What Makes Computational Thinking so Troublesome?

Arnold Pears
Department of Learning in Engineering Sciences
KTH Royal Institute of Technology
Stockholm, Sweden
orcid.org/0000-0002-5184-4743

Matti Tedre
School of Computing
University of Eastern Finland
Joensuu, Finland
orcid.org/0000-0003-1037-3313

Teemu Valtonen
School of Applied Educational
Science and Teacher Education
University of Eastern Finland
Joensuu, Finland
orcid.org/0000-0002-1803-9865

Henriikka Vartiainen
School of Applied Educational
Science and Teacher Education
University of Eastern Finland
Joensuu, Finland
orcid.org/0000-0001-6005-907X

***Abstract*—This Research Full Paper addresses the definition and implementation of Computational Thinking (CT) in K-12 education. CT is the focus of ongoing debate about the future of computing in schools, and this poses a number of challenges. Early discussions on the topic were plagued by vague and all-encompassing definitions of the term, which raised awareness of a need for school learning focused on emerging computational systems and their impact on society and the lives of citizens, but failed to address what should actually be included in the revised curricula. As some of those issues have been addressed, a number of problems remain. This paper analyses some recent and future developments in the field of computing, and expands the discussion on the vexed question of the nature and limits of computational thinking. We ask whether the current mainstream literature and curriculum for CT is well positioned in terms of the future towards which computing is heading. We conclude by summarising the implications these questions have for the future development of K-12 curricula and teaching practices.**

***Index Terms*—computational thinking, machine learning, programming paradigms, artificial intelligence, data agency**

I. Introduction

Over the past twelve years, computational thinking (CT) has emerged as a central theme for computing education, embraced by many teachers, educators, and policymakers interested in introducing an agenda of computing education for all [19]. The phrase "computational thinking" has become popular, and has been featured in the names of research programs, research centers, journal special issues, conferences, and NSF funding calls [11]. A large number of countries have embarked on a series of actions, and have introduced or are about to introduce CT into their national curricula [28], [39].

There is good reason to be wary in the face of these rapid changes. CT is not a well defined area, other than a general agreement that there is a unique kind of thinking which is applied in formulating computational solutions and systems in such a manner that they can be automated (executed).

Many advocates of CT believe that CT skills are transferable across domains, a claim that has been falsified many times over in learning studies [19]. Computational thinking has also been backed by large and influential organizations, including the International Society for Technology in Education (ISTE), the Computer Science Teachers Association (CSTA), and the Association for Computing Machinery (ACM). This has given the term considerable visibility and political influence, despite the lack of a clear understanding of what distinctly new types of thinking are involved.

The concept of computational thinking and the CT movement in computing education have not emerged uncontested. The conceptual foundations of the term are problematic, and have been criticized from a range of perspectives. Many critics have pointed out a disconnect between the CT movement and the long history of computing education research, as well as the tendency to consider it something new and different [19], [41], [43]. Whether CT ideas are really unique, or whether they are to some degree common to many fields of science, technology, and mathematics, has been debated extensively [3], [17], [29], [32]. At the curricular level there is also disagreement on whether CT education should be concerned with high-level concepts and skills, programming, design, or machine concepts [2], [22], [26], [28].

In his essay "Remaining Trouble Spots with Computational Thinking," Peter J. Denning [9] argued that education researchers and teachers struggle especially over three questions: "What is computational thinking?," "How can it be assessed?," and "Is it good for everyone?." He then continued to address each of the questions, expanding on the historical roots of computational thinking, research on skill development, and the transfer hypothesis in education

This paper extends the work begun in Denning's essay on "trouble spots". We identify eight new troublesome ideas and trends in CT and the CT movement. We argue that each of

these areas warrant substantial further research and discussion. In particular we draw attention to problems with CT imagery, the abstraction levels at which CT is envisaged to operate, and the kinds of skills involved in CT. Some of these trouble spots are more common across the multitude of different CT initiatives than others, but all of them are regularly found in some form or another.

II. Why "Coding"?

A. Historically, "coding" is a poor choice of terminology

It is somewhat symptomatic that the first troublesome element in regard to CT is terminological. At one level the issue appears quite trivial and is directly related to interpretations placed on the terms "coding" and "programming." In the formative years of the computing profession in the early 1940s, coders were skilled in mathematics, while operators (whose job was to punch cards and manipulate plug boards) needed much less training: Translated into the wartime context the former had the military rank and salary of officers, and the latter those of enlisted personnel [4]. The computing pioneer Grace Hopper reminisced that the word "programmer" arrived in the US from the UK [20], and the language soon changed so that "coders" referred to low-level clerical workers while "programmers" assumed a higher position in the computing hierarchy [13]. Unlike programmer and systems analyst, "coder" did not appear on the list of recognized occupations of the US Department of Labor.

Many programming pioneers adopted the "programmer" label and wore it with pride. Edsger Dijkstra, for example, campaigned for recognition of programming as a theoretically grounded, intellectually noble discipline, and shunned the idea of programmers as glorified coders and designers of clever tricks [12]. Even when computing pioneers said "programmer" and meant it the Dijkstra way, the public image associated with the term was that of low-level technicians or coders [14], [15]. Unable to successfully promulgate "programming" as a high status and noble discipline, computer science educators gave ground, and instead focused on counteracting the misleading public perception that computer science was synonymous with programming [33], [35]. They lamented the public perception in which the rich intellectual depth of computing as a discipline was reduced to, and encapsulated by, a minor part of the software construction process.

When computing started to develop as a discipline, many pioneers of computing considered that the "CS=programming" myth undermined the academic credibility of this emerging field, and that in the eyes of the public it rendered computing jobs technical jobs [35], [36]. This conception of computing, and by implication computational thinking, persists in current definitions of CT, which strongly emphasise a mix of software design practices and sequential imperative control flow constructs (e.g. sequences and loops).

The software engineering movement of the 1970s struggled unsuccessfully to dispel this myth, which persists to the current day [15]. A number of influential reports both denounced and disowned the "CS=programming" idea [1], [44] and advocated a view of computing as a combination of theory, science, and engineering, with the 1989 "Computing as a Discipline" report being one of the most influential [10]. As Armoni [2] noted, those episodes remain in the collective memory of older generations of computer scientists, yet despite a widespread agreement that "coding" is a poor and historically loaded term, it remains deeply entrenched in numerous initiatives around the world. Initiatives such as Sweden's programming at school have resuscitated the "CS=programming" misconception from the iniquity it deserves and used it in modern times to label important initiatives. Hence, the first trouble spot, self-evident it may be [2], is as follows:

> CT initiatives centered on the idea "CS = programming," recently narrowed to "CS = coding," anchor themselves to a barren interpretation of computing.

B. Coding paints a narrow image of software design

The second trouble spot, closely related to the first, is that focus on "learning coding" emphasizes skills exclusively relevant to the limited context of the tasks concerned with software construction and systems design. In fact, those of most pedestrian and routine nature! Coding—a subset of programming—is not generally denoted a crucial part of the intellectual effort required to build high-quality information systems or software systems. By focusing on coding skills, many K-12 educational initiatives, such as code.org, Hour of Code, and European Code Week [28], contribute to a troubling tendency to overemphasize the importance of coding in the broad spectrum of other skills essential to the field. Although there is little disagreement with this objection among computer scientists, "coding" persists as the rallying cry of K-12 computing initiatives, such as the three above.

This is exceptionally problematic in an environment where design, impact of systems on the functioning of society at large, and ethical considerations are major challenges of an importance far beyond the scope of the technical details of implementing computational systems. The complexity of "'coding" achievable in most K-12 contexts combined with the limited computing background of teachers makes it highly unlikely that these activities will lead to pupils developing deep insights into the computing systems that surround them. The second trouble spot is:

> Those CT initiatives that emphasize coding focus on one narrow and technical element of software design, hiding a whole spectrum of other important skills necessary for the field.

III. Abstractional quagmire

A. Programming focuses attention on a very specific aspect of computing

The third trouble spot in current CT education concerns the constant evolution of software production from lower-level coding tasks towards higher and higher levels of abstraction. Even if we consider only the programming element in the software development process, each decade has seen programming effort become increasing automated, encapsulated into

libraries, and rendered irrelevant to most programmers. In the early years of programming, compiling assembly language to machine code was called "automatic programming," [34] and simplifying or automating programming tasks has been a major driver of development of computing ever since. Evidence of this lies in the difference between software development of the late 1990s and today. Modern development platforms, libraries, engines, and cloud services automate an increasing number of systems development activities that required repetitive and tedious programming work just two decades ago.

The tools and knowledge in computing have evolved at such a rate, that a topic that was worthy of a Ph.D. thesis in the year 2000 is often a routine task today. As a case in point consider game development and app development packages that liberate the designer from a substantial part of the programming work previously associated with game development. By providing game and simulation engine platforms recent developments shift much of the computation under the hood obviating the need for access to sophisticated programming knowledge. More importantly, programming jobs are not exempt from the ongoing automation of knowledge work. Quite the contrary, many aspects of programming work bear all the signs of incoming automation: easy outsourcing and offshoring, high predictability, symbolic nature, and a wealth of data about previous solutions [6].

An old adage in computing says that you need to be familiar with no more than one level of abstraction below the one you are working at. It was beneficial for machine language programmers to know how their computers worked at the level of electronics, although that was not necessary. It was good (but not necessary) for C programmers to know machine language. For Java programmers, pointer arithmetic and details of memory management are beneficial but certainly not necessary, and for users of ready-made data container libraries it does not hurt to be familiar with how their data structures and algorithms are actually implemented (but that knowledge is not really required). We are currently at a point where it is legitimate to ask whether any knowledge about how to implement data structures and the algorithms to control them, is any longer a central skill for most computing professions. Hence, the third trouble spot is:

> Many skills and abilities taught in CT education are at a low level of abstraction and likely to be increasingly irrelevant to the skills and competences underlying the automation of jobs in the future. We are teaching the CT of the past, not the future!

B. Coding is a clumsy way to control the computer

The fourth trouble spot is an extension of the third one, and concerns the continuous diversification and development of software. The constant development of software tools guarantees that programming is increasingly rarely the best interface through which to command the computer or with which to automate things using a computer. Soon after the birth of the modern computer, programming became the primary way and usually the only way to control automatic computers [18]. Programming became the interface to the computer, and it remained the primary interface for controlling computer-based automation until the last two decades of the 1900s [18].

Today, programming continues to provide its cognoscenti more power and flexibility in controlling the computer than any other means, but often at prohibitively high cost. The advent of user-friendly software in an increasing range of application areas now renders problems that previously required sophisticated programming skills tractable, and largely independent of programming skill, thus freeing end-users to focus on the problem domain instead of the technical details of how to program a solution. For instance, statistical software has gradually automated and abstracted tasks that used to require a substantial quantity of non-trivial coding. Design environments have automated numerous software development tasks, particularly in terms of "drag and drop" support for user interface design and coding. Modern machine control environments have automated manufacturing in ways that previously required many programmers. Instead of requiring everyone to control the computer the clumsy programming way, progress in high level software has enabled an increasing number of users to harness the potential of computers without the need to program them. There is no reason to believe that this trend towards higher level algorithmic systems will not persist. Although programming knowledge may retain an important role as the ABCs of technological literacy, the fourth problem spot is:

> Many lower-level programming concepts and skills are becoming less and less relevant to controlling computational processes and are required to automate tasks in fewer and fewer problem domains.

C. Determinism and reductionism no longer suffice

Two of the credos of traditional, "good old-fashioned computing," were that computation is deterministic, and that all behaviors of a computer executing a program can be reduced to a small set of rules, transactions, that define state changes in the medium the computation is run on. This wisdom was long summarised in the von Neumann architecture and the idea of the "'notional machine" [40].

Although they remain correct in principle, both started to lose their currency a long time ago in relation to a range of applications. A vast amount of the computing that lies behind today's sociotechnical infrastructure is run on massively parallel, not tightly synchronized, intercommunicating networks of machines and users, programs, processes, wires, and signals, with bugs (and error-correction) prevalent at all levels of the system. Even perfect knowledge of the entire state space of the system will not allow one to completely predict the future states of either the whole system, nor most of its parts [16], [27]. Nondeterminism has emerged as a key paradigm in the world of the multi-core processors and optimisation of performance exploiting the computational capacity of graphical processing units (GPU), for instance.

The shift from rule-driven control flow based programs to data-driven systems and machine learning has further diluted

the idea that every computational state and step can be unambiguously inferred from the states and state transition rules of the system. Artificial neural networks are deterministic systems in principle, but the massive number of neurons and connections between them makes them black boxes in practice, whose behavior is not guaranteed but that work approximately correctly with satisfactorily high levels of probability in most circumstances. With the triumph of techniques like deep learning, some computing communities' objectives and measures for success have shifted from guaranteed optimal solutions to consistently getting very good results with very high probability [7]. Examples of the limits of deterministic and reductionist thinking are many outside the field of computing. For example, one cannot reduce and explain, in computational states and steps, why a self-driving car does what it does. How the Internet still works, with all its complexity, and known bugs in many protocol implementations, is a tribute to the power of pragmatic engineering on a large scale. Redundancy, overcapacity, fault-tolerant protocols, and over-engineering provide a workable system that is not reducible to state transitions at the scale upon which it operates. Many computing systems are increasingly chaotic, complex systems that require different kinds of tools from those in which insight is offered by many CT initiatives.

> CT is typically defined in terms of reductionist and deterministic thinking, the power of which to explain computational systems is diminishing.

D. Basic CT doesn't teach you how your world works

One common justification for including programming in the K-12 school curriculum is that programming teaches children to understand how the world around them works [19]. However, the ABCs of computing typically taught in elementary computing, such as binary representation of data, Boolean logic, and the three control structures (sequential execution, branching, and looping), are low-level concepts that will not explain the working of advanced, complex computing systems or high-level emergent phenomena. That cognitive gap between levels of explanation is also familiar to us in many other fields. One does not understand animal behavior by studying the principles of DNA, and one will not learn how weather works by studying the kinetic theory of gases. While both conceptions are sufficient to explain functional outcomes at one level of abstraction, neither explains the extremely complex and emergent behaviors at the level of whole systems. By analogy we conclude that learning how to program may teach one the ABCs of the computing field, but it not does generate higher-level insights into the workings of social media, recommender systems, Google, deep fakes, or self-driving cars. Learning the basics of CT might not be a bad idea, but it does not fulfil the promise above; that is,

> Learning CT as currently defined does not lend children understanding of how their virtual worlds work.

IV. CT AND JOBS

A. Design determines profits

The seventh trouble spot concerns the increasingly important role played by design skills in computing. Most groundbreaking services today are about understanding a community and its needs; about a sense of human networks and interactions; about an appreciation of habits, behavior, and culture; and about designs that cater to those elements [11]. Service and app development is about design skills, and programming the services and apps (albeit usually not a trivial task) often does not constitute the central intellectual challenge. Software and hardware delights are increasingly an outcome of virtuosity in design instead of in algorithmic virtuosity or programming prowess [8]. The crucial breakthroughs are increasingly not programming breakthroughs but design breakthroughs. Most of the next ten thousand successful startups are expected to be based on understanding a problem domain, understanding user needs in it, understanding what can be automated there using ready-made libraries and cloud services, and designing a solution that fills these needs [24]. Classical programming plays a role, but only as one of the many roles involved in systems implementation and deployment.

There is a body of research that suggests that learning by designing can be applied to computing education with children [30], [31], [37] but that pedagogical approach has fallen out of favor in many education systems today [38]. Many current models of instruction consist of rigorous and highly scripted tasks and instructions from a teacher who guides the learners to a pseudo-discovery of some unifying principle [5]. In those models of instruction, the content and process of teaching are sequenced into well-defined subtasks, rules, and steps to follow, and learners' success is largely assessed in terms of their ability to reproduce what they have been taught [45]. We observe that these are exactly the kinds of skills that are subject to automation, and as such, they risk training learners in models of thinking that are increasingly antiquated.

What is more, design itself has made a transformation from rule-driven systems to data driven design. The deductive reasoning and rule-based programming that drove many early programming language experiments in schools [30] should be replaced by data-driven design and inductive reasoning as we argue that these skills are becoming increasingly important for future innovation and working life.

In response to global challenges compounded by the pace of automation, many companies and public sector organizations have also begun to create strategies that pursue new products, interfaces, processes, environments, skills and user experiences through collaborative design. Much of this intense interest is driven by the recognition that the vast majority of innovation and business-development opportunities call for cross boundary collaboration in order to connect computational thinking and data-driven design with deep understanding of human activity [24]:

> Virtuosity in communication and design, not programming prowess, is increasingly the key factor

behind bestselling apps and services.

B. Learning programming may not keep you your job, or get you a job!

The eighth, and perhaps the most important, trouble spot is related to the observation that teaching programming is frequently proffered, at least in newspaper op-eds and political speeches, as a solution to the job market disruptions caused by automation of knowledge work [6], [24]. Unfortunately, proposing programming as a solution to job disruption caused by automation of knowledge work almost completely misses the point. Firstly, jobs that disappear due to computerisation are not being transformed into large numbers of new programming jobs. Secondly, the force behind the latest disruptions in the job markets is not programming per se: it is a combination of traditional computing and machine learning. The latest disruptions have been more often data-driven than rule-driven [6], [24].

A look at the software solutions that have had the most radical impact in terms of changing our job markets in the past five years, and a look at the role programming plays in those solutions, reveals that the new horizons in automation of knowledge work are based on a combination of classic programming and machine learning [6], [24]. Here we diverge from the classical software development view, where software is explicitly written to exhibit certain behaviors: in machine learning behaviors are determined by neural network weights, which cannot be explicitly derived due to the enormous number of those weights. Instead, systems and algorithms are "trained" with massive volumes of data. Thus, in a world where everything is quantified and collected using sensors and software, it is much easier for algorithmically defined systems to "learn" behaviors and patterns from massive data sets instead of humans explicitly programming rules into the software [23]. We have reached a meta level of algorithmics and automation, where we define the manner in which the ultimate functionality is derived rather than the functionality itself. In many cases the functionality is not even definable in traditional logic but only discoverable from data like self-organizing maps.

Although classical programming will likely remain the only tool for many tasks and is not going away, and whilst the machine in "machine learning" is a classically programmed algorithm, an increasing number of tasks in tomorrow's software development process are going to be about selecting, extracting, labeling, converting, re-coding, and visualizing data for machine learning solutions run on ready-made cloud services [23]. It is already apparent that major cloud computing providers can offer extremely powerful and flexible off-the-shelf machine learning solutions where classical programming often plays, if anything, a minor role. With advances in development suites and intuitive interfaces, powerful machine learning solutions can be developed without programming them. We can only expect those tools to become less programming intensive in the future. What job markets need in the era of automation of knowledge work is people who are able to work with development environments, scripts, and data-driven solutions. This requires an understanding of machine learning principles, information flows, statistical thinking, and most of all, domain knowledge. What employers are increasingly looking for is not programmers but a combination of subject matter expertise and expertise in data science.

> CT focuses on rule-driven programming in an era where data-driven machine learning is becoming the most influential disrupter of knowledge-work markets.

V. Discussion

It is not our intention to say that programming (or programming education) is becoming obsolete. Programming is a crucial skill for a range of jobs, and it will be a great asset in many fields. It will provide a good basic vocabulary for technology education. However, with automation of programming tasks and advances in development environments, the very nature of programming is changing, and the number of classical programmers, skilled in the imperative, control-flow paradigm, required is not growing at the same pace as other computing specializations. Other skills such as design, data literacy, and social and human interaction skills are becoming all the more important. Imperative programming is no longer, and indeed never was, the only skill for controlling computers and information flows. If we wish to prepare today's children for the next thirty years of the information era instead of the past thirty years, we should be careful to look into the future.

As much as education has recognized the importance of computational thinking, there seems to be a widening gap between the complex CT design challenges and the narrow problem-solving capabilities that are promoted in prevailing educational practices [11]. Earlier models that have examined CT in education can provide complementary perspectives, but skills such as data-driven design, machine learning, data agency, and social and emotional intelligence are becoming increasingly central to future innovation and working life. Computer scientists are contributing to that shift by building ever better tools to push the design challenge to ever higher levels of abstraction: software development suites, automatic programming, cloud-based development environments, and so forth. To support a dialogue focused on the educational mission, this paper has identified eight potentially troublesome issues for educators to consider.

In this regard, there is an ever-increasing need for skills to harness these evolving technologies to capitalize on the strengths of human cognition as well as to help overcome its weaknesses in information processing [25]. If we want to teach our children the future of computing and knowledge work, we must revise and expand how we conceptualize the emergence of the new skills, which support our progress into the era of the automation of knowledge work.

Regarding the ways of living and working in a world pervaded by computing systems, it is evident that teachers and teacher educators play a key role in preparing future generations to understand how their world works and how to

use technology to explore the world. In such a world, new skills have the potential to create new participatory divides [21]. Such inequalities shape who will succeed and who will be left behind as coming generations enter further studies and working life. The new digital divides are not about access, and possibly not even skills as much as they are a divide between those with strong data agency [42] and those who have to adopt or reject technology by blind faith. More broadly, advancing data agency seeks to promote progressive social change in which citizens have the capacity and volition for informed decisions and actions that make a difference in their digital world [42]. Yet, hitherto there has been relatively little research designed to inform initiatives that help school teachers and students develop those skills and social practices that characterize modern CT and the new kinds of design-intensive work in general.

VI. Conclusion

CT must expand to embrace, once again, paradigms beyond those focused on sequential imperative programming. The history of CT and programming education are rich with competing programming paradigms, languages, and visualization and teaching aids. There is work on new entrants to computing education, such as abstraction, data representations and habits of mind associated with data driven classification approaches to AI and ML. However, we must broaden our thinking beyond consideration of control flow, replacing the focus on sequences, loops and if statements that currently feature prominently in the literature as important aspects of CT.

One of the most serious problems we identify, and the main contribution of the paper, is that the literature has hitherto defined CT in such a way as to exclude much of what computing has traditionally been about. The rich flora of computational paradigms and alternative computational hardware approaches have been reduced to a focus on a small part of the computing discipline, namely imperative programming. Rather than empowering citizens and education in computing, the CT conceptualization has built a box around us, restricting our view of computing and computing education. Replicating this into primary and secondary education exacerbates the situation, limiting the world view of computing as it is implemented in school curricula. The dominant discourse of imperative programming has been so pervasive that the CT community has been unable to see beyond it, like a fish unaware of the water in which it swims.

Finally, curriculum must expand to include concepts associated with data-transmission in cooperating systems, classification and training of ML solutions and the computational strengths and weaknesses of these approaches. Parallelism and concurrency, ideas of data-flow and non-determinism, all critical to understanding what computation and automation can realistically deliver in terms of computationally driven solutions. Only then will CT become relevant to future citizens.

Acknowledgments

The authors would like to thank Peter J. Denning for comments and critique of an early version of the paper, originally accepted to SITE conference in 2020 but cancelled due to COVID-19 travel restrictions to New Orleans. The authors wish to thank January Collective for insightful ideas on this paper. However, the authors bear responsibility for any errors or incorrect interpretations. Arnold Pears contributions to this work are supported by ERASMUS+ project grant 2019-1-LT01-KA203-060767

References

[1] B. W. Arden, Ed., *What Can Be Automated? Computer Science and Engineering Research Study*. Cambridge, MA, USA: The MIT Press, 1980.

[2] M. Armoni, "Computer science, computational thinking, programming, coding: The anomalies of transitivity in k–12 computer science education," *ACM Inroads*, vol. 7, no. 4, pp. 24–27, 2016.

[3] V. Barr and C. Stephenson, "Bringing computational thinking to K-12: What is involved and what is the role of the computer science education community?" *ACM Inroads*, vol. 2, no. 1, pp. 48–54, 2011.

[4] K. W. Beyer, *Grace Hopper and the Invention of the Information Age*. Cambridge, MA, USA: The MIT Press, 2009.

[5] P. Blikstein and M. Worsley, "Children are not hackers: Building a culture of powerful ideas, deep learning, and equity in the maker movement," in *Makeology: Makerspaces as Learning Environments*, K. Peppler, E. Rosenfeld Halverson, and Y. B. Kafai, Eds. New York, NY, USA: Routledge, 2016, vol. 1.

[6] E. Brynjolfsson and A. McAfee, *The Second Machine Age: Work, Progress, and Prosperity in a Time of Brilliant Technologies*. New York, NY, USA: W. W. Norton & Company, 2014.

[7] A. Darwiche, "Human-level intelligence or animal-like abilities?" *Communications of the ACM*, vol. 61, no. 10, pp. 56–67, 2018.

[8] P. J. Denning, "Fifty years of operating systems," *Communications of the ACM*, vol. 59, no. 3, pp. 30–32, 2016.

[9] ——, "Remaining trouble spots with computational thinking," *Communications of the ACM*, vol. 60, no. 6, pp. 33–39, 2017.

[10] 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.

[11] P. J. Denning and M. Tedre, *Computational Thinking*. Cambridge, MA, USA: The MIT Press, 2019.

[12] E. W. Dijkstra, "The humble programmer," *Communications of the ACM*, vol. 15, no. 10, pp. 859–866, 1972.

[13] Editors of DATA-LINK, "What's in a name?" *Communications of the ACM*, vol. 1, no. 4, p. 6, 1958.

[14] N. Ensmenger and W. Aspray, "Software as labor process," in *History of Computing: Software Issues*, U. Hashagen, R. Keil-Slawik, and A. Norberg, Eds. Berlin / Heidelberg, Germany: Springer-Verlag, 2002, pp. 139–165.

[15] N. L. Ensmenger, "The 'question of professionalism' in the computer fields," *IEEE Annals of the History of Computing*, vol. 23, no. 4, pp. 56–74, 2001.

[16] J. H. Fetzer, "Program verification: The very idea," *Communications of the ACM*, vol. 31, no. 9, pp. 1048–1063, 1988.

[17] S. Grover and R. Pea, "Computational thinking: A competency whose time has come," in *Computer Science Education: Perspectives on Teaching and Learning in School*, S. Sentance, E. Barendsen, and C. Schulte, Eds. London, UK: Bloomsbury Academic, 2018, pp. 19–37.

[18] J. Grudin, "The computer reaches out: The historical continuity of interface design," in *CHI '90: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA: ACM, 1990, pp. 261–268.

[19] 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.

[20] G. M. Hopper, "Keynote address," in *History of Programming Languages*, R. L. Wexelblat, Ed. New York, NY, USA: Academic Press, 1981, pp. 7–20.

[21] H. Jenkins, R. Purushotma, M. Weigel, K. Clinton, and A. J. Robison, *Confronting the Challenges of Participatory Culture: Media Education for the 21st Century*, ser. The John D. and Catherine T. MacArthur Foundation Reports on Digital Media and Learning. Cambridge, MA, USA: The MIT Press, 2009.

[22] 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.

[23] A. Karpathy. (2017, November) Software 2.0. [Online]. Available: https://medium.com/@karpathy/software-2-0-a64152b37c35

[24] K. Kelly, *The Inevitable: Understanding the 12 Technological Forces That Will Shape Our Future*. New York, NY, USA: Penguin Books, 2017.

[25] K. Lonka, L. Hietajärvi, M. Moisala, H. Tuominen-Soini, L. J. Vaara, K. Hakkarainen, K. Salmela-Aro, V. Cho, and A. Steiner. (2015) Report for EU parliament 2015: Innovative schools: Teaching & learning in the digital era: Workshop documentation.

[26] S. Y. Lye and J. H. L. Koh, "Review on teaching and learning of computational thinking through programming: What is next for K–12?" *Computers in Human Behavior*, vol. 41, pp. 51–61, 2014.

[27] D. MacKenzie, *Mechanizing Proof: Computing, Risk, and Trust*. Cambridge, MA, USA: The MIT Press, 2001.

[28] L. Mannila, V. Dagienė, 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.

[29] E. Nardelli, "Do we really need computational thinking?" *Communications of the ACM*, vol. 62, no. 2, pp. 32–35, 2019.

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

[31] S. Papert and I. Harel, "Situating constructionism," in *Constructionism*, S. Papert and I. Harel, Eds. Ablex Publishing Corporation, 1991, vol. 36, no. 2, pp. 1–11.

[32] A. N. Pears, "Developing computational thinking, "fad" or "fundamental"?" *Constructivist Foundations*, vol. 14, no. 3, pp. 410–412, 2019.

[33] A.-K. Peters and A. Pears, "Students' experiences and attitudes towards

[43] J. Voogt, P. Fisser, J. Good, P. Mishra, and A. Yadav, "Computational thinking in compulsory education: Towards an agenda for research and practice," *Education and Information Technologies*, vol. 20, no. 4, pp. 715–728, 2015.

learning computer science," in *2012 Frontiers in Education Conference Proceedings*, 2012, pp. 1–6.

[34] M. Priestley, *A Science of Operations: Machines, Logic and the Invention of Programming*. London, UK: Springer-Verlag, 2011.

[35] A. Ralston, "Computer science, mathematics, and the undergraduate curricula in both," *The American Mathematical Monthly*, vol. 88, no. 7, pp. 472–485, Aug. - Sep. 1981.

[36] A. Ralston and M. Shaw, "Curriculum '78–is computer science really that unmathematical?" *Communications of the ACM*, vol. 23, no. 2, pp. 67–70, 1980.

[37] M. Resnick, *Lifelong Kindergarten: Cultivating Creativity through Projects, Passion, Peers, and Play*. Cambridge, MA, USA: The MIT Press, 2017.

[38] M. Resnick and E. Rosenbaum, "Designing for tinkerability," in *Design, Make, Play: Growing the Next Generation of STEM Innovators*, M. Honey and D. E. Kanter, Eds. New York, NY, USA: Routledge, 2013, pp. 163–181.

[39] M. Saqr, K. Ng, S. S. Oyelere, and M. Tedre, "People, ideas, milestones: A scientometric study of computational thinking," *ACM Transactions on Computing Education*, vol. 21, no. 3, 2021.

[40] J. Sorva, "Notional machines and introductory programming education," *ACM Transactions on Computing Education*, vol. 13, no. 2, pp. 8:1–8:31, 2013.

[41] M. Tedre and P. J. Denning, "The long quest for computational thinking," in *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*, ser. Koli Calling '16. New York, NY, USA: ACM, 2016, pp. 120–129.

[42] M. Tedre, H. Vartiainen, J. Kahila, T. Toivonen, I. Jormanainen, and T. Valtonen, "Machine learning introduces new perspectives to data agency in K–12 computing education," in *2020 IEEE Frontiers in Education Conference (FIE)*, 2020, pp. 1–8.

[44] P. Wegner, "Three computer cultures: Computer technology, computer mathematics, and computer science," in *Advances in Computers*, F. L. Alt and M. Rubinoff, Eds. Elsevier, 1970, vol. 10, pp. 7–78.

[45] G. Wells, "Dialogue, inquiry, and the construction of learning communities," in *Transforming Learning in Schools and Communities: The Remaking of Education for a Cosmopolitan Society*, B. Lingard, J. Nixon, and S. Ranson, Eds. London, UK: Continuum Publishing Group, 2008, pp. 236–256.