To Write Code: The Cultural Fabrication of Programming Notation and Practice

Ian Arawjo
Department of Information Science
Cornell University
Ithaca, NY, USA
iaa32@cornell.edu

ABSTRACT

Writing and its means have become detached. Unlike written and drawn practices developed prior to the 20th century, notation for programming computers developed in concert and conflict with discretizing infrastructure such as the shift-key typewriter and data processing pipelines. In this paper, I recall the emergence of high-level notation for representing computation. I show how the earliest inventors of programming notations borrowed from various written cultural practices, some of which came into conflict with the constraints of digitizing machines, most prominently the typewriter. As such, I trace how practices of "writing code" were fabricated along social, cultural, and material lines at the time of their emergence. By juxtaposing early visions with the modern status quo, I question long-standing terminology, dichotomies, and epistemological tendencies in the field of computer programming. Finally, I argue that translation work is a fundamental property of the practice of writing code by advancing an intercultural lens on programming practice rooted in history.

Author Keywords

programming; notation; culture; materiality; infrastructure

INTRODUCTION

> One historical particularity is generalised into a timeless and spaceless universality... shifting the focus of particularity to a plurality of centres, is a welcome antidote.
> – Ngũgĩ wa Thiong'o [135]

When we hear of computer programming, we think of a posture and a place, a culture, a normalized and designed "way of being" [137]. This dominant culture and its assumptions become entrenched in discourse: the Oxford dictionary defines a programmer as "a person who writes computer programs," leaving the means of writing open to interpretation [42]; the computing discipline mobilizes metaphors of language, conjuring linearity and literacy [101, 130]. In fact, "to write programs" today almost always implies a very specific assemblage, regardless of geography, involving the keyboard, screen, mouse or trackpad, and practices tied to the affordances of these devices. A quick type in a search engine reflects the dominant vision of the modern programmer: hunched over a screen, typing small lines of print.

Our imaginations are conditioned by sociotechnical status quo; over time, as assemblages stabilize and erect infrastructure in their image, we struggle to imagine any alternative [37, 114, 91]. Assemblages become socially "normalized" and materially "naturalized," entrenched in a design initially stabilized around different technical, cultural, and historical circumstances, sometimes regardless of the proficiency of the design years later [77, 19]. This status quo becomes engrained in hearts and minds, in professionalization practices, in organizations, and is reproduced unwittingly by new designs [114, 103]. As researchers with postcolonial sensibilities know only too well, those looking to create or account for alternative ways of being and knowing often struggle with the need for "translations" from their (and their readers') encultured perspective [131, 23, 124]. Yet despite writing code's central importance to the creation of new forms of HCI, there is a significant lack of sustained reflection on how this commonplace practice was historically constructed with prior ready-to-hand infrastructure and cultural practices [62].

This paper traces the sociomaterial fabrication of early computer programming notation and practice: of how 'to write code' came to imply typing characters in text editors and terminals, rather than (for example) a practice involving handwriting or drawing. Through three case studies of the earliest visions of 'writing code,' I recall the emergence of high-level[1] programming notation and computing's extension of earlier social and material infrastructure of the typewriter and card-based processing. Adopting a cultural-historical sensibility to technological emergence, I trace how early inventors' practices in logic, physics, mathematics, engineering, and art –with their varying, handwritten and drawn notations and sensibilities –informed and directed how they fabricated programming notations and practice. I argue that an initial diversity of

[1]High-level refers to notations that would need to be considerably interpreted into machine code (numbers) in order to run [83]. Assembly code largely serves as a mnemonic device to numeric codes and are excluded.

styles quickly came into conflict with typewriters and card-based pipelines, and that notations were thereby serialized and transformed. Corresponding to this transformation and coordinated with the need to align computer science with formal abstraction to justify the new discipline [125, 34], values became attached to different forms of programming notations that reflect prior modernist dichotomies in North American and European societies between the 'textual' and the 'visual,' the 'written' and the 'drawn' [73], aligning the former with objective authority and the latter with aesthetic choice [106].

By tracing this history, I (provocatively) redefine programming as not just involving the design of algorithms or systems, but often "a problem of mapping from one culture to another" [38, p. 138] –from its very inception, a practice which so often involves *translation work* between representations and which is intimately tied to intercultural conflict, compromise, and innovation. In so doing, I advance an intercultural lens [74] on programming practice as a site of social, material, and epistemological contestation, not just in the design of contemporary software or for those outside the societies where computing emerged, but embedded in the design and history of the very tools and practices which support the development of software, and the communities and discourses which have formed around them. In the process, I join other scholars [8, 23, 90] in seeking to "move the centre" [135] in discourse around programming, making, and HCI more broadly towards a plurality of cultural perspectives and practices, while leaning away from overly simplistic rhetoric of the 'West' that denies inner heterogeneity.

To begin, I contend with why 'writing code' deserves attention, when there have been numerous attempts –such as tangible programming or direct manipulation languages [94, 71] –to reconstitute programming practice. To demonstrate how even the most radical visions of programming can end up recentring a typewritten status quo, I offer the following contemporary anecdote.

THE PARADOX OF CHANGE IN HCI

In 2017, a group of artists, designers, and software engineers, led by former Apple designer Bret Victor, came together in Oakland, California to forge a new vision for the future of computer programming. Their goal was "to incubate a humane dynamic medium whose full power is accessible to all people," pledging to liberate professional programming practice from the confines of stuffy offices, isolating screens, and constraining devices. "No screens, no devices" became the project's motto and organizing principle. No longer impersonal and individualized, programming would become communal, social, and democratized. People would "think with their hands, their bodies, spread out, walk around, compare possibilities, improvise, and experiment" [132].

DynamicLand is a room-sized operating system that realizes visions from ubiquitous computing and tangible programming, an impressive achievement by any measure. Yet despite the refrain of "no screens, no devices," DynamicLand's core infrastructure is Lua code printed on pieces of paper. To make alterations to the room-sized program, programmers regularly re-constitute the bottleneck of the shift-key keyboard and associated standards of ASCII symbols laid out in left-to-right, top-to-bottom sequence on a screen [118]. In order for "all people" to gain access to computing's full power, to liberate themselves from all the screens and devices, the bottleneck of the screen and keyboard again re-formed.

From a broader vantage point, the DynamicLand paradox highlights a growing discomfort with contradictions between utopian rhetoric and technology's actual ability to enact change, whether in education, politics, international development, organizations, or design [126, 108, 90, 114, 119, 103]. In HCI and beyond, wide swaths of people are now wary of failed promises. Contradictions abound: companies who profit off of user attention release tools to monitor attention; the rich, after buying the newest marginal phone upgrade, pay again to have it taken away [70]; teenagers, whose computer use we are told we should be concerned about, protest computer over-exposure [120]; the same billionaires that send their children to 'disconnected' schools pour millions into connected learning programs [126]. As the anthropologist Alexei Yurchak found of the ailing Soviet Union, tech today is "simultaneously eternal and stagnating, vigorous and ailing, bleak and full of promise" [138].

Alongside this growing disillusionment with computing technology is a corresponding de-mystification of its design and professionalization processes. As the field of science and technology studies (STS) has shown, ostensibly technical disciplines are replete with social elements: technologies are socially constructed, co-produced with society, and value-laden [24, 136, 77, 3]. STS perspectives have shed light on programming practice and histories, whether the role of trust and professional vision in data science [106, 105], the marginalization of weavers as programmers in the Apollo program [114], the construction of computing as a science [125, 3, 38], or the framing of coding as a literacy [130]. Postcolonial scholars have also attempted to decentre dominant narratives in the maker movement's rhetoric [90, 8], suggesting that maker practices should be more inclusively framed as "making do": "using the materials and competencies on hand to create objects or processes that aid in everyday life," rather than framed as inherently revolutionary or democratizing [8].

A complementary issue to the goal of decentring dominant narratives is how the tools developed to support programming condition thoughts and imaginations. Past scholarship on the influence of material representations on knowledge construction argues that the structure of (typewritten) notations influence the kinds of problems encountered in their usage [44, p. 8-9]. This argument is suggestive of situated theories of cognition such as cultural-historical activity theory (CHAT), widely applied in CSCW and CSCL[2] and deriving from Russian psychologists Vygotsky and Leont'ev [79, 35]. A CHAT perspective emphasizes the cultural and historical roots of social activity and its mediators and argues that learning relies on gradual 'internalizations' of 'externalized' cultural tools ("such as algebraic notation, a map, or a blueprint" [79, p. 42]). The term culture here operates in a generative, rather

[2]Computer-Supported Cooperative Work and Learning, respectively.

than a taxonomic sense, and is defined by Irani & Dourish as "a lens through which people collectively encounter the world, a system of interpretive signification which renders the world intersubjectively meaningful... [A]n individual may participate in many cultures –cultures of ethnicity, nationhood, profession, class, gender, kinship, and history –each of which, with their logics and narratives, frame the experience of everyday life" [74, p. 2-3]. From these perspectives, programming notations are cultural tools, affording certain thoughts and frames while resisting others.

However, tools and practices for new disciplines like computing do not arise in a vacuum; rather, situated practice extends and appropriates pre-existing culture to new ends. As Pickering argued of physicists, this extension operates through a mangle of practice, a dialectic of resistance and accommodation between humans and machines [109]. Comments in postcolonial-oriented papers can cast programming languages as arbiters of a Western monoculture (e.g., [23, 58]), and while reflecting on the entrenchment of certain values, representations, and assumptions is useful[3] –in the spirit of Ong's contrasting of written and oral societies [102] –it is perhaps too simplistic an argument to accommodate inner heterogeneity. As Dourish notes, in order to "get a grip" on the cultural consequences of technology, we must take seriously its "material specificities" rather than speaking of an amorphous and unexamined presence [43]. What exactly is the cultural heritage of programming notation and practice, beyond the obvious use of English keywords? How did ready-to-hand cultural practices influence the design and development of new ones? And how might the spread and influence of early approaches impact the current field, whether visibly or in deeply held, almost invisible ways? While not claiming to answer these questions definitively, this work aligns itself with a growing number of scholars at CHI and beyond arguing for deeper engagement with HCI's early history [114, 115, 81, 4, 8]. I build on this work by trying, as much as possible, to avoid casting our present-day assumptions onto the earliest history of programming. Unlike other work on programming at CHI focused on end-users, for novices, or otherwise tools or studies to support the typewritten status quo, I call into question here the entrenchment of the dominant regime.

HUMAN-MACHINE INTERACTION BEFORE HCI

To orient ourselves in the past, I briefly outline the state of "human-machine interaction" around the advent of digital computers. Some historians now regard the emergence of digital computing as a period of gradual change, rather than a revolutionary discontinuity [38, 61, 36]. The machine 'computer' extended workflows of industrial data processing at a time when writing was being disassociated from writing 'by hand' and increasingly associated with writing through a discretizing (or "technolinguistic" [97]) mediator [36, 10, 73]. The history of HCI is thus also a history of the "eternal recurrence" [82] of the typewriter, and to this technology I shed some light here.

[3]Importantly, groups outside of Anglo societies have raised these concerns, such as indigenous Hawaiians converting C# to their language [98], Nasser's Arabic programming language Qalb [99], and Nguyen's account of a Vietnamese software community [8].

The dominance of typing is of course a topic many scholars have commented on. Like Latour's crashing of distinctions between the human and nonhuman [87], early twentieth century philosophers framed typewriters as having agency. Nietzsche wrote that typewriters have "fine fingers, to use us," and Heidegger, who believed the hand was "the essential distinction of man," warned that the typewriter "imposes its own use" on humankind, transforming "the relation of Being to man" [82, p. 198-200, 207]. Phenomenologists and cognitive scientists would echo these notions later in their concerns about, for instance, shifts from the level of "paragraph" to "sentence" and shortening attention spans [92, 32]. Recently, the anthropologist Tim Ingold harkened back to these concerns when he called on HCI researchers to imagine an antidote to discretizing technology: "a technologically enhanced sensitivity, brought into the service of hands-on engagement with materials in making, [which] could genuinely enlarge the scope of humanity, rather than further eroding it" [72, p. 124].

The history of the typewriter is some respects reflects warnings about its homogenizing effects. While much has been made (and debated) over the QWERTY layout [19, 39], a less reflected on and even more pernicious case of path dependency is the shift-key mechanism. The Remington II shift-key typewriter was optimized for the lower frequency of capitalized letters in English and first designed by American companies embedded in a left-to-right, top-to-bottom written culture [3]. In *The Chinese Typewriter*, Mullaney shows how non-English writing systems were adapted to the standard with as little modifications as possible in order to lower factory assembly costs. In popular media, the typewriter became a symbol of modernity, a machine whose dissemination brought the coming of civilization. Written cultures became judged by their ability to be consumed by the shift-key style and faced pressures to change or romanize. For instance, Chinese logograms were debased in both global and domestic discourse (e.g., by Mao Zedong), and the Thai language lost two characters due to space limitations (a change which persists today). Moreover, the name 'typewriter' conditioned how people interpreted and imagined other technolinguistic machines; for instance, early Chinese typewriters did not in fact have keys [97].

As the shift-key typewriter monopolized writing and threatened cultural diversity, an intimate symbiosis between the computer and the typewriter became drawn in the theories and metaphors that retroactively [60] came to define the discipline of computing. As a boy, Alan Turing pictured himself inventing typewriters to mitigate his poor handwriting; as an adult he preferred to type than write [82, 68], behavior atypical among many of his mathematical contemporaries.[4] In his seminal 1937 paper on computability, Turing describes a generalized typewriter that prints characters on a page with four extensions: it uses an infinite paper tape; can remember and erase symbols in place; and, inspired by the shift-key, may switch between a variable number of configurations [128, 68]. Turing's later paper on artificial intelligence calls the typewriter interface

[4]As was common at the time, Turing would handwrite in mathematical notation that extended beyond his typewriters' abilities (e.g. [68, p. 356]).

the "ideal arrangement" through which to verify intelligence, echoing the civilizing guise of earlier discourse [129].

At the same time, infrastructures of large-scale data processing came to prominence in U.S. accounting. Dorr Felt's Comptometer and William Burroughs' adding machine products, fitted with keys inspired by typewriters, were "the two most popular sets of devices available in the U.S. at the turn of the [20th] century" and could add and multiply numbers [10, p. 30-1]. For larger businesses or government data processing, Herman Hollerith's punch card machines formed a system for storing, tabulating, and sorting data [38] (prominent manufacturers included Hollerith's International Business Machines (IBM) and Remington Rand). As Aspray notes, these three "legs" of computing –typewriters, adders, and punch card processing –were already stabilized for around three decades before the advent of electronic computers [10]. The "operators" of these machines –typists, clerks, secretaries –were largely and increasingly women. Only the advent of World War II brought additional attention on applied mathematics and temporarily destabilized gender roles in the data processing industry [3, 121]. By the advent of digital computers, the social elements of computing –routinization and the division of labour, and feminization of the workforce –had long been in place [84].

THE CULTURE IN EARLY PROGRAMMING NOTATIONS: THREE VISIONS

According to a report by Knuth & Pardos [83], the two first high-level programming notations emerged around the mid-1940s in Germany and at the ENIAC project in the United States. Several years later emerged several typewritten visions of programming in Europe and North America. Here I explore these earliest acts of HCI design through lenses of materiality and situated, cultural perspectives on knowledge construction.

Konrad Zuse's Vision

During the cold opening months of 1945, as Allied planes bombed Berlin, the German inventor Konrad Zuse huddled in safety with his family. His inventions the Z1-3 computers lay blown to pieces in the ruins of his center-city workshop. He secured a truck to transport his wife, assistants, and sole remaining computer, the several-ton Z4, out of the city. Reaching the alpine village of Hinterstein over 650km south, Zuse setup the Z4 in a barn, but found the machine was broken. Secluded from the world with no means to continue construction on computers, he put pen to page, seeking a "universal formula language" for computation as an extension of his dissertation [141, p. 212]. This language he named the Plankalkül, the first high-level programming notation [83, 61].

All of Zuse's prior training and interests accompanied him to Hinterstein. As a photographer and artist, he had painted posters, wrote poetry, and acted and directed theatre, performing as "unknown inventors or artists" [141, p. 26]; as an engineer and inventor, he was trained in mathematics, formal logic, and civil engineering. He combined these seemingly disparate interests in efforts such as applying descriptive geometry to the optimal viewing of artistic work, or using punch cards to automate photography dark room processes [141, p. 17-19, 28]. Zuse seemed to relish the benefits technology of writing entailed, and became agitated when they were suppressed. In college, he switched majors two times in upset over drafting classes, stating that they "had shattered my illusions. The creative spirit was left little freedom in the manner of presentation; everything was standardized, everything was decided: the line thickness, manner of dimensioning, even the positioning of dimension figures" [141, p. 15]. While living in Hinterstein, he continued practicing art by engraving scenes into wood blocks.

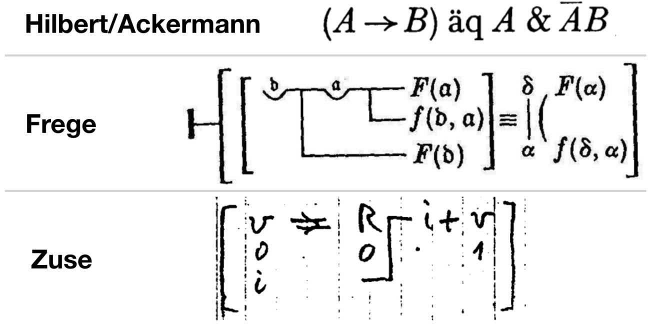

Figure 1: Excerpted notations of German logicians Hilbert/Ackermann and Frege, whose works Zuse studied. Below, a handwritten excerpt from Zuse's 1945 Plankalkül notebook, showcasing the *Zeilenverschiebung*, or 'line shift' notation [139]. *From original texts* [67, 53, 140].

In creating the Plankalkül, Zuse drew from his artistic disposition and passionate interest in formal logic. Of the former, he never appeared to question coding as involving drawn lines and written notation. Of the latter, he dreamt of the universe as "a giant computing machine" and became obsessed with visions of a "mechanical brain," what today we would call general artificial intelligence. This analytic brand of philosophy he likely inherited from German logicians whose works he studied closely, notably David Hilbert, Wilhelm Ackermann, and Gottlob Frege [141, p. 44-6,83,105]. Acknowledging the limits of numeric computation, he designed his 'calculus' around propositional and predicate logic in order to solve chess problems [83].

Frege's work deserves some comment here. European mathematicians at this time preferred the aesthetics of linear sequences, and, if they were aware of Frege's work, frowned upon his two-dimensional notation (Figure 1). For instance, historian Florian Cajori called it "repulsive" [31] and logician Ernst Schröder "ridiculed" it as Japanese [95], hinting at xenophobic underpinnings behind some aesthetic judgements. But Zuse did not seem deterred. What today would be considered a single "line" of code is expressed across three rows as a matter of routine: the first row commonly including variables, the second denoting subscripts, and the third types [113]. Zuse remarks on the ease of "drawing a line" across multiple rows of characters to connect indices and liberally adopted notation[5] from mathematics and formal logic [141, p. 219]. Later

[5]Notation included the square root $\sqrt{}$, power ^{n}, times $\times$, infinity ∞, dot $\cdot$, delta Δ, Greek letters σ, ϕ, τ, ε; logic notation included $\wedge$ and

scholars, in reflecting on the Plankalkül, commented that the notation was "clumsy," "unorthodox," and puzzling [21, 113], reflecting a similar aesthetic valuation as that held against Frege.

From a technical standpoint, Zuse's Plankalkül contained what would be considered "standard features" in approaches over a decade later and embodied a functional programming style twenty-five years before similar developments occurred in the Anglocentric community [54, 21]. When John Backus gave a Turing Award Lecture on functional style in 1977, he lamented the entrenchment of the imperative paradigm he helped create, calling numeric languages "conventional" and arguing that they had to be "liberated" from a "fixation" on the von Neumann computer [12, p.616]. Ironically, Zuse had built a similar functional style into his language around the same time von Neumann was handwriting the *First Report on the EDVAC* [134] which came to define what a "von Neumann computer" is. Zuse wrote of the times, "as a German it would have been difficult to gain the necessary attention at discussions dominated by Americans" [141, p. 128].[6]

The ENIAC Vision

A similar vision of computer programming as involving written and drawn coding emerged around the same time in the United States on the ENIAC project, although for different reasons. The ENIAC project is of course well-trodden territory in the history of computing, considered a landmark for the electronic stored-program [62, 3, 88]. The women of the project, for decades dismissed as "operators," gained belated recognition [20]. This section focuses on the cultural influences behind the written practices of the project's vision of high-level coding, formally published in the 1947 *Planning and Coding Reports* by Herman Goldstine & John von Neumann (hereafter GvN) with the aid of Adele Goldstine and Arthur Burks [57, 56]. These reports were the first to publicly formalize computer programming as a methodology and popularized the term "programming" [61].

The ENIAC computer was built and run in the Moore School of Electrical Engineering at the University of Pennsylvania between 1943-55. In the early stages of development, ENIAC programs were entirely represented as a sequence of machine operations ("machine code" or, in their terminology, order codes), then painstakingly converted to switch flips, plugboard arrangements, and punch cards by women who physically programmed the machines [62]. Deciphering meaning from sequences of orders was extremely difficult. Von Neumann had worked out sorting algorithms in detail, and so had intimate knowledge of the challenges facing the translation task. The first merge sort algorithm he wrote by hand [133] which was common; his handwritten reports often had to be typed up by others [111, p. 6]. To cope with the growing complexity of machine-level coding, GvN developed a notation of box-and-arrow diagrams they called "flow diagrams." In the

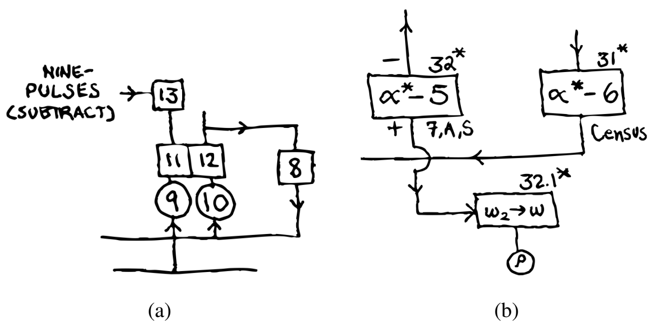

Figure 2: (a) Section of ENIAC accumulator block diagram abstracting an electronic circuit, Arthur Burks, Aug. 1947; (b) Section of flow diagram hand-drawn by Adele Goldstine, Dec. 1947. *Rewritten from originals* [30, 55].

report, GvN state that "coding begins with the drawing of the flow diagrams." Such was the "dynamic or macroscopic stage of coding" [57, p. 20]. Although later work in software would characterize drawing diagrams as purely documentation, extraneous to the practice of coding and "drawn after, not before, writing" programs [29, p. 194], coding in the ENIAC vision was firstly inscribing by disciplined hand. In later stages of "static coding," equations in boxes were converted into machine code and substituted for numbers. The four stages of preparation reflected the division of labour established in the data processing industry decades earlier [110].

The written culture of the Moore School's electrical engineers was central to how GvN conceived of this practice (Figure 2). GvN originally used the engineering term "block diagram" in an early draft of the *Planning and Coding* reports [111]. Block diagram had been terminology used in electrical engineering for at least a decade prior [96, 22] and similar diagrams were drawn by hydrodynamic engineers and in industrial manufacturing [47, p. 326]. Block diagrams were commonplace in the Moore School, as may be seen in the initial proposal for the ENIAC machine itself in spring 1943 [3, p. 90] and a later paper by Burks [30]. The notation gave an abstract picture of a circuit or system. Rather than including all the fine details in one picture, the block diagram would "block" out sections with labelled boxes, to be filled in later with more detailed figures. Flow diagrams did similarly, except mathematical operations would be written inside the boxes. The similarities between notations go beyond from the mere use of blocks and connecting lines, however, and include $+$ and $-$ beside conditional blocks (taken from polarity in electrical schematics), directional arrows, and semi-circles to 'hop' over visually intersecting lines [62]. When the flow diagrams were later in use, "the interaction of mathematicians and computer operators, among others, created a pidgin version" of the notation as it met the realities faced by implementation [111, p. 88].

Yet other factors appeared to contribute to GvN's choice of notation. GvN were familiar with formal logic and abstract mathematics, so they arguably had the technical ability to approximate Zuse's vision. However, several factors worked against this possibility. First, their institutional directive was

$\vee$, open arrow $\Rightarrow$, and overline $\bar{c}$ as negation [113]. These last four are likely from Hilbert/Ackermann [67].

[6]While Zuse's work was only published almost two decades later, Rutishauser, Böhm, and the British and French governments appeared aware of it [141, 26, 116].

to calculate equations, not solve logic conjectures. Second, von Neumann by this time had grown disillusioned with formal logic [3] and the ENIAC programmers were unfamiliar with it [69]. Third, the ENIAC administrators had to coordinate between a large staff and a rigid, secretive organizational hierarchy [3, 47]. It is thus likely that GvN believed that choosing a representation Moore School members were familiar with was superior to introducing a more radical departure in notation. The remarkable historical work of Priestley [111] unearths similar reasoning in a first draft of the *Planning and Coding* reports:

> "[W]e have acquired a conviction that this programming is best accomplished with the help of some graphical representation of the problem. We have attempted... to standardize upon a graphical notation... in the hope that [it] would be sufficiently explicit to make quite clear to a relatively unskilled operator the general outline of the procedure. We further hope that from such a block-diagram the operator will be able with ease to carry out a complete coding of a problem." (p. 59)

Note here the assumption that the "unskilled operator" –the women –could easily follow a notation rooted in the culture of electrical engineering. This was not true at the start of the project. Unlike members of the Moore School, the six ENIAC programmers (formerly human computers and mathematicians) had no training in electrical engineering. Upon arrival, they were merely handed block diagrams and asked to study them.[7] Jennings Bartik recounts, "I had never read a block diagram in my life. Betty [Snyder] hadn't either, and we assumed it was read from left to right like a book... I am still amazed at how little help, instruction, or supervision we had" [20, p. 75, 80]. This suggests that Bartik & Snyder applied their existing cultural knowledge to the new material, making sense of the situation as best they could.

Flow diagramming later spread largely due to von Neumann's celebrated status [3, 96], rather than any concerted effort by a group or individual. Yet in the 1950s, Saul Gorn of the Moore-affiliated Ballistics Research Lab [3, 101] would make explicit the move to universalize the method. Gorn founded a research programme to find a "Universal Code" where "the flow chart would be the code" to be translated later into a representation suitable for any underlying machine [78, p. 20]. This argument influenced the later ALGOL international commission, a (largely failed, but highly influential) North American and European effort to standardize a universal notation to define algorithms [125]. Ultimately, flow diagramming would have a powerful influence on programming and software engineering for decades to come, becoming the basis for the "visual" paradigm of coding [28, 96] –a point to which I will return.

[7]After extensive archival work, Haigh *et al.* conclude that programmers' later block and flow diagramming methods "were based on work done long before they were hired," which corrects some prior accounts [62, p. 95].

The Typewritten Vision and the Serialization of Programming Notation

From 1950 onward, visions of typing (non-numeric) symbols to program emerged, beginning with assembly code [83, 78]. Prior historical work on writing and programming often begins with this era (e.g., see [130, 101, 5]). Unlike the relative isolation of the prior two visions, by this time a computing community began to form [3], making it more difficult to trace influences.[8] Both Swiss mathematician Heinz Rutishauser and Italian student Corraldo Böhm envisioned use of a keyboard years before the more well-known MIT WHIRLWIND and IBM FORTRAN projects [116, 26]. Here I touch on the work of Böhm, at MIT and at IBM.

At a high level, and more rigidly than the prior two visions, typewritten approaches were fabricated through a "dialectic of resistance and accomodation" between humans and machines [109, p. 22]. In this dialectic, machines resisted cultural practices which had developed along different material constraints. In turn, inventors accommodated the resistance through workarounds or modifications. The degree of accommodation depended on *what* specific machines inventors had on-hand, whether they actually implemented their vision, and the flexibility of the organization (if any) they operated under.

A strong commonality between the inventors of typewritten visions was how they designed for mathematical users. Böhm wanted to "[adhere], as closely as possible, to the notational conventions followed by mathematicians" [26]; Laning & Zierler's goal at MIT was "to stay as close as possible to ordinary mathematical notation" [86, p. 1-2]; and FORTRAN stands for "Mathematical Formula Translating System" and its preliminary report depicts pages of translations from mathematical notation (Figure 3) –considerations that were removed from discussion in their 1957 paper [15, 17]. The culture of applied mathematics thereby came into conflict with the culture of English commerce and business baked into IBM punch card machines and keypunches, which had been outfitted in the 1930s with alphanumeric encoding that excluded mathematical notation as simple as multiplication [10, 3]. Jones lamented the "prohibitively expensive" issue in 1954: "the typewriters and similar equipment... just doesn't have the necessary symbols... we are shackled by the design of a relatively insignificant... piece of auxiliary machinery" [78, p. 24].

Examples abound of the resulting accommodations. In early 1953 at MIT Project WHIRLWIND [52], Laning & Zierler took a Flexowriter teleprinter (a combined manual keypunch and punch-card controlled printer whose design had passed through IBM) and began development of an "interpretive program" that they optimistically describe as being able to "accept algebraic equations... (within certain limits imposed by the Flexowriter)" [51, p. 4]. They were motivated by the reduction of "mistakes" and the need to educate outside, timeshared mathematicians and scientists on the machines' operation [52].

[8]For example, from 1948-9 Swiss mathematicians Eduard Stiefel and Heinz Rutishauser visited von Neumann, returned to Europe and met Zuse, his Z4 and Plankalkül –effectively bridging the isolation between visions. In 1952, they helped develop the Swiss computer ERMETH [100].

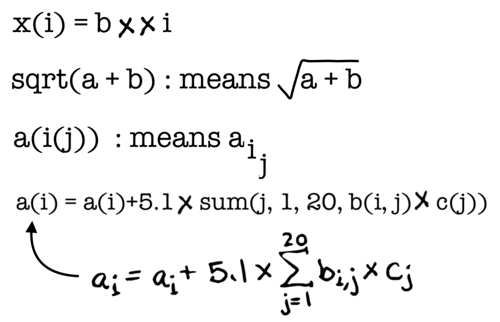
x(i) = b ✕✕ i
sqrt(a + b) : means $\sqrt{a+b}$
a(i(j)) : means a_{i_j}
a(i) = a(i)+5.1 ✕ sum(j, 1, 20, b(i, j) ✕ c(j))
$a_i = a_i + 5.1 \times \sum_{j=1}^{20} b_{i,j} \times c_j$

Figure 3: Examples of translations from mathematical notation to what FORTRAN designers anticipated could be typed on an IBM keypunch (1954). Notice the handwritten $\times$ symbols and lowercase letters: by 1956, these became asterisks and uppercase letters [14, 15]. *Rewritten from original* [16].

Inventors accommodated Flexowriter limits with workarounds like representing a subscript by a vertical bar before a superscript (e.g. $n|^2$), or by "fil[ing] off... [the] lower dot of the colon [key]" to make a multiplication symbol [86, p. 10-15]. The interface also inspired an innovation for overcoming ambiguity: keys for toggling between upper and lower case were appropriated to disambiguate between inputting commands and variables.

At IBM, FORTRAN designers had less flexibility to alter machines to suit their needs. John Backus, who founded the effort, was a college-educated mathematician hired to calculate Fourier series; long hours and difficulties of using machine code motivated him to "make it a little easier" [27]. He submitted a proposal to IBM management to lead a team to expand on this idea [13]. Operating in relative obscurity due to political tensions between Thomas Watson Jr. and Sr., the FORTRAN team had to work within the constraints of IBM's ecosystem of standardized calculating machines built for business, aircraft, and government markets. They made do with IBM keypunch limitations by, for instance, using parentheses to denote super- and sub-scripts, adopting * and ** for multiplication $\times$ and exponentiation, and enforcing all uppercase letters. Though these changes may be seen as concessions, they may also be seen as standards enforcing an economy of notation rather than idiosyncrasy.[9]

Yet for typewritten visions, the material form of the machines enforced constraints that went beyond mere symbol swapping. The linearity and limited size of punch cards and the left-to-right, top-to-bottom norm of Anglo- and European societies enforced a notation that involved a sequential series of horizontal rows of characters, where the number of characters was limited by punch card size. Semi-sequential notations like $\Sigma_{j=1}^{n}$ confounded serial input (i.e. it is unclear whether n or $i = 1$ came first), and thus had to be serialized e.g. `SUM(J,1,N,...)` [17, p. 10]. Backus & Herrick's IBM Speedcoding paper in 1954 describes the challenges facing the translation task between "rich" mathematical notation into "fairly involved" typed expansions:

> "Obviously the programmer would like to write... '$X+Y$' instead of: 'CLEAR AND ADD 100'... To go a step further he would like to write $\Sigma\, a_{ij} \cdot b_{jk}$ instead of the fairly involved set of instructions corresponding to this expression." [16, p. 112]

Unlike before, here 'writing' is no longer assumed the domain of the handwritten; instead, primacy is granted to the digitizing, standardized interfaces one grapples with, even when they are not there. Through this transition from writing to 'writing' (what can be typed), the serialization of notation enforced by typewriters and punch cards enabled an easy alliance with the metaphor of 'language,' a metaphor that even computing historians "forget... has its own history" [101]. In the Zuse and ENIAC visions, 'language' did not appear in any major way: Zuse preferred calculus and the term appears in GvN's reports only once in reference to machine code [57]. By contrast, FORTRAN papers describe the notation as a language rather than as code or psuedo-code, announcing that this approach should "virtually eliminate coding" [17, p. 2]. Around this time, Grace Hopper and the popular media also played a large role in spreading the metaphor [101].

Typewritten visions of programming, rather than the Zuse and ENIAC visions, "[asked] what was possible to implement rather than what was possible to write" [83, p. 15]. But the vision of typing code, despite its suggestion, still did not fully attach 'writing' to 'typing.' Following the division of labour in data processing at Remington Rand and IBM, among others, women were employed as keypunch typists on the UNIVAC (e.g., [112]) or in institutions that ran FORTRAN [84]. Statements were handwritten or typed onto paper slips called coding sheets and handed off to typists for punching [14]. It was not until punch cards were phased out in the electromechanical teleprinter era that typing "directly" into the machine displaced the need to handwrite code [50].

DISCUSSION

These case studies of the origins of programming notation reveal several insights about the earliest history of HCI. First, 'code,' even in the current sense, was foremost handwritten and drawn before it was typed. The materiality of writing afforded alternative representations more closely associated with 'notation' than language. Second, the design of notations and practices for programming originally extended and adapted prior cultural activity. These adoptions included, but went beyond the simple use of natural language for keywords. Methods designed to suit one community's culture, such as flow diagramming at the Moore School adapted from the practices of its electrical engineers, spread and were widely adopted with little reflection on their situatedness. Two of these methods, ENIAC and FORTRAN, thereafter delineated the dominant culture against which later approaches were measured and justified [125, 130]. Other methods, such as Zuse's Plankalkül, remained ignored for cultural and historical reasons. Third and most importantly, programming notations and their use are –and always were –social and material sites of intercultural conflict, compromise, and innovation. Especially for those

[9]While Backus claimed he knew little about the ENIAC [117], he drew flow diagrams in 1951-2 which quite closely resemble Goldstine & von Neumann's notation [11].

marginalized from male, Anglocentric norms, programming did not just mean punching cards, flipping switches, planning or typing code, but was (and is) often "a problem of mapping from one culture to another" [38, p. 138], a recurrent site of translation work, cultural tensions and learning (see also [58, 8]). Beyond the translation work of social collaborations 'around' programming systems [106], however, I argue that translating between representations is a fundamental quality of the practice of writing code, and is not always the result of technical limitations.

Still, some readers may cling to the feeling that the shift-key keyboard was inevitable and typing code is actually the 'best' method (similar arguments to Brooks [29] and those who tried to explain the failure of Engelbart's keyset [18, p. 217-8]). Indeed, like all standards in HCI [76], the metaphor of language and English keyboard interface did prove generative, such as inspiring approaches like Hopper's COBOL [5] and enabling global traffic in code [58]. And it is also true that technical limitations hampered early alternatives such as GRAIL's light pen coding [46], which Alan Kay called the most "intimate" interface he had ever experienced, but which suffered from a heavy stylus and low refresh rates [80]. But cases like Zuse's Plankalkül should give us pause, raising serious questions about how the generalizing of some early, highly situated designs not only act to enable our interactions with computers, but to enframe and constrain later imaginations. The historical record suggests that feelings of inevitability or superiority, rather than being rooted in technical facts, may instead be the outgrowth of a deep-rooted ethnocentrism, an understandable resistance to cognitive dissonance. If HCI is truly to support and engage with a diversity of minds and cultural backgrounds, then we should resist the urge to centre the typewritten with moves which imply its dominance or universality. We continue to do so in two major ways.

First, alternative approaches to programming are often framed and justified for publication as 'educational' (to inculcate newcomers into the old regime) or for mere 'end-users' –i.e., people who are not, in the end, 'experts.' The implication is, of course, that the typewritten is the domain of the expert, and the 'visual' (or anything else) is for "newbies" [127]. If we are to change, we must be willing to challenge the practices and values of experts (as, for instance, the notation design work of Bob Coecke and collaborators have in quantum physics [33]). Second, and as I expand upon below, the way we speak about programming and measure programming knowledge centres the typewritten –from the ACM classifier of this paper ("History of Programming Languages"), to the organizational and theoretical focus on single languages, to gate-keeping exams like the AP Computer Science A [25], to the kinds of questions asked by HCI researchers and neuroscientists [107], whose experiments seek to influence pedagogy, evaluation, and design. For instance, in August 2019, one of the creators of a 'language-independent' coding assessment apologized for their claims of independence and claimed that "we as a research community haven't thought deeply enough yet about the interaction between programming languages and cognition" [59]. Following situated theories of cognition, we must learn to see programming systems as cultural tools that are embedded in particular social activity. Keeping this broad point in mind, I now connect my work to other scholars and draw further insights this framing of HCI's early history might provide to the current field.

The Early History of HCI as Situated Knowledge and "Making Do"

On the one hand, many progenitors of writing code were in a relatively privileged position in their respective societies, with many having the time, education, and resources to invent a new practice, even if some were marginalized by their contemporaries. On the other hand –and unlike later 'hackers' motivated by revolution, liberation, or democratization [7] –many inventors were motivated by the need to simplify the everyday difficulties of handling error-prone data processing machines, systems which had existed in a similar form for decades. These inventors were "making do" [8] with their particular situation and on-hand materials to make incremental improvements to their interactions with computers. Keyboard interfaces were appropriated not out of a suite of alternatives or by some leap of imagination, but because they were literally lying around, ready to be repurposed –just like the other written practices I mention.

This situated, contingent perspective on knowledge construction connects with standpoint theory and third paradigm HCI [66, 122, 45]. Drawing from Donna Haraway's situated knowledges [65], standpoint theory argues that scientific or technological visions often present themselves as objective truth –in the parlance of programming, masking themselves as universal or general-purpose –but are in fact "coming from particular points of view and generated through particular mechanisms" [66]. For example, rather than seeing FORTRAN as the first 'general-purpose,' compiled notation, this perspective would argue that FORTRAN was specific to a domain, in exactly the same way as, say, Max/Msp [1] is a programming environment for musicians. Similarly, rather than seeing the rise and fall of flow diagrams as reflecting the failure of ostensibly 'visual' thinking [96, 104, 28], this perspective instead would argue that flow diagrams were ill-suited to the wide range of different contexts in which they travelled. It was not the 'visual' that was flawed –such a blanket statement reflects what Dourish & Mainwaring call a "colonial impulse" [45] –but the early computing culture's habit of universalizing and marginalizing, coupled with the constraints of machines and infrastructure. This same habit of generalizing "[o]ne historical particularity... into a timeless and spaceless universality" [135] drove the entrenchment of the imperative (i.e. FORTRAN) paradigm.

The Naturalization of the Textual/Visual Dichotomy

My analysis also builds on the broader historical marginalization of handwork as something outside of programming [114]. Zuse and ENIAC project members imagined coding as involving written and drawn forms, connecting to their dispositions and practices as artists, mathematicians, and engineers. Later, however, coding became imagined as foremost typewritten, and programming notations became described as 'languages' belonging to either 'textual' or 'visual' paradigms [28]. Zuse's zig-zagged, row-crossing line challenges the very distinction between the 'textual' and 'visual,' revealing it as a fabrication tied to many factors: the early dominance of the keyboard, the metaphor of language, and the serialization enforced by early machines' processes. For instance, a paper in 1995 recounts arguments against visual languages which claimed that they are "not equally acceptable for all" and rationalized the position with a myth about right-left brain hemispheres [104].[10] Over a decade earlier in a Turing Award talk, Kenneth Iverson justified a typewritten approach by claiming that mathematical notation "lacks universality" and the typewritten is instead "universal (general-purpose)" [75, p. 340] –indeed a stark value shift from early inventors' deference to mathematical notation.[11]Ingold argues that this constructed boundary between the textual and the drawn "hinges upon a dichotomy between technology and art that has become deeply entrenched within the modern constitution" but that "dates back no more than three hundred years" and has its genesis in the rise of industrial capitalism, division of labour, and routinization [73, p. 127].

To disrupt the textual/visual fabrication in future work, we might consider the translation work of even the most technical people. Two burgeoning fields with this property are machine learning and quantum computing. Quantum computing practitioners communicate via a variety of diagrams and notation, yet when 'writing code' for a quantum computer, APIs require users to translate these representations into a FORTRAN-like sequence of calls [9]. While this typewritten standard allows easy inter-operation with other code and infrastructure, it also perpetuates a value-laden idea that the 'visual' is, in the words of one user, for those in "kindergarten" [9, p. 7], echoing those who ridiculed Frege's notation. Such statements, I argue, should not be taken at face value in deference to the 'experts' or user-centred design. Instead, we should pay attention to how our interfaces have naturalized and centred typewritten notations, precluding the possibility that alternatives offer improvements over the typewritten (e.g., diagrams for unruly tensor indices [33, p. 10]).

Embracing Heterogeneity in Programming Practice

So far, I have mainly been concerned with questions of computing culture and history, rather than speaking more directly to the subfield of programming (usually appended with languages and abbrev. as PL). Today, many programming communities continue a tendency to be biased towards a single approach –towards a single language or a one-size-fits-all vision –rather than viewing programming as a practice involving interactions between a plurality of representations, practices, infrastructure, people, and (possibly contradictory) perspectives. This tendency is embedded in the way programming is taught, where content often focuses on learning a language or paradigm, and avoids other learning about how to contend with infrastructures setup to support programming, or communication between people, software, and indeed other notations [89, 106, 64]. Said another way, PL researchers' continued attention to single languages, boxed-in categories, and traditional eschewing of HCI methods and factors [123] resists efforts to conceptualize and design programming systems as interminglings of practices and representations.

As I have suggested above, future work in programming system design can build on the lessons of the past by embracing, rather than avoiding, heterogeneity in programming practice (e.g., drawing diagrams in Jupyter notebooks that *are* the code). In part, such efforts may benefit from paying a deeper attention to the 'translation work' users perform when writing code and how new notations and practices extend existing culture (whether to support that existing culture, or to design new practices that reflectively reject it). Suchman's concepts of "partial translations" and "artful integrations" are important resources here: that "in place of the vision of a single technology that subsumes all others (*the* workstation, *the* ultimate multifunction machine), [designers] assume the continued existence of hybrid systems composed of heterogeneous devices" [122, p. 99]. As Lindtner et al. argue, those that seek to alter the status quo might also draw from feminist concepts of "walking alongside" (roughly, tolerance and respect without comprehension) and "parasitic resistance" – "an entity that is dependent on a host yet pursues independent goals, including goals that go against the interests of the host" [91] –indeed familiar concepts to intercultural competence education [63], infrastructure studies of organizational change [119], and the tension of learning within a dominant culture articulated by Lisa Delpit in *Other Peoples' Children* [40].

Finally, I return to the anecdote where we began: DynamicLand's programming ecosystem and rhetoric of liberation. Although the keyboard and screen reform as an "obligatory passage point" [87], an alternative, optimistic reading considers the project as a parasitic resistance grafted onto the status quo that it will eventually consume. Indeed, multi-domain approaches to programming are increasingly gaining acceptance in coding communities, reflected in the confluence of paradigms supported by Python and JavaScript, in approaches like React and Darklang, and also in PL theory, expanding on an earlier body of work on foreign function interfaces [2, 93]. In particular, the metaphor of language is now extended to "multilingual," [59] "polyglot" [64], or "multi-language worlds" [2]. While these approaches remain largely beholden to a typewritten, English status quo, they do represent a shift towards the embrace of the pidgin and creole, the hybrid, towards a kind of epistemological inclusivity. A prominent example is the typewritten, Lisp-based Racket, framed (perhaps strategically) as an educational language. Its manifesto declares: "A proper approach [to programming] uses the language of the domain to state the problem and articulate solution processes... [S]ystems will necessarily consist of interconnected components in several different languages" [49, p. 114]. Though Racket too tends towards a totalizing project ("there must be no need to step outside" [49]), its approach is rare in the field of programming languages and represents a promising turn towards the intercultural.

[10]In cogniton and neuroscience, a growing number of studies suggest that processing of ostensibly formal (symbolic) notations utilizes visuo-spatial, nonlinguistic parts of the brain [85, 6].

[11]Theorists might raise questions here about computational universality and Turing completeness. Although not my focus, I ask theorists to notice how notions of completeness are upheld through translation work, e.g. to Turing machines or lambda calculus, encodings for numbers, etc. One might also keep in mind that, as Felleisen notes, proving completeness does little for insightful PL design [48].

CONCLUSION

"The future... is that which breaks absolutely with constituted normality and can only be proclaimed, presented, as a sort of monstrosity."

– Jacques Derrida [41]

Is it possible that the phrase "to write code" will not immediately imply typing in the future? Although some in programming and HCI continue to centre typewritten approaches and deploy universalist language, a historical perspective suggests that notations and practices of programming are likely much more situated than we typically imagine. Those in HCI and CS education should keep in mind how programming notations are cultural tools that are products of intercultural tensions and compromise, and not neutral descriptors of algorithms or systems. The development of new programming systems will not just involve appropriation of the past, but conscious reflection on –and sometimes rejection of –its practices, notation, and discourse, of how they have come to condition our bodies and imaginations. While some may resist such efforts, I am hopeful that with enough reflection and an intercultural perspective, we in HCI can truly take steps towards a "plurality of centres" [135] in expert –not just end-user or educational –programming practice.

ACKNOWLEDGEMENTS

The author would like to thank Matthew Milano, Molly Feldman, Matthew Bierbaum, Stephen Hilgartner, Steve Jackson, Tapan Parikh, Phoebe Sengers, and the PLDG Seminar Group at Cornell University for their insights and support.

REFERENCES

[1] Cycling '74. 2019. Max/Msp/Jitter. (2019). https://cycling74.com

[2] Amal Ahmed. 2015. Verified Compilers for a Multi-Language World. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, Vol. 32. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 15–31. DOI: http://dx.doi.org/10.4230/LIPIcs.SNAPL.2015.15

[3] Atsushi Akera. 2008. *Calculating a natural world: scientists, engineers, and computers during the rise of US Cold War research*. MIT Press.

[4] Ali Alkhatib, Michael S. Bernstein, and Margaret Levi. 2017. Examining Crowd Work and Gig Work Through The Historical Lens of Piecework. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. ACM, New York, NY, USA, 4599–4616. DOI: http://dx.doi.org/10.1145/3025453.3025974

[5] Ben Allen. 2017. *Critical Approaches to the Materiality of Source Code: Between Text and Machine*. Ph.D. Dissertation. Stanford University.

[6] Marie Amalric and Stanislas Dehaene. 2016. Origins of the brain networks for advanced mathematics in expert mathematicians. *Proceedings of the National Academy of Sciences* 113, 18 (2016), 4909–4917.

[7] Morgan G. Ames. 2018. Hackers, Computers, and Cooperation: A Critical History of Logo and Constructionist Learning. *Proc. ACM Hum.-Comput. Interact.* 2, CSCW, Article Article 18 (Nov. 2018), 19 pages. DOI:http://dx.doi.org/10.1145/3274287

[8] Morgan G. Ames, Silvia Lindtner, Shaowen Bardzell, Jeffrey Bardzell, Lilly Nguyen, Syed Ishtiaque Ahmed, Nusrat Jahan, Steven J. Jackson, and Paul Dourish. 2018. Making or making do? Challenging the mythologies of making and hacking. *Journal of Peer Production* 12 (2018).

[9] Zahra Ashktorab, Justin D. Weisz, and Maryam Ashoori. 2019. Thinking Too Classically: Research Topics in Human-Quantum Computer Interaction. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. ACM, New York, NY, USA, Article Paper 256, 12 pages. DOI: http://dx.doi.org/10.1145/3290605.3300486

[10] William Aspray. 1990. *Computing before computers*. Iowa State University Press.

[11] John W. Backus. 1951-2. Problem #29 Flow Chart for IBM SSEC. (1951-2). John W. Backus Papers, Library of Congress Manuscripts Division, box OV1, item 71.

[12] John W. Backus. 1978a. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Commun. ACM* 21, 8 (1978), 613–641.

[13] John W. Backus. 1978b. The history of Fortran I, II, and III. *ACM Sigplan Notices* 13, 8 (1978), 165–180.

[14] John W. Backus, R.J. Beeber, S. Best, R. Goldberg, H.L. Herrick, R.A. Hughes, L.B. Mitchell, R.A. Nelson, R. Nutt, D. Sayre, and others. 1956. The FORTRAN Automatic Coding System for the IBM 704 EDPM: Programmer's Reference Manual. *Applied Science Division and Programming Research Department, International Business Machines Corporation* 590 (1956).

[15] John W. Backus, Robert J. Beeber, Sheldon Best, Richard Goldberg, L. Mitchell Haibt, Harlan L. Herrick, Robert A. Nelson, David Sayre, Peter B. Sheridan, H. Stern, and others. 1957. The FORTRAN automatic coding system. In *Papers presented at the February 26-28, 1957, Western Joint Computer Conference: Techniques for reliability*. ACM, 188–198.

[16] John W. Backus and Harlan Herrick. 1954. IBM 701 Speedcoding and other automatic-programming systems. In *Symposium on Automatic Programming for Digital Computers, Washington, DC*, Vol. 13. 106–113.

[17] John W. Backus, Harlan Herrick, and Irving Ziller. 1954. Preliminary Report, Specifications for the IBM Mathematical FORmula TRANslating System, FORTRAN. *Applied Science Division, IBM* (1954).

[18] Thierry Bardini. 2000. *Bootstrapping: Douglas Engelbart, coevolution, and the origins of personal computing*. Stanford University Press.

[19] William Barnes, Myles Gartland, and Martin Stack. 2004. Old habits die hard: path dependency and behavioral lock-in. *Journal of Economic Issues* 38, 2 (2004), 371–377.

[20] Jean Jennings Bartik. 2013. *Pioneer programmer: Jean Jennings Bartik and the computer that changed the world*. Truman State University Press.

[21] F. L. Bauer and H. Wössner. 1972. The "Plankalkül" of Konrad Zuse: A Forerunner of Today's Programming Languages. *Commun. ACM* 15, 7 (July 1972), 678–685. DOI:http://dx.doi.org/10.1145/361454.361515

[22] Harold H. Beverage. 1937. Signaling. (Oct. 5 1937). US Patent 2,095,050 (Filed Apr. 26, 1933).

[23] Nicola J. Bidwell. 2016. Moving the centre to design social media in rural Africa. *AI & Society* 31, 1 (2016), 51–77.

[24] Wiebe E. Bijker, Thomas Parke Hughes, and Trevor J. Pinch. 1989. *The social construction of technological systems: New directions in the sociology and history of technology*. MIT Press.

[25] College Board. 2019. AP Computer Science A. (2019). https://apcentral.collegeboard.org/courses/ap-computer-science-a/course

[26] Corrado Böhm. 1951. On encoding logical-mathematical formulas using the machine itself during program conception. Translated 2016 by Peter Sestoft from Corrado Böhm: Calculatrices digitales. Du déchiffrage de formules logico-mathématiques par la machine même dans la conception du programme. *ETH Zürich* (1951). PhD dissertation. French original published in Bologna 1954.

[27] Grady Booch. 2006. Oral History of John Backus. (2006). Computer History Museum.

[28] Marat Boshernitsan and Michael Sean Downes. 2004. *Visual programming languages: A survey*. Citeseer.

[29] Frederick P. Brooks. 1975. The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition. *Addison-Wesley* (1975).

[30] Arthur Walter Burks. 1947. Electronic computing circuits of the ENIAC. *Proceedings of the IRE* 35, 8 (1947), 756–767.

[31] Florian Cajori. 1929. *A History of Mathematical Notations (Vol. I and II)*. Chicago: Open Court Pub. Co.

[32] Daniel Chandler. 1992. The phenomenology of writing by hand. *Digital Creativity (Intelligent Tutoring Media)* 3, 2-3 (1992), 65–74.

[33] Bob Coecke and Aleks Kissinger. 2017. *Picturing quantum processes*. Cambridge University Press.

[34] I. Bernard Cohen, Robert V. Campbell, Gregory W. Welch, and Robert V.D. Campbell. 1999. *Makin'numbers: Howard Aiken and the computer*. MIT Press.

[35] Michael Cole, Distributive Literacy Consortium, and others. 2006. *The fifth dimension: An after-school program built on diversity*. Russell Sage Foundation.

[36] James W. Cortada. 2000. *Before the computer: IBM, NCR, Burroughs, and Remington Rand and the industry they created, 1865-1956*. Princeton University Press.

[37] Adam Curtis. 2016. HyperNormalisation. *BBC Documentary* (2016). Film.

[38] Subrata Dasgupta. 2014. *It Began with Babbage: The Genesis of Computer Science*. Oxford University Press.

[39] Paul A. David. 1985. Clio and the Economics of QWERTY. *The American economic review* 75, 2 (1985), 332–337.

[40] Lisa Delpit. 2006. *Other people's children: Cultural conflict in the classroom*. The New Press.

[41] Jacques Derrida. 1976. Of Grammatology, trans. Gayatri Chakravorty Spivak. (1976).

[42] Lexico (Oxford Dictionary). 2019. Definition of Programmer in English. (2019). Retrieved September 17, 2019 from https://www.lexico.com/en/definition/programmer

[43] Paul Dourish. 2015. Not the internet, but this internet: how othernets illuminate our feudal internet. In *Proceedings of The Fifth Decennial Aarhus Conference on Critical Alternatives*. Aarhus University Press, 157–168.

[44] Paul Dourish. 2017. *The stuff of bits: An essay on the materialities of information*. MIT Press.

[45] Paul Dourish and Scott D. Mainwaring. 2012. Ubicomp's Colonial Impulse. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing (UbiComp '12)*. ACM, New York, NY, USA, 133–142. DOI:http://dx.doi.org/10.1145/2370216.2370238

[46] Thomas O. Ellis, John F. Heafner, and William L. Sibley. 1969. *The GRAIL Project: An experiment in man-machine communications*. Technical Report. RAND Corp. Retrieved Aug. 8, 2019 from https://www.rand.org/pubs/research_memoranda/RM5999.html

[47] Nathan Ensmenger. 2016. The multiple meanings of a flowchart. *Information & Culture* 51, 3 (2016), 321–351.

[48] Matthias Felleisen. 1991. On the expressive power of programming languages. *Science of computer programming* 17, 1-3 (1991), 35–75.

[49] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2015. The Racket Manifesto. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, Vol. 32. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 113–128. DOI: http://dx.doi.org/10.4230/LIPIcs.SNAPL.2015.113

[50] Dale Fisk. 2005. *Programming with punched cards*. Columbia University, Computing History Archives. Retrieved August 10 2019 from http://www.columbia.edu/cu/computinghistory/fisk.pdf

[51] Jay Forrester and others. 1953a. *Reports of MIT Project WHIRLWIND (Biweekly Report for Nov. 16, 1953)*. Technical Report. MIT Digital Computer Laboratory.

[52] Jay Forrester and others. 1953b. *Reports of MIT Project WHIRLWIND (Summary Report No. 33)*. Technical Report. MIT Digital Computer Laboratory.

[53] Gottlob Frege. 1879. *Begriffsschrift: eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*. Halle.

[54] W. K. Giloi. 1997. Konrad Zuse's Plankalkül: the First High-level, "non-von Neumann" Programming Language. *IEEE Annals of the History of Computing* 19, 2 (April 1997), 17–24. DOI: http://dx.doi.org/10.1109/85.586068

[55] Adele Goldstine and John von Neumann. Dec. 4, 1947. Monte Carlo Flow Diagram. (Dec. 4, 1947). John von Neumann Papers, Library of Congress Manuscripts Division box 12, folder 6.

[56] Herman H. Goldstine. 1993. *The computer from Pascal to von Neumann*. Princeton University Press.

[57] Herman H. Goldstine and John Von Neumann. 1947. *Planning and coding of problems for an electronic computing instrument*. Technical Report. Moore School of Electrical Engineering, University of Pennsylvania.

[58] Philip J. Guo. 2018. Non-Native English Speakers Learning Computer Programming: Barriers, Desires, and Design Opportunities. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM, New York, NY, USA, Article Paper 396, 14 pages. DOI: http://dx.doi.org/10.1145/3173574.3173970

[59] Mark Guzdial. 2019. We Should Stop Saying 'Language Independent.' We Don't Know How To Do That. *Communications of the ACM Blog* (13 Aug 2019).

[60] Thomas Haigh. 2014. Actually, Turing did not invent the computer. *Commun. ACM* 57, 1 (2014), 36–41.

[61] Thomas Haigh and Mark Priestley. 2015. Where code comes from: Architectures of automatic control from Babbage to algol. *Commun. ACM* 59, 1 (2015), 39–44.

[62] Thomas Haigh, Peter Mark Priestley, Mark Priestley, and Crispin Rope. 2016. *ENIAC in action: Making and remaking the modern computer*. MIT Press.

[63] Mitchell R. Hammer and Milton Bennett. 2012. The intercultural development inventory. *Student learning abroad* (2012), 115–136.

[64] Rebecca L. Hao and Elena L. Glassman. 2019. Approaching polyglot programming: what can we learn from bilingualism studies? (2019).

[65] Donna Haraway. 1988. Situated knowledges: The science question in feminism and the privilege of partial perspective. *Feminist studies* 14, 3 (1988), 575–599.

[66] Steve Harrison, Phoebe Sengers, and Deborah Tatar. 2011. Making epistemological trouble: Third-paradigm HCI as successor science. *Interacting with Computers* 23, 5 (2011), 385–392.

[67] David Hilbert and Wilhelm Ackerman. 1928. *Grundzüge der Theoretischen Logik* (2 ed.). Springer-Verlag.

[68] Andrew Hodges. 2012. *Alan Turing: The Enigma*. Random House.

[69] Frances E. Holberton. 1983. Personal interview. 14 Apr. 1983, OH 50. Oral history interview by James Baker Ross, Potomac, Maryland. Charles Babbage Institute. *University of Minnesota, Minneapolis* (1983).

[70] Ellen Huet. 2014. Camp Grounded: Where People Pay $570 To Have Their Smartphones Taken Away From Them. *Forbes* (Jun 2014).

[71] Edwin L. Hutchins, James D. Hollan, and Donald A. Norman. 1985. Direct manipulation interfaces. *Human-computer interaction* 1, 4 (1985), 311–338.

[72] Tim Ingold. 2013. *Making: Anthropology, archaeology, art and architecture*. Routledge.

[73] Tim Ingold. 2016. *Lines: a brief history*. Routledge.

[74] Lilly C. Irani and Paul Dourish. 2009. Postcolonial Interculturality. In *Proceedings of the 2009 International Workshop on Intercultural Collaboration (IWIC '09)*. ACM, New York, NY, USA, 249–252. DOI:http://dx.doi.org/10.1145/1499224.1499268

[75] Kenneth Iverson. 1987. Notation as a Tool of Thought, ACM Turing Award Lecture, 1979. ACM Turing Award Lectures, The first twenty years. (1987).

[76] Steven J. Jackson and Sarah Barbrow. 2015. Standards and/As Innovation: Protocols, Creativity, and Interactive Systems Development in Ecology. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. ACM, New York, NY, USA, 1769–1778. DOI: http://dx.doi.org/10.1145/2702123.2702564

[77] Sheila Jasanoff and others. 2004. *States of knowledge: the co-production of science and the social order*. Routledge.

[78] John L. Jones. 1954. *A survey of automatic coding techniques for digital computers*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[79] Victor Kaptelinin and Bonnie A. Nardi. 2006. *Acting with technology: Activity theory and interaction design*. MIT Press.

[80] Alan Kay. 1987. Doing with Images Makes Symbols: Communicating with Computers. (1987). University Video Communications. Film.

[81] Vera Khovanskaya, Lynn Dombrowski, Ellie Harmon, Matthias Korn, Ann Light, Michael Stewart, and Amy Voida. 2018. Designing against the status quo. *interactions* 25, 2 (2018).

[82] Friedrich A. Kittler. 1999. *Gramophone, film, typewriter*. Stanford University Press.

[83] Donald E. Knuth and Luis Trabb Pardo. 1980. The early development of programming languages. In *A history of computing in the twentieth century*. Elsevier, 197–273.

[84] Philip Kraft. 2012. *Programmers and managers: The routinization of computer programming in the United States*. Springer Science & Business Media.

[85] David Landy and Robert L. Goldstone. 2007. Formal notations are diagrams: Evidence from a production task. *Memory & cognition* 35, 8 (2007), 2033–2040.

[86] J. Halcombe Laning and Neal Zierler. 1954. *A program for translation of mathematical equations for Whirlwind I*. Instrumentation Laboratory, Massachusetts Institute of Technology.

[87] Bruno Latour. 2005. Reassembling the Social: an Introduction to Actor-Network-Theory. *Oxford University Press* (2005).

[88] Jennifer S. Light. 1999. When computers were women. *Technology and culture* 40, 3 (1999), 455–483.

[89] Seongtaek Lim, Rama Adithya Varanasi, and Tapan Parikh. 2018. GLIDE (Git-Learning IDE; Integrated Development Environment): In-class Collaboration in Web Engineering Curriculum for Youths (Abstract Only). In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18)*. ACM, New York, NY, USA, 1112–1112. DOI: http://dx.doi.org/10.1145/3159450.3162203

[90] Silvia Lindtner, Shaowen Bardzell, and Jeffrey Bardzell. 2016. Reconstituting the Utopian Vision of Making: HCI After Technosolutionism. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*. ACM, New York, NY, USA, 1390–1402. DOI: http://dx.doi.org/10.1145/2858036.2858506

[91] Silvia Lindtner, Shaowen Bardzell, and Jeffrey Bardzell. 2018. Design and Intervention in the Age of "No Alternative". *Proc. ACM Hum.-Comput. Interact.* 2, CSCW, Article Article 109 (Nov. 2018), 21 pages. DOI: http://dx.doi.org/10.1145/3274378

[92] Peter Lyman. 1984. Reading, writing and word processing: Toward a phenomenology of the computer age. *Qualitative Sociology* 7, 1-2 (1984), 75–89.

[93] Jacob Matthews and Robert Bruce Findler. 2009. Operational Semantics for Multi-Language Programs. *ACM Trans. Program. Lang. Syst.* 31, 3, Article Article 12 (April 2009), 44 pages. DOI: http://dx.doi.org/10.1145/1498926.1498930

[94] Edward F. Melcer and Katherine Isbister. 2018. Bots & (Main)Frames: Exploring the Impact of Tangible Blocks and Collaborative Play in an Educational Programming Game. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM, New York, NY, USA, Article Paper 266, 14 pages. DOI: http://dx.doi.org/10.1145/3173574.3173840

[95] Richard L. Mendelsohn. 2005. *The Philosophy of Gottlob Frege*. Cambridge University Press.

[96] Stephen J. Morris and OCZ Gotel. 2006. Flow diagrams: rise and fall of the first software engineering notation. In *International Conference on Theory and Application of Diagrams*. Springer, 130–144.

[97] Thomas S. Mullaney. 2017. *The Chinese typewriter: A history*. MIT Press.

[98] Kyle Muzyka. 2018. A Hawaiian team's mission to translate programming language to their Native language. (2018). Retrieved August 13, 2019 from https://bit.ly/2lXWnzr

[99] Ramsey Nasser. 2012. Qalb. (2012). Retrieved September 17 2019 from http://nas.sr/%D9%82%D9%84%D8%A8/

[100] Hans Neukom. 2005. ERMETH: The first Swiss computer. *IEEE Annals of the History of Computing* 27, 4 (2005), 5–22.

[101] David Nofre, Mark Priestley, and Gerard Alberts. 2014. When technology became language: The origins of the linguistic conception of computer programming, 1950–1960. *Technology and Culture* 55, 1 (2014), 40–75.

[102] Walter J. Ong. 2013. *Orality and literacy*. Routledge.

[103] Wanda J. Orlikowski. 1991. Integrated information environment or matrix of control? The contradictory implications of information technology. *Accounting, Management and Information Technologies* 1, 1 (1991), 9–42.

[104] V.D. Parondzhanov. 1995. Visual syntax of the DRAKON language. *Programming and Computer Software* 21, 3 (1995).

[105] Samir Passi and Steven Jackson. 2017. Data Vision: Learning to See Through Algorithmic Abstraction. In *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing (CSCW '17)*. ACM, New York, NY, USA, 2436–2447. DOI:http://dx.doi.org/10.1145/2998181.2998331

[106] Samir Passi and Steven J. Jackson. 2018. Trust in Data Science: Collaboration, Translation, and Accountability in Corporate Data Science Projects. *Proc. ACM Hum.-Comput. Interact.* 2, CSCW, Article Article 136 (Nov. 2018), 28 pages. DOI: http://dx.doi.org/10.1145/3274405

[107] Norman Peitek, Janet Siegmund, Sven Apel, Christian Kästner, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. 2018. A look into programmers' heads. *IEEE Transactions on Software Engineering* (2018).

[108] Kavita Philip, Lilly Irani, and Paul Dourish. 2012. Postcolonial computing: A tactical survey. *Science, Technology, & Human Values* 37, 1 (2012), 3–29.

[109] Andrew Pickering. 1995. *The Mangle of Practice: Time, Agency, and Science*. University of Chicago Press.

[110] Mark Priestley. 2018a. The mathematical origins of modern computing. In *Technology and Mathematics*. Springer, 107–135.

[111] Mark Priestley. 2018b. *Routines of Substitution: John von Neumann's Work on Software Development, 1945–1948*. Springer.

[112] Remington Rand. 1952. Remington-Rand Presents the UNIVAC. (1952). Computer History Museum. Film.

[113] Raúl Rojas, Cüneyt Göktekin, Gerald Friedland, Mike Krüger, Olaf Langmack, and Denis Kuniß. 2000. Plankalkül: The first high-level programming language and its implementation. *Freie Universität Berlin* (2000).

[114] Daniela K. Rosner. 2018. *Critical Fabulations: Reworking the Methods and Margins of Design*. MIT Press.

[115] Daniela K. Rosner, Samantha Shorey, Brock R. Craft, and Helen Remick. 2018. Making Core Memory: Design Inquiry into Gendered Legacies of Engineering and Craftwork. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM, New York, NY, USA, Article Paper 531, 13 pages. DOI: http://dx.doi.org/10.1145/3173574.3174105

[116] Heinz Rutishauser. 1952. Automatische Rechenplanfertigung bei programmgesteuerten Rechenmaschinen. *Zeitschrift für angewandte Mathematik und Physik ZAMP* 3, 4 (1952), 312–313.

[117] Larry Saphire. Dec. 15, 1967. Interview with John W. Backus. (Dec. 15, 1967). John W. Backus Papers, Library of Congress Manuscripts Division, box 1, item 7.

[118] Toby Schachman. March 1, 2018. "We've been playing with editing printed code by just pointing a keyboard at a page and typing." Tweet. (March 1, 2018). Retrieved November 23 2018 from https://twitter.com/mandy3284/status/969286629281087488?s=20

[119] Susan Leigh Star and Karen Ruhleder. 1996. Steps toward an ecology of infrastructure: Design and access for large information spaces. *Information systems research* 7, 1 (1996), 111–134.

[120] Valerie Strauss. 2018. Students protest Zuckerberg-backed digital learning program and ask him: 'What gives you this right?'. *The Washington Post* (Nov 2018).

[121] Sharon Hartman Strom. 1989. "Light manufacturing": The feminization of American office work, 1900–1930. *ILR Review* 43, 1 (1989), 53–71.

[122] Lucy Suchman. 2002. Located accountabilities in technology production. *Scandinavian journal of information systems* 14, 2 (2002), 7.

[123] Joshua Sunshine, Elena Glassman, and Sarah Chasins. 2019. PLATEAU Workshop. (2019). https://plateau-workshop.org/

[124] Alex S. Taylor. 2011. Out there. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 685–694.

[125] Matti Tedre. 2014. *The science of computing: shaping a discipline*. Chapman and Hall/CRC.

[126] Kentaro Toyama. 2015. *Geek heresy: Rescuing social change from the cult of technology*. PublicAffairs.

[127] Jasper Tran O'Leary and Nadya Peek. 2019. Machine-o-Matic: A Programming Environment for Prototyping Digital Fabrication Workflows. In *The Adjunct Publication of the 32nd Annual ACM Symposium on User Interface Software and Technology*. 134–136.

[128] Alan M. Turing. 1937. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London mathematical society* 2, 1 (1937), 230–265.

[129] Alan M. Turing. 1950. Computing Machinery and Intelligence. *Mind* 59, 236 (1950), 433–460.

[130] Annette Vee. 2017. *Coding literacy: How computer programming is changing writing*. MIT Press.

[131] Helen Verran. 2001. *Science and an African logic*. University of Chicago Press.

[132] Bret Victor, Paula Te, Josh Horowitz, Luke Iannini, Toby Schachman, and Virginia McArthur. 2017. DynamicLand. (2017). Retrieved November 23 2018 from https://dynamicland.org

[133] John Von Neumann. 1987. *Papers of John von Neumann on computing and computer theory*. Vol. 12. MIT Press.

[134] John Von Neumann. 1993. First Draft of a Report on the EDVAC. *IEEE Annals of the History of Computing* 15, 4 (1993), 27–75.

[135] Ngũgĩ Wa Thiong'o. 1993. *Moving the centre: The struggle for cultural freedoms*. James Currey.

[136] Langdon Winner. 1980. Do artifacts have politics? *Daedalus* (1980), 121–136.

[137] Terry Winograd, Fernando Flores, and Fernando F. Flores. 1986. *Understanding computers and cognition: A new foundation for design*. Intellect Books.

[138] Alexei Yurchak. 2013. *Everything was forever, until it was no more: The last Soviet generation*. Princeton University Press.

[139] Konrad Zuse. 1945a. Der Plankalkül. (1945). Transcribed report. Courtesy of Konrad Zuse Internet Archive (http://zuse.zib.de).

[140] Konrad Zuse. 1945b. Urschrift des Plankalküls. (1945). Original notebook (scanned). Courtesy of Konrad Zuse Internet Archive (http://zuse.zib.de).

[141] Konrad Zuse. 1993. *The computer-my life*. Springer Science & Business Media.