Logic, Code, and the History of Programming

Mark Priestley

IEEE Annals of the History of Computing, Volume 43, Number 4,
October-December 2021, pp. 92-96 (Article)

Published by IEEE Computer Society

DEPARTMENT EDITOR: Colette Perold, colette.perold@colorado.edu

THINK PIECE

Logic, Code, and the History of Programming

Mark Priestley, *The National Museum of Computing, Bletchley Park, U.K.*

A striking feature of the debates around the perceived software crisis in the 1960s and 1970s is the frank contempt expressed by some elite computer scientists for much work in the fields of programming and programming language design. The writings of computer scientist Edsger Dijkstra are a familiar source of such material: in his Turing Award lecture, he opined that "the sooner we can forget that FORTRAN ever existed the better" and likened an advocate of the PL/I language to a drug addict [1]. In a slightly more restrained register, John Backus (another Turing Award winner) used his acceptance speech to denounce existing languages as "fat and flabby [2]." Dijkstra's contempt for the tools of his trade easily slipped into contempt for their users. For example, he described software engineering as the "doomed discipline" whose charter is "how to program if you cannot," and BASIC programmers as "mentally mutilated beyond hope of regeneration [3]."

Such comments often frame visions of different styles of language and approaches to programming. Backus' critique was a prologue to a presentation of a new system of functional programming, and Dijkstra was a career-long advocate of small languages and a rigorous approach to program development. Both represent a tradition within computer science that sees programming as an unruly and uncontrollable activity that requires disciplining [4]. In the broadest terms, this tradition aims to subordinate programming to the logico-mathematical activities of axiomatization and proof. Ideally, one should program by writing a formal specification of a problem and then formally deriving code from this specification. This is a perfectly reasonable research program within computer science, of course, but also an ideal that characterizes only a tiny fragment of the programming activity that has taken place since the computer was invented.

1058-6180 © 2022 IEEE
Digital Object Identifier 10.1109/MAHC.2021.3127289
Date of current version 7 February 2022.

In this essay, I argue that instead of seeing coding as subordinate to logic, both should be understood as instances of the more general activity of working with formal symbolic notations [5]. In this perspective, programming appears as an autonomous activity, and I conclude by arguing that an appreciation of this autonomy is necessary for writing an adequate history of programming.

SURPRISING DIFFICULTY OF PROGRAMMING

At the moment of the emergence of the automatic high-speed general-purpose digital computer, both Alan Turing and John von Neumann characterized programming as a new form of logic [6]. This highlighted the distinction between the parts of the machine that carried out arithmetical operations and the parts that dealt with the sequencing of those operations, often referred to as the "logical control." The usage also appeared to situate coding within the intellectual space opened up by the development of symbolic logic in the early twentieth century. However, as computer pioneer Arthur Burks pointed out in 1950, traditional logic dealt only with declarative sentences that could be true or false, whereas machine-language programs were made up of imperatives [7]. To Burks, the relevance of logic to programming lay not in its account of validity and proof, but rather in its detailed analysis of the syntax of formal languages.

Turing and von Neumann also saw a need to develop a technique of programming, at a time when the idea that programming might be difficult came as a surprise to many [8]. The ENIAC developers recognized that new numerical methods would be needed for high-speed calculation, but they defined the sequences of operations that ENIAC should execute in a format very similar to that used by Charles Babbage and Ada Lovelace a century before [9]. In common with other workers, such as Howard Aiken's group at Harvard, they do not seem to have considered that writing instructions for an automatic machine would be a problem. For many years, large-scale manual computation had utilized a division of

labor between those who planned the work and those who carried it out by performing simple arithmetical operations and filling in boxes in highly structured computation sheets [10]. It seemed a straightforward task to replace the latter group by the new machines.

However, when the renowned mathematician and expert calculator Douglas Hartree set up the solution to a system of differential equations on ENIAC in 1946, the machine's response to an unexpected situation surprised him [11]. The calculation involved a variable *r*, which was known from its physical properties to be always positive and was stored as an integer in the range 0–99. However, at one point in the computation *r* unexpectedly took on the value –1. ENIAC interpreted this as the complement, 99, and continued computing with this erroneous value. Hartree, speaking only a few weeks after the event, was evidently quite shaken by it, saying: "I saw that I ought to have foreseen the possibility of this situation arising [. . .] I didn't foresee it despite thirty years of experience in computing of various kinds [12]."

Hartree attributed this breakdown to a qualitative difference between human and machinic agency. The planners of large computations could make allowances for the familiar sorts of mistakes that a human computer might make and the checks they could be expected to carry out, but "the machine may make quite different kinds of mistakes and has not got the same experience and knowledge to hand" for resolving them. While the machine could "exercise some degree of judgment and discrimination," the occasions for this "have all to be foreseen in the program of operating instructions furnished to the machine [13]." Hartree returned to this example in several lectures and publications over the following years and sloganized it by stressing the need for programmers to take what he called "the machine's-eye view" when coding a problem.

History suggests that this breakdown was not unprecedented, however. Since at least the time of Leibniz, formal manipulation of symbols had been metaphorically described as a "mechanization" of thought, and computer programming was far from being the first occasion when machinic interpretation of a symbolic calculus had caused surprise [14]. In the late 1700s, for example, John Playfair described as a "paradox" the fact that manipulation of the apparently meaningless symbols referring to imaginary numbers, or "impossible quantities" as they were suggestively called, could lead to results that appeared to be not only formally correct but also practically applicable. At the start of the twentieth century, the project of developing mathematical logic was shaken to its foundations by the discovery of the paradoxes that could be derived from the apparently innocent assumptions of naïve set theory and logic. A few years later, Kurt Gödel's stunningly unexpected demonstration of the incompleteness of logical systems cast doubt on the whole enterprise of logicism [15].

On the face of it, these breakdowns are puzzling. In the 1940s, several writers described and motivated automatic computers by drawing a systematic analogy between human and machine practices of computation. Turing, for example, described the machine's memory as the analog of "the computing paper on which the [human] computer writes down his results and his rough workings," while the human computer would normally carry "the instructions as to what processes are to be applied [. . .] in his head [16]." The intent behind such descriptions was to make a new technology more familiar; a side-effect was to make it seem natural that human behavior could be described in terms of the technology. But how could the formalization and automation of a familiar human practice lead to such surprises?

The situation looks rather different if we question the assumption that there is anything "natural" or "given" about human calculation. As with any other discipline of mechanization, such as factory work in the Industrial Revolution, humans must be extensively trained to carry out formal calculation, with distinct practices being developed in different fields. In mathematics, for example, formal passages of work can be checked by appealing to the meaning of derived formulas, or by computing a result in different ways. In principle, the same could be done in logic, but in practice large-scale logical proofs were rare. The logical calculi of the early twentieth century were designed with an eye to simplifying metalogical reasoning rather than facilitating proof in an interpreted calculus. There are of course exceptions—there are many proofs in Russell and Whitehead's *Principia Mathematica*, and outliers like J. H. Woodger's use of logic to formalize aspects of biological theory [17]—but overall, extended proofs in the object languages of logic were rarely carried out, and when they were, they were necessarily checked by human readers rather than machines.

In other words, "mechanical" or formal processing of symbolic systems that ignores the meaning of the symbols being manipulated is not an intrinsic characteristic of human thought, but a skill deployed by highly trained individuals in quite restricted fields of work, such as science and accountancy. Automatic digital computers executed formal processes that were orders of magnitude longer than any performed before and revealed with unforgiving clarity the ways in which computation by a real machine differed from "mechanical" computation carried out by humans. This was widely taken as evidence that computer programming would pose a new

and significant challenge, but also highlights the general observation that humans do not intuitively grasp the nature and processes of mechanical "thought." Interestingly, a sense of the opacity of formal systems had surfaced in popular culture even before the computer. Particularly apposite here are Isaac Asimov's early robot stories, some of which dramatized the difference between human and machinic thought and drew attention to the surprises that lurk within even quite simple formal systems. Stories such as "Runaround" explore ways in which Asimov's famous three laws of robotics could have unexpected consequences [18]. The tension in these stories is the drama of debugging, as unpredictable or divergent robot behavior is gradually revealed to be a consequence of the interplay of an apparently simple and transparent set of rules.

AUTONOMY OF PROGRAMMING

The problem of working out how to adopt the "machine's-eye view" and deliver more predictable and reliable programs was exacerbated by an assumption prominent among groups such as those developing the EDSAC and Whirlwind computers at Cambridge University and MIT: that many programs would be written not by experts, but by scientists and engineers wanting to use the powerful new machines without first undergoing an extensive period of training. This assumption addressed the fear, voiced by John Mauchly as early as 1946, that programming would become an economic bottleneck [19]. Electronic machines were so fast that for many problems the time spent computing solutions would be dwarfed by the time required for problem preparation. If the expensive new machines were to be kept busy, coding could not become the preserve of a small cadre of experts.

Von Neumann had doubts about "the ability of the computing mechanism to take our intention correctly" and, like Hartree, emphasized that programmers must "foresee where [the machine] can go astray, and prescribe in advance for all contingencies [20]." Working with Herman Goldstine, he proposed a technique that would separate the development and expression of a plan to "foresee all contingencies" from the machine-specific details of coding. The highly influential presentation given in the *Planning and Coding* reports introduced new notation, representing the plan as an annotated flow diagram from which, it was asserted, coding could be carried out in a routine way [21].

The planning approach, with its desire for total foresight of and control over the course of a computation, came in for criticism in the mid-1950s, as programmers began to address problems more open-ended than simple calculations, such as image recognition, theorem proving, and game playing. Early AI researchers argued for replacing explicit plans with programs that embodied "heuristics," rules that would guide a program through a space of possible solutions to find results that could not have been foreseen by the programmer [22]. An interesting aspect of this proposal was a revaluation of the surprise latent in formal systems, now enlisted as a desirable outcome of computations enabling the discovery of solutions that would otherwise not have been obtainable [23].

Even as the planning approach was gaining momentum, however, programmers and computer designers were developing alternative strategies and articulating the field from the inside out. The idea of using subroutine libraries was particularly prominent in the late 1940s, a practice given its canonical description in a textbook written by the EDSAC group at Cambridge University [24]. The book paid lip-service to the planning ideal, stating in familiar language that in programming "every contingency must be foreseen," but its practical recommendations were firmly centered on code, with never a flow diagram in sight. Rather than being planned, programs were to be assembled out of pre-existing subroutines, extensively tested and therefore reliable, coordinated by a relatively small main routine.

David Wheeler's "initial orders"—which squeezed a simple translator, assembler, and loader into 41 instructions—made this ambitious idea workable. Unlike the examples in the *Planning and Coding* reports, Wheeler's program fully embraced and exploited the possibilities and ambiguities opened up by the new style of coding. Certain words in memory were interpreted as numbers or instructions at different stages of execution, and the program rewrote entire instruction words, not just the address field. The initial orders could not have been adequately represented in the flow diagram notation, indicating the extent to which Goldstine and von Neumann sought to constrain the intrinsic possibilities of code in order to achieve the goals of predictability and correctness.

The tradition that began with the *Planning and Coding* reports represents a wager that the difficulty of programming is something specific to code, rather than a general feature of symbolic systems brought to prominence by the speed of the new machines. This tradition aims to discipline coding by wrapping it within a process-oriented methodological framework and subordinating it to a variety of diagrammatic and logico-mathematical techniques. Proponents hoped that these techniques would prove more tractable than code itself [25]. At the same time, however, developments internal to programming led to a rich body of new knowledge and technique

that, in practice, owed little to formal logic. The opposition between these approaches remains unresolved, as recent arguments made for practices of agile software development have illustrated [26].

CONCLUSION

What implications does this have for the writing of history? Code is the hinge on which the pervasive influence of computers in the contemporary world turns, and yet the historiography of software is far less well developed than that of hardware. The place of software within the history of computing was discussed at a pivotal meeting in 2000 where delegates characterized the existing literature as being too technical and too focused on individual languages and applications [27]. They called instead for wider narratives that would set software in a range of contexts, including business and economic history and the history of engineering. The implementation of this program, however, has left some feeling that technical history has been neglected, a view memorably expressed by Donald Knuth [28]. But the idea of a sharp divide between the technical and the non-technical has been questioned by scholars for many years, and there is no reason why historians should not follow programmers and language designers through the simultaneous technical, social, and political contexts within which they operate [29].

I have argued in this essay that an influential line of thought in computer science proposes that programming can essentially be reduced to longer-established forms of symbolic work, particularly formal logic. Coupled with the prominence of the machines themselves, this has made it difficult for historians to see programming as a significant autonomous activity that generates and deploys distinctive forms of knowledge and practice. But the relationship between programming and logic develops in the real time of history and is not something that can be laid down in advance by theoreticians [30]. Liberation from *a priori* characterizations of programming will enable historians to gain a more realistic and comprehensive view of its development and draw together the rich, albeit somewhat fragmented, existing literature.

Computer scientist Mary Shaw has described such characterizations as "myths," countering them with more empirical descriptions that provide useful starting points for historians [31]. She points out that software systems are not simply symbolic program texts but complex technological artefacts, coalitions of components and tools simultaneously situated in many different contexts and embodying multiple layers of historicity [32]. Developing a form of "software archaeology" to study these artefacts on their own terms would allow scholars to further develop themes important in the wider history of technology, such as the study of the use and evolution of technological artefacts, and to make connections with related work in disciplines such as critical code studies and media archaeology.

We can push this line of thought to the limit by asking, what is the history of programming the history *of*? Asking a parallel question about the history of science, Peter Dear noted that historians had turned to "naturalistic" studies of ideas, practices, and institutions [33]. This tendency is also visible in the history of computing, where historians are actively exploring the diversity of cultures of programming and contexts of use, an important step toward rescuing code and its history from the condescension of computer scientists such as Dijkstra [34]. But until historians examine the history of programming in the round, free from the disciplinary assumptions of computer science, the broader question will remain unanswered.

ACKNOWLEDGMENTS

The work on this article has been greatly stimulated by the author's participation in the ANR project "PROGRAMMme" (ANR-17-CE38-0003-01). The author would like to thank Liesbeth de Mol for invitations to join the project and to contribute to this Special Issue. He also thanks Gerardo Con Díaz for extensive editorial assistance which has greatly improved the text.

REFERENCES/ENDNOTES

[1] E. W. Dijkstra, "The humble programmer," *Commun. ACM*, vol. 15, no. 10, pp. 859–866, 1972.

[2] J. Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs," *Commun. ACM*, vol. 21, no. 8, pp. 613–641, 1978.

[3] E. W. Dijkstra, *Selected Writings on Computing: A Personal Perspective*. Hoboken, NJ, USA: Prentice-Hall, 1982, pp. 129–131; E. W. Dijkstra, "On the cruelty of really teaching computer science," *Commun. ACM*, vol. 32, no. 12, pp. 1398–1404, 1989.

[4] "Discipline" was an important word for Dijkstra. In his 1972 lecture, he repeatedly referred to the discipline required of a programmer, and it provided the title of his book *A Discipline of Programming*. Hoboken, NJ, USA: Prentice-Hall, 1976.

[5] I use the two terms "coding" and "programming" (somewhat loosely) to distinguish the largely formal activity of working with programming notations from the larger body of practice within which it is set.

[6] Strings of adjectival qualifiers were frequently used to characterize the new machines. For more on this, and the later introduction and use of the problematic term "stored-program computer," see T. Haigh, M. Priestley, and C. Rope, "Reconsidering the stored-program concept," *IEEE Ann. Hist. Comput.*, vol. 36, no. 1, pp. 4–17, Jan.–Mar. 2014. For programming considered as logic, see, for example, A. M. Turing, "Lecture to the London Mathematical Society," 1947; reprinted in B. E. Carpenter and R. W. Doran, *A. M. Turing's ACE Report of 1946 and Other Papers*. Cambridge, MA, USA: MIT Press, 1986, pp. 106–124, on p. 122; H. H. Goldstine and J. von Neumann, *Planning and Coding of Problems for an Electronic Computing Instrument,* vol. 1. Princeton, NJ, USA: Inst. Adv. Study, 1947, p. 2.

[7] A. W. Burks, "The logic of programming electronic digital computers," *Ind. Math.*, vol. 1, no. 36–52, p. 39, 1950.

[8] Maurice Wilkes gave a well-known description of the moment he came to realize this. See M. Wilkes, *Memoirs of a Computer Pioneer*. Cambridge, MA, USA: The MIT Press, 1985, p. 145.

[9] For ENIAC details, see T. Haigh, M. Priestley, and C. Rope, *ENIAC in Action*. Cambridge, MA, USA: MIT Press, 2014, ch. 2.

[10] Large-scale manual computation at the time of the development of the computer is vividly described in D. A. Grier, "The Math Tables Project of the Work Projects Administration: The reluctant start of the computer era," *IEEE Ann. Hist. Comput.*, vol. 20, no. 3, pp. 33–50, Jul.–Sep. 1998.

[11] Hartree described this episode in a lecture in July 1946 entitled "Some general considerations in the solutions of problems in applied mathematics." See M. Campbell-Kelly and M. R. Williams, *The Moore School Lectures: Theory and Techniques for Design of Electronic Digital Computers*. Cambridge, MA, USA: The MIT Press, 1985, pp. 53–72; see, in particular, pp. 63–65.

[12] D. Hartree, p. 65.

[13] D. Hartree, pp. 63–65.

[14] For more details, see M. Priestley, *A Science of Operations*. New York, NY, USA: Springer, 2011, sec. 1.4.

[15] T. Franzén, *Gödel's Theorem: An Incomplete Guide to Its Use and Abuse*. Boca Raton, FL, USA: CRC Press, 2005.

[16] A. Turing, "Proposed electronic calculator," reprinted in B. E. Carpenter and R. W. Doran, *op. cit.*, 1946, p. 20.

[17] J. H. Woodger, *The Axiomatic Method in Biology*. Cambridge, U.K.: Cambridge Univ. Press, 1937.

[18] I. Asimov, "Runaround," *Astounding Sci.-Fiction*, vol. 29, no. 1, pp. 94–103, 1942.

[19] J. W. Mauchly, "Digital and analog computing machines," M. Campbell-Kelly and M. R. Williams, *op. cit.*, 1946, pp. 25–40.

[20] J. von Neumann, "Electronic methods of computation," *Bull. Amer. Acad. Arts Sci.*, vol. 1, no. 3, pp. 2–4, 1948.

[21] H. Goldstine and J. von Neumann, "Planning and coding."

[22] For more on this episode, see M. Priestley, "AI and the origins of the functional programming style," *Minds Mach.*, vol. 27, no. 3, pp. 449–472, Sep. 2017.

[23] The revaluation of surprise and intertwining of human and machinic agency in heuristic systems is described in the context of the AURA theorem prover in S. Dick, "AfterMath: The work of proof in the age of human-machine calculation," *Isis*, vol. 102, no. 3, pp. 494–505, 2011.

[24] M. V. Wilkes, D. J. Wheeler, and S. Gill, *The Preparation of Programs for an Electronic Digital Computer*. Boston, MA, USA: Addison-Wesley, 1951.

[25] For details, see, C. B. Jones, "The early search for tractable ways of reasoning about programs," *IEEE Ann. Hist. Comput.*, vol. 25, no. 2, pp. 26–49, Apr.–Jun. 2003.

[26] K. Beck *et al.*, *Manifesto for Agile Software Development*, 2001. Accessed: Sep. 30, 2021. [Online]. Available: http://agilemanifesto.org

[27] U. Hashagen, R. Keil-Slawik, and A. L. Norberg, Eds., *History of Computing: Software Issues*. New York, NY, USA: Springer, 2000.

[28] D. E. Knuth, "Let's not dumb down the history of computer science," *Commun. ACM*, vol. 64, no. 2, pp. 33–35, 2021.

[29] Classic early descriptions of the intertwining of contexts can be found in W. E. Bijker, T. P. Hughes, and T. Pinch, Eds., *The Social Construction of Technological Systems*. Cambridge, MA, USA: MIT Press, 1987. For an overview of one influential approach, see B. Latour, *Reassembling the Social: An Introduction to Actor-Network Theory*. Oxford, U.K.: Oxford Univ. Press, 2005.

[30] For "real-time understanding of practice," see A. Pickering, *The Mangle of Practice: Time, Agency, and Science*. Chicago, IL, USA: Univ. Chicago, 1995.

[31] M. Shaw, "Myths and mythconceptions: What does it mean to be a programming language, anyhow?," *HOPL IV: 4th ACM SIGPLAN Hist. Program. Lang. Conf.*, 2021, (keynote talk). Accessed: Oct. 17, 2021. [Online]. Available: https://www.pldi21.org/prerecorded_hopl.K1.html

[32] The intrinsic historicity of software systems has recently been emphasized in K. Tracy, *Software: A Technical History*. New York, NY, USA: ACM Books, 2021.

[33] P. Dear, "What is the history of science the history *of*?," *Isis*, vol. 96, no. 3, pp. 390–406, 2005.

[34] BASIC, in particular, is rehabilitated in J. L. Rankin, *A People's History of Computing in the United States*. Cambridge, MA, USA: Harvard Univ. Press, 2018, ch. 2.

MARK PRIESTLEY is currently a Senior Research Fellow at the U.K.'s National Museum of Computing, Bletchley, U.K. He works on the history and philosophy of computing, with a focus on programming in the 1940s and 1950s. Contact him at m.priestley@gmail.com.