorithms run on supercomputers [62]. Some of them used the term "computational thinking" to describe the habits of mind they developed while doing computational science. They turned their campaign into a political movement that culminated with the US Congress in 1991 passing the High Performance Computing and Communication Act that funded research in grand challenge problems. By 2000, leaders in many scientific fields had embraced computational science. And a few of them, notably Nobel Laureate David Baltimore in biology [63] claimed that their fields had become information sciences studying information processes such as DNA transcription found in nature.

The shift in thinking about science was as widespread as it was radical: Winsberg called 2000s "the age of computer simulation" [64], Chazelle wrote that algorithmic thinking was about to cause "the most disruptive paradigm shift in the sciences since quantum mechanics" [65], and Newell said most operational science was focused on information processes [66]. The 1980s computational science revolution opened a new wave of computational thinking—this time initiated not by computer scientists but by scientists in other fields. Computer simulation became the main engine of progress across sciences and engineering fields, and computational thinking was its mental toolbox. It also fueled another way of looking at "transfer" of CT: if natural phenomena in many fields are treated as computational information processes, then learning computing is not only useful but essential for work in those fields. People in those fields learn CT not by studying computer science, but by designing their own computations [3]. All the fields that have set up a computational branch—such as computational physics, computational chemistry, and bioinformatics—are natural fits for CT.

The idea that computation had become a "third pillar" of science (alongside theory and experiment) led to a new description of computing as a discipline, too. The older descriptions focused on study of algorithms; the newer focused on the study of information processes both natural and artificial [67, 68]. Within computing, the notion was that we not only *study* information processes, we aim to *harness* them for human purposes. The idea of harnessing led to increased attention to design, which was one of three pillars of computing articulated in 1989–the other two being theory and abstraction [29]. Design skills became central for creating dependable, reliable, usable, safe, and secure software. Design was seen as much broader than programming or coding. Design relies increasingly on the capacity to listen, innovate, and propose and prototype new solutions. Design also became one of the key elements of CT for many (e.g., [7]). Most importantly, design is the bridge between the technical and theoretical realms of computing and the needs and problems of communities and customers.

4.2 Computational Thinking Revived

In 2006, Jeannette Wing [5] revived the phrase "computational thinking" and started to market it to the broader academic audiences. Wing's description of CT was well aligned with arguments made in the previous decades: CT is a general-purpose thinking tool [22, 30, 31]; it builds on natural and artificial information processes [67]; it is about problem-solving, design, and the human condition [15, 21]; it has to take into account the available resources and reduce problems to smaller parts, abstract out some concerns, and choose appropriate representations [21, 23]. Similar to Dijkstra's "mental zoom lens" [21], Wing emphasized the ability to think at multiple layers of abstraction. Wing's essay listed a vast range of textbook level computing techniques, and paired some with everyday examples: packing a school bag is "prefetching and caching", seeking one's lost mittens by retracing one's steps is "backtracking", and choosing a line at a supermarket is "performance modeling for multi-server systems" [5]. A few years later Wing presented the "Cuny-

Snyder-Wing" definition of CT: "the thought processes involved in formulating problems and their solutions so that the solutions are represented in a form that can be effectively carried out by an information-processing agent"[1].

Wing's timing was opportune. The computational science revolution was finished and well publicized. The digitalization of society's major functions was proceeding at full speed. The field of computing education research had matured [69] and a multidisciplinary understanding of pedagogy of programming was emerging [70]. Many educators and political leaders were interested in STEM education and willing to include computer science in the definition of STEM. Scientists from many fields had embraced CT in their fields. Wing's formulation struck a resonant chord. She successfully used her position at NSF to spread word about CT and rally people around what soon became a movement to push CT into K–12 education.

Wing's rallying cry was successful on a number of levels, as witnessed by surveys on CT today [3, 4]. Some CT initiatives were deep and thoughtful, and represented the full richness of various CT visions. For example, major organizations like CSTA (Computer Science Teachers Association), CAS (Computing at School, a subsidiary of the British Computer Society), and ACARA[2] (Australian Curriculum, Assessment, and Reporting Authority) presented their own frameworks for computational thinking. CSTA's framework involved problem formulation, data organization and analysis, abstractions (including models and simulations), algorithmic thinking, evaluation of efficiency and correctness, and generalization and transfer to other domains [1]. CAS's framework consisted of a similar list: logical reasoning, algorithmic thinking, decomposition, generalization, patterns, abstraction, representation, and evaluation [2]. Both presented a rich portrayal of CT complete with skills, attitudes, useful techniques, and ideas for the classroom. After CT became a popular theme in K–12 education, there have been numerous articles, research studies, blog posts, and essays written on the topic. Different accounts of and approaches to CT have filled journals and books [3]. Surveys [4, 71] have charted the perspectives, definitions, and practices of CT in schools.

4.3 Risks Looming Over CT

In the enthusiasm to spread computational thinking, there is a risk of losing sight of CT's historical roots and of making claims that cannot be fulfilled. We have listed below seven such risks.

Lack of Ambition. Lack of historical insight and "reinventing the wheel" may lead to CT initiatives that are watered down versions of their 1980s predecessors. Decades earlier people like Wilson [62], Papert [44, 72], Russ, and Beynon [50, 51, 52] imagined a methodological and epistemological revolution—a transformation in how knowledge is produced, how scientific findings are made, and how learning happens in a thoroughly empiricist environment. Bolter [32] described how the computer redefined notions of space, time, progress, language, memory, creation, and intelligence. DiSessa [38] discussed major epistemological shifts in his empiricist visions of learning. Those visions were backed by the info-computational revolution of science [73] and the rapid digitalization of society's functions and recently people's everyday lives. It has been argued that programming has become an integral part of our socially constructed reality—a building block of our mental and societal infrastructure—and that our conceptions of literacy must account for programming and other forms of digital composition [42].

CT initiatives that focus solely on programming tools and techniques market a tasteless, scentless view of computing that emphasizes analytical abstract world far distant from the hands-on dirty complexities of the real world. In the early stages of the computer revolution, the focus on calculation may have justified a programming-and-techniques view, but since the 1980s the revolution has produced radical changes in the way we see the world and move in it. CT is no longer a way of adding new facts and statements to the computing body of knowledge. It is new, radically different way of looking at the world. It now aims for insight, understanding, more productive practice, and even wisdom. These fundamental changes should not go unheeded in K-12 education.

Dogmatism. In our excitement about the "unreasonable effectiveness" [74] of computing in thousands of topics, we should be generous and inclusive as were our holistic and pluralistic predecessors. Papert, for instance, was explicit that he does not advocate CT as the "best" way of thinking—he declared his openness to alternative approaches: in Papert's words and in the spirit of epistemological pluralism [75], true computer literacy is knowing when it is appropriate to make use of computers and computational ideas [44, p.155] while continuing to be open for alternative ways of knowing. In Bolter's description, humanity in the computer age does not speak of "destiny" but "options" [32].

In particular, we should not claim that CT is the best method of thinking and problem solving. Many other kinds of thinking have been invaluable in advancing science and technology and have been advocated by educators—for example, engineering thinking, science thinking, systems thinking, logical thinking, rational thinking, network thinking, ethical thinking, design thinking, critical thinking, and more. If all we have to work with is CT, our view of the world is diminished.

Knowing Versus Doing. The increasingly popular argument that CT is a skill rather than a particular set of applicable knowledge [1, 2] is remarkably light on what constitutes the skill or how to assess it. For example, the CSTA and CAS descriptions say that computational thinkers display a few characteristic "behaviors", but are vague about details [1, 2]. Moreover, a teacher's choice of educational objects—for example, employability, cognitive skills, general pedagogical value, basic computer concepts, or new ways of representing and manipulating existing knowledge—affects which skills are considered important [34].

A skill is an ability acquired over time with practice—not knowledge of facts or information. Most approaches to assessing CT assume that the body of knowledge—as outlined in the CSTA or CAS guidance—is the key driver of the skill's development. Consequently, we test students' knowledge, but not their competence or their sensibilities. Thus it is possible that a student who scores well on tests to explain and illustrate abstraction and decomposition can still be an incompetent or insensitive algorithm designer. The teachers sense this and wonder what they can do. The answer is, in

[1]https://www.cs.cmu.edu/link/research-notebook-computational-thinking-what-and-why

[2]http://www.australiancurriculum.edu.au/technologies/key-ideas

a nutshell, to directly test for competencies.

The realization that mastering a domain's body of knowledge need not confer skill at performing well in the domain is not new. As early as 1958, Polanyi discussed the difference between "explicit knowledge" (descriptions written down) and "tacit knowledge" (skillful actions) [76]. He famously said: "We know more than we can say." Polanyi gave many examples of skilled performers being unable to say how they do what they do, and of aspirants being unable to learn a skill simply by being told about it or reading a description. Familiar examples of tacit knowledge are riding a bike, recognizing a face, or diagnosing an illness. Many mental skills fall into this category too, such as learning a foreign language, programming, or thinking computationally. Every skill is a manifestation of tacit knowledge. People learn a skill only by engaging with it and practicing it.

To certify skills you need a model for skill development. One of the most famous and useful models is the framework created by Stuart and Hubert Dreyfus in the 1970s [77]. They said that practitioners in any domain progress through six stages: beginner, advanced beginner, competent, proficient, expert, and master. A person's progress takes time, practice, and experience. The person moves from rule-based behaviors as a beginner to fully embodied, intuitive, and game-changing behaviors as a master. Hubert Dreyfus gives complete descriptions of these levels in his book *On the Internet* [78]. We need guidelines for different skill levels of computational thinking to support competency tests.

The CAS[3] and K12CS[4] organizations have developed CT frameworks that feature progressions of increasingly sophisticated learning objectives in various tracks including algorithms, programming, data, hardware, communication, and technology. These relationship between these knowledge progressions and skill acquisition (e.g., [78]) is unclear. The CAS framework does not discuss abilities to be acquired during the progression. The K12CS framework gets closer by proposing seven practices—only three of which are directly related to competence at designing computations. Their notion of practice is "way of doing things" rather than an ability accompanied by sensibilities. Teachers who use these frameworks may find that the associated assessment methods do not test for the abilities they are after.

Exaggerated Claims. Despite the amount of empirical evidence to the contrary [3, 8, 55, 56, 57, 58], bold claims that computational thinking confers problem-solving skills transferable to non-computational knowledge domains still surface as if the transfer problem had never been studied. Surely there are CT strategies that are useful in different contexts. But over the long history of claims about such transfer those claims have never been substantiated [3].

Lack of historical insight also risks repeating the same mistakes again. For example the CSTA and CAS definitions strongly overlap with the definitions of object-oriented programming advocated in the late 1990s and later abandoned when the US Advanced Placement curriculum founded on them failed. These programming oriented definitions miss the richness of computer architecture, systems, networks, design, and computational science [29]. CT initiatives would benefit from knowing what went wrong in earlier, similar initiatives.

[3]https://community.computingatschool.org.uk/resources/2324

[4]https://k12cs.org

Narrow Views of Computing. Some critics of CT have argued that computational thinking is programming in disguise, "*a battle cry for coding in K–12 education*" [79]. A number of initiatives, such as Year of Code, Hour of Code, Code.org, and European Code Week, indeed adopted a coding-oriented view and promoted "coding" as something all children should learn. Two risks arise if coding is accepted as the aim of CT. The first is terminological confusion: "Coding" is just one part of the program construction process and not even the part that requires the most computational thinking. Many central concepts of coding—like iteration and selection—are not even central to computational thinking.

The second risk is the implication that "coding" is the essence of CT or CS. The myth that "CS=programming" emerged in the 1970s, and was fueled by the software engineering discussions of the 1970s. It was beaten back but re-emerged in computer literacy pushes of the 1990s [20]. It took a sustained effort of CS educators to demonstrate that the field is much broader than coding and to exorcise that myth [29, 20]. That myth was never embraced by the computing pioneers we discussed earlier [20]. Coding skills are less and less relevant to the typical design challenges and design tools of modern computing. CT initiatives should try to avoid the "computing = programming" trap. A deep understanding of the disciplinary history and breadth of computational thinking might help to avoid that trap.

Overemphasis on Formulation. The word "formulate" appears frequently in CT definitions—for example, Aho [7], Cuny-Snyder-Wing[5], and Wolfram[6] all say that CT is formulating problems so that a computer can solve them. The word "formulate" is being used in two different ways. One is "design computations" and the other is "express commands calling for a computation." The problem with the second is that people can issue commands or push buttons without engaging in CT. Descriptions of CT are either unclear about what they mean by formulation or they vacillate between the meanings of formulation. We would be better off saying that CT is to design rather than to formulate.

Losing Sight of Computational Models. In his discussion of CT, Aho was explicit that computational thinking is *design relative to a computational model* [7]. The computational steps and algorithms are means to control and instruct the computational model. We need to show our students what their programs are controlling before they can understand how to design programs that produce intended effects. Computer scientists are fond of Turing machines, register machines, and advanced neural networks (deep learning) as higher- or lower-level computational models. Every computational science has its own dominant models. For example, computational fluid dynamics uses simulations of the Stokes Equation on a grid, and bioinformatics uses string matching to sift through mounds of genome data. Computational thinking in all these fields is not only control of existing computational models; it is design of new ones. CT initiatives should bear in mind the strong relationship between CT and the behavior of machines—theoretical or practical—because losing that insight may risk exaggerated claims of applicability of CT.

[5]https://www.cs.cmu.edu/link/research-notebook-computational-thinking-what-and-why

[6]http://blog.stephenwolfram.com/2016/09/how-to-teach-computational-thinking/

5. CONCLUSIONS

Computational thinking has come a long way since numerical recipes of scientific computing and Perlis's 1960 descriptions of algorithmizing. The computational thinking movement has successfully established itself in K-12 education in a growing number of countries [4]. There is a large body of empirical research on CT and learning outcomes as well a growing body of literature on CT [3]. The latest computational thinking movement has great intellectual debt to three important historical currents.

First, the key concepts, narratives, and major arguments of CT were worked out during many years of debate from the 1950s to the 1990s. The numerous descriptions of algorithmic thinking, computing's unique thought patterns, and computing's general-purpose thinking tools are all direct predecessors of today's descriptions of CT. In both computing and CT, definitions varied from narrow to broad.

Second, computational thinking owes much to the many educators who launched computing initiatives in schools. Seymour Papert stands out for his pedagogical vision of constructionism, wherein students learn programming by exploring and practicing it. Papert joined many of his predecessors in advocating that the problem-solving skills learned in programming carry over into other domains. If true, this transference claim could cause a revolution in education. However, after many thorough investigations, education researchers have concluded that this claim cannot be substantiated. Even so, we should not let the many other powerful and radical empiricist ideas of Papert and others disappear from CT discussions. There is much work left—for example, the pedagogical and educational roles of CT [80], the need to take each child's individual learning style into account and avoid a one-size-fits-all approach [44], learning how to assess CT learning outcomes, and deciding what exactly should be taught, on what levels, and how [3].

Third, CT owes a great debt to the computational sciences movement begun in the 1980s. Grand challenges in science yielded one after the next to new computational methods and explorations backed by massive supercomputers. Computational scientists proclaimed that computing had become the third pillar of science and developed their own notions of CT. Similarly, researchers in the social sciences and humanities have been making great strides from bringing powerful computational methods into their own domains. The computing revolution is spreading into all fields because of the tremendous benefits computing brings. Many are learning CT from engaging with computing in their fields. As we look over all these accomplishments, and take pride that the technology developed in our field has had such an impact, we should resist two temptations of hubris. One is the hypothesis that every domain of knowledge is ultimately reducible to computing. The other is the belief that CT drives the revolution, when in truth the revolution drives the spread of CT. While computing gives a new lens to interpret the world, it does not render other lenses obsolete. Computing and CT should enrich science, not homogenize it.

Computational thinking—the habits of mind that many of us have developed from designing programs, software packages, and computations performed by machines—offers very powerful mental tools for people who design computations. There is no need to make exaggerated claims—notably automatic transfer of CT skill across domains or about superiority of CT over other ways of thinking and practicing. CT has a rich and broad history of many competing and complementing ideas. Many of its central ideas have been discovered, rediscovered, rebranded, and redefined over and over again. Many dead-ends have been found and misconceptions have been debunked, just to see them rise again in the next iteration. Many ambitious and powerful ideas have been all but forgotten. Ignoring the history and the work of the field's pioneers diminishes the computational thinking movement rather than strengthening it.

6. REFERENCES

[1] Computer Science Teachers Association. Operational definition of computational thinking. [Online]. Available: http://www.csta.acm.org/Curriculum/sub/CurrFiles/CompThinkingFlyer.pdf

[2] Computing at School. Computational thinking: A guide for teachers. [Online]. Available: http://community.computingatschool.org.uk/files/6695/original.pdf

[3] M. Guzdial, *Learner-Centered Design of Computing Education: Research on Computing for Everyone*, ser. Synthesis Lectures on Human-Centered Informatics. San Rafael, CA, USA: Morgan & Claypool, 2015.

[4] L. Mannila, V. Dagiene, B. Demo, N. Grgurina, C. Mirolo, L. Rolandsson, and A. Settle, "Computational thinking in K–9 education," in *Proceedings of the Working Group Reports of the 2014 on Innovation & Technology in Computer Science Education Conference*, ser. ITiCSE-WGR '14. New York, NY, USA: ACM, 2014, pp. 1–29.

[5] J. M. Wing, "Computational thinking," *Communications of the ACM*, vol. 49, no. 3, pp. 33–35, 2006.

[6] ——, "Computational thinking and thinking about computing," *Philosophical Transactions of the Royal Society A*, vol. 36, no. 1881, pp. 3717–3725, 2008.

[7] A. V. Aho, "Ubiquity symposium: Computation and computational thinking," *Ubiquity*, vol. 2011, no. January, 2011.

[8] R. D. Pea and D. M. Kurland, "On the cognitive effects of learning computer programming," *New Ideas in Psychology*, vol. 2, no. 2, pp. 137–168, 1984.

[9] H. A. Simon, *The Sciences of the Artificial*, 1st ed. Cambridge, MA, USA: MIT Press, 1969.

[10] L. Fein, "The role of the university in computers, data processing, and related fields," *Communications of the ACM*, vol. 2, no. 9, pp. 7–14, 1959.

[11] S. Gorn, "The computer and information sciences: A new basic discipline," *SIAM Review*, vol. 5, no. 2, pp. 150–155, April 1963.

[12] L. A. Zadeh, "Computer science as a discipline," *The Journal of Engineering Education*, vol. 58, no. 8, pp. 913–916, 1968.

[13] R. W. Hamming, "Numerical analysis vs. mathematics," *Science*, vol. 148, no. 3669, pp. 473–475, April 23 1965.

[14] ——, "One man's view of computer science," *Journal of the ACM*, vol. 16, no. 1, pp. 3–12, 1969.

[15] G. E. Forsythe, "A university's educational program in computer science," *Communications of the ACM*, vol. 10, no. 1, pp. 3–11, 1967.

[16] P. C. Hammer, "Computer science and mathematics," in *Papers of the First IFIP World Conference on Computer Education*, B. Scheepmaker and K. L. Zinn, Eds. Amsterdam, The Netherlands: International Federation for Information Processing, August 24–28 1970, pp. I/65–67.

[17] A. Newell, A. J. Perlis, and H. A. Simon, "Computer science," *Science*, vol. 157, no. 3795, pp. 1373–1374, 1967.

[18] G. E. Forsythe, "The role of numerical analysis in an undergraduate program," *The American Mathematical Monthly*, vol. 66, no. 8, pp. 651–662, 1959.

[19] D. L. Katz, "Conference report on the use of computers in engineering classroom instruction," *Communications of the ACM*, vol. 3, no. 10, pp. 522–527, 1960.

[20] M. Tedre, *The Science of Computing: Shaping a Discipline*. New York, NY, USA: CRC Press / Taylor & Francis, 2014.

[21] E. W. Dijkstra, "Programming as a discipline of mathematical nature," *American Mathematical Monthly*, vol. 81, no. 6, pp. 608–612, 1974.

[22] D. E. Knuth, "Computer science and its relation to mathematics," *American Mathematical Monthly*, vol. 81, no. Apr.1974, pp. 323–343, 1974.

[23] ——, "Algorithmic thinking and mathematical thinking," *American Mathematical Monthly*, vol. 92, no. March, pp. 170–181, 1985.

[24] J. Statz and L. Miller, "Certification of secondary school computer science teachers: Some issues and viewpoints," in *Proceedings of the 1975 Annual Conference*, ser. ACM '75. New York, NY, USA: ACM, 1975, pp. 71–73.

[25] J. F. Traub, *Iterative Methods for the Solution of Equations*. Murray Hill, NJ, USA: Bell Telephone Labs, Inc., 1964.

[26] J. A. Feldman and W. R. Sutherland, "Rejuvenating experimental computer science: A report to the National Science Foundation and others," *Communications of the ACM*, vol. 22, no. 9, pp. 497–502, 1979.

[27] P. J. Denning, E. Feigenbaum, P. Gilmore, A. Hearn, R. W. Ritchie, and J. Traub, "A discipline in crisis," *Communications of the ACM*, vol. 24, no. 6, pp. 370–374, 1981.

[28] P. J. Denning, "Eating our seed corn," *Communications of the ACM*, vol. 24, no. 6, pp. 341–343, 1981.

[29] P. J. Denning, D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner, and P. R. Young, "Computing as a discipline," *Communications of the ACM*, vol. 32, no. 1, pp. 9–23, 1989.

[30] G. E. Forsythe, "What to do till the computer scientist comes," *American Mathematical Monthly*, vol. 75, no. May 1968, pp. 454–461, 1968.

[31] M. Minsky, "Form and content in computer science," *Journal of the ACM*, vol. 17, no. 2, pp. 197–215, 1970.

[32] J. D. Bolter, *Turing's Man: Western Culture in the Computer Age*. Chapel Hill, NC, USA: The University of North Carolina Press, 1984.

[33] H. Abelson and G. J. Sussman, *Structure and Interpretation of Computer Programs*, 2nd ed. Cambridge, MA, USA: MIT Press, 1996.

[34] P. Mendelsohn, T. R. G. Green, and P. Brna, "Programming languages in education: The search for an easy start," in *Psychology of Programming*, J.-M. Hoc, T. R. G. Green, R. Samurçay, and D. J. Gilmore, Eds. London, UK: Academic Press, 1990, pp. 175–200.

[35] W. Feurzeig, S. Papert, M. Bloom, R. Grant, and C. Solomon, "Programming-languages as a conceptual framework for teaching mathematics," *SIGCUE Outlook*, vol. 4, no. 2, pp. 13–17, 1970.

[36] P. Kugel, "Computer science departments in trouble," *Communications of the ACM*, vol. 31, no. 3, p. 243, 1988.

[37] B. A. Sheil, "Teaching procedural literacy," in *Proceedings of the ACM 1980 Annual Conference*. New York, NY, USA: ACM, 1980, pp. 125–126.

[38] A. A. diSessa, *Changing Minds: Computers, Learning, and Literacy*. Cambridge, MA, USA: MIT Press, 2000.

[39] J. A. Culbertson, "Whither computer literacy?" in *Microcomputers and Education*, J. A. Culbertson and L. L. Cunningham, Eds. The University of Chicago Press, 1986, vol. 1, pp. 109–131.

[40] A. P. Ershov, "Programming: The second literacy," *Microprocessing and Microprogramming*, vol. 8, no. 1, pp. 1–9, 1981.

[41] A. Vee, "Proceduracy: Computer code writing in the continuum of literacy," Ph.D. dissertation, University of Wisconsin at Madison, Madison, WI, USA, 2010.

[42] ——, "Understanding computer programming as a literacy," *Literacy in Composition Studies*, vol. 1, no. 2, pp. 42–64, 2013.

[43] D. E. Knuth, *Literate Programming*, ser. CSLI Lecture Notes. Stanford, CA, USA: Center for the Study of Language and Information, 1992, no. 27.

[44] S. Papert, *Mindstorms: Children, Computers, and Powerful Ideas*. New York, NY, USA: Basic Books, 1980.

[45] G. Pólya, *How to Solve It*, 2nd ed. London, UK: Penguin Books Ltd., 1957.

[46] A. C. Kay, "A personal computer for children of all ages," Xerox Palo Alto Research Center, Technical Memo, 1972.

[47] C. Solomon, *Computer Environments for Children: A Reflection on Theories of Learning and Education*. Cambridge, MA, USA: MIT Press, 1986.

[48] H. Abelson and A. A. diSessa, *Turtle Geometry: The Computer as a Medium for Exploring Mathematics*. Cambridge, MA, USA: MIT Press, 1980.

[49] A. A. diSessa and H. Abelson, "Boxer: A reconstructible computational medium," *Communications of the ACM*, vol. 29, no. 9, pp. 859–868, 1986.

[50] S. Russ, "Empirical modelling: The computer as a modelling medium," *Computer Bulletin*, vol. 39, no. 2, pp. 20–22, 1997.

[51] M. Beynon, "Constructivist computer science education reconstructed," *Innovation in Teaching and Learning in Information and Computer Sciences*,

vol. 8, no. 2, pp. 73–90, 2009.

[52] ——, "Modelling with experience: Construal and construction for software," in *Ways of Thinking, Ways of Seeing: Mathematical and other Modelling in Engineering and Technology*, C. Bissell and C. Dillon, Eds. Berlin, Heidelberg: Springer, 2012, pp. 197–228.

[53] Committee on Information Technology Literacy, *Being Fluent with Information Technology*. Washington, DC, USA: National Academy Press, 1999.

[54] L. Snyder, *Fluency With Information Technology: Skills, Concepts, & Capabilities*, 6th ed. Harlow, Essex, UK: Pearson, 2014.

[55] R. E. Mayer, J. L. Dyck, and W. Vilberg, "Learning to program and learning to think: What's the connection?" *Communications of the ACM*, vol. 29, no. 7, pp. 605–610, 1986.

[56] D. H. Clements and D. F. Gullo, "Effects of computer programming on young children's cognition," *Journal of Educational Psychology*, vol. 76, no. 6, pp. 1051–1058, 1984.

[57] G. Salomon and D. N. Perkins, "Transfer of cognitive skills from programming: When and how?" *Journal of Educational Computing Research*, vol. 3, no. 2, pp. 149–169, 1987.

[58] T. Koschmann, "Review: Logo-as-Latin redux," *The Journal of the Learning Sciences*, vol. 6, no. 4, pp. 409–415, 1997.

[59] R. S. Westfall, *Never at Rest: A Biography of Isaac Newton*. New York, NY, USA: Cambridge University Press, 1980.

[60] D. A. Grier, *When Computers Were Human*. Princeton, NJ, USA: Princeton University Press, 2005.

[61] P. J. Denning, "Remaining trouble spots with computational thinking," 2016, forthcoming in *Communications of the ACM*.

[62] K. G. Wilson, "Grand challenges to computational science," *Future Generation Computer Systems*, vol. 5, no. 2–3, pp. 171–189, 1989.

[63] D. Baltimore, "How biology became an information science," in *The Invisible Future*, P. J. Denning, Ed. New York, NY, USA: McGraw-Hill, 2002, pp. 43–55.

[64] E. B. Winsberg, *Science in the Age of Computer Simulation*. Chicago, IL, USA: The University of Chicago Press, 2010.

[65] B. Chazelle, "Could your iPod be holding the greatest mystery in modern science?" *Math Horizons*, vol. 13, no. 4, pp. 14–15, 30–31, 2006.

[66] D. G. Bobrow and P. J. Hayes, "Artificial intelligence – where are we?" *Artificial Intelligence*, vol. 25, pp. 375–415, 1985.

[67] P. J. Denning, "Computing is a natural science," *Communications of the ACM*, vol. 50, no. 7, pp. 13–18, 2007.

[68] P. J. Denning and C. H. Martell, *Great Principles of Computing*. Cambridge, MA, USA: MIT Press, 2015.

[69] Simon, "Emergence of computing education as a research discipline," Ph.D. dissertation, Aalto University, Finland, 2015.

[70] J. Sorva, "Visual program simulation in introductory programming education," Ph.D. dissertation, Aalto University, Finland, 2012.

[71] S. Grover and R. D. Pea, "Computational thinking in K–12: A review of the state of the field," *Educational Researcher*, vol. 42, no. 1, pp. 38–43, 2013.

[72] S. Papert, "An exploration in the space of mathematics educations," *International Journal of Computers for Mathematical Learning*, vol. 1, no. 1, pp. 95–123, 1996.

[73] G. Dodig-Crnkovic and V. C. Müller, "A dialogue concerning two world systems: Info-computational vs. mechanistic," in *Information and Computation: Essays on Scientific and Philosophical Understanding of Foundations of Information and Computation*, ser. World Scientific Series in Information Studies, G. Dodig-Crnkovic and M. Burgin, Eds. Singapore: World Scientific, 2011, vol. 2.

[74] R. W. Hamming, "The unreasonable effectiveness of mathematics," *The American Mathematical Monthly*, vol. 87, no. 2, pp. 81–90, 1980.

[75] S. Turkle and S. Papert, "Epistemological pluralism: Styles and voices within the computer culture," *Signs*, vol. 16, no. 1, pp. 128–157, 1990.

[76] M. Polanyi, *The Tacit Dimension*. Chicago, IL, USA: The University of Chicago Press, 1966.

[77] S. E. Dreyfus and H. L. Dreyfus, "A five-stage model of the mental activities involved in directed skill acquisition," University of California, Berkeley, Research Report ORC-80-2, 1980.

[78] H. L. Dreyfus, *On the Internet*, 2nd ed. London / New York: Routledge, 2001.

[79] Y. B. Kafai, "From computational thinking to computational participation in K–12 education," *Communications of the ACM*, vol. 59, no. 8, pp. 26–27, 2016.

[80] C. Schulte, "Reflections on the role of programming in primary and secondary computing education," in *Proceedings of the 8th Workshop in Primary and Secondary Computing Education*, ser. WiPSE '13. New York, NY, USA: ACM, 2013, pp. 17–24.