Coding Literacy

How Computer Programming Is Changing Writing

Annette Vee

The MIT Press
Cambridge, Massachusetts
London, England

This book was set in ITC Stone Serif Std by Toppan Best-set Premedia Limited. Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data is available.

ISBN: 978-0-262-03624-5

10 9 8 7 6 5 4 3 2 1

Contents

2 Sociomaterialities of Programming and Writing

As ambiguously as that first apple, programming and writing help to create and reveal knowledge. Jack Goody called writing a "technology of the intellect," Walter Ong considered it a "technology that restructures thought," and Marshall McLuhan considered writing a form of media that was an "extension of man."[1] While these concepts of writing are deliberately imprecise, even horoscopic, they suggest the differences between a technology that works with knowledge and one that works with materials. Goody, Ong, and McLuhan don't tell the whole story, but writing—and programming—are fundamentally different from other technologies that make our lives easier or help us fight or eat or stay healthy. They help us to think; they build knowledge.

While Goody, Ong, and McLuhan asked important questions about what it means for people and societies to write, they answered those questions by putting too much pressure on the technologies themselves, in isolation from the complex social worlds in which technologies are created and used. More recent approaches to writing as a technology avoid their tendencies toward technological determinism by accounting for social and material factors that complicate the use of writing. This chapter draws on those sociomaterial approaches in order to explore the shared characteristics of programming and writing, what makes them both technologies that build, create, and communicate knowledge. Programming and writing are not the same thing, but they have a lot in common and can even merge into each other. Because of their complex and multiple intersections, I want to suggest that programming and writing both deserve a central place in our thinking about human relationships with communication technologies; they should both be part of our concept of contemporary literacy. If the previous chapter examined the ways this claim was justified rhetorically, the current chapter does so theoretically, examining the productive

overlaps between theories of writing and programming as social and material technologies.

I rely on Andrea diSessa's concept of "material intelligences" as a frame because I find it productive for exploring the possibilities—as well as the limits—of how programming builds knowledge, particularly in terms of procedures. According to diSessa, a "material intelligence" combines a material component—a book, a piece of paper, an alphabet—with an ability to interpret that material. He writes, "We can install some aspects of our thinking in stable, reproducible, manipulable, and transportable physical form. These external forms become in a very real sense part of our thinking, remembering, and communicating."[2] A material intelligence becomes a literacy when the ability to interpret its material component becomes widespread—as has been true with writing, but not yet with programming.[3] Because material intelligences can be precursors to literacy, I sometimes refer to programming and writing as literacy technologies here.

According to diSessa, any material intelligence or literacy is supported by three "pillars": technological, social, and cognitive. The cognitive pillar concerns human intelligence; people need certain cognitive skills in order to learn to read and write, for instance. The technological intersects with the social and cognitive in the ways that the material form of literacy affects its circulation. diSessa uses the example of calculus to illustrate this point: we use Leibniz's notation rather than Newton's because it is more transparent and easier to understand. Calculus would never have become so widely taught if its notation were as arcane as Newton's, diSessa argues.[4] Thus, a specific notation technology made the kinds of knowledge that calculus fosters more available to a broader group of people. Of course, social factors can also mean that the easier or more transparent technology does not always win out, as demonstrated by the QWERTY keyboard's persistence. A concept of material intelligences can explain how specific technologies affect the kinds of cognitive abilities fostered and the circulation of those abilities in a society. I focus here on the social and the technological (what I sometimes call material) pillars of literacy rather than the cognitive pillar. Although I don't disagree with diSessa's assertion that there are individual cognitive factors in the learning and practicing of literacy, I find the humanistic and historical tools at my disposal inadequate for attending to them.

Below, I begin by distinguishing my claim that programming and writing are literacy technologies from claims about overly deterministic "consequences" of technologies on societies and cognition. Much work was done in the 1950s and 1960s on the relationships between orality and literacy and speech and writing—including work by Goody, Ong, and

McLuhan—and was then subsequently dismissed as technologically deterministic. Particularly in the case of McLuhan, too much emphasis was placed on the technology of writing as a sole cause of change in human society and individual consciousness. Contemporary work in rhetoric, media, and literacy studies has revived this attention to material technologies, however, with the language of "affordances" rather than "consequences." As diSessa argues, we can think of writing and programming as technologies that make certain kinds of thinking easier, and thus more probable.

To understand the kinds of affordances programming provides, it is helpful to go into detail on what programming is and how it relates to writing and speech. To do this, I outline a historical trajectory of programming: how it moved from being a feat of machine engineering to being a complex symbolic system that could be deployed across multiple computers and global networks. As a symbolic system embodied in text, computer programming becomes similar to writing. But as a time-based performance, something that "runs," programming bears similarities to speech. Theories of writing and speech provide ways to think about programming as a symbolic system used to encode, communicate, and enact information. Speech act theory has taught us that differences between writing and speech are best measured by degree rather than kind, and I contend that this is also the case for speech, writing, and programming. All three are symbolic systems that can effect changes in the world, but their differences in audience and context of use mean they function quite differently.

The complex ways that code affects and effects things in the world can be attributed, in part, to the fact that the computer calls for explicit definitions of concepts. To see how this relates to human language and writing, I use Wittgenstein's theories of language-in-use and Bernard Stiegler's theories of *grammatization*—or, the discretizing of information. Discretization—for example, any move from analog to digital—always throws out some information. That code reduces information is a common source of critique for its ability to represent processes. But discreteness gives programming certain affordances of replication, distribution, and scale. Writing, of course, is also discrete, which allows it to travel in ways that speech cannot, albeit stripped of the nuances of tone and gesture that speech performances can have. Constraints in form such as those that discretization provides can foster creativity, even poetry, in both programming and writing. In programming, these poetic forms often highlight the dual audience for code: computers and humans. Different programming languages and paradigms offer different trade-offs between the expression of processes for the

computer and the legibility of these processes for the programmers. All of these constraints and trade-offs contribute to the ways that programming and writing help to build knowledge, or function as material intelligences.

The arguments presented in this chapter hinge on the idea that programming, like writing, is a symbolic system operating through an inscribed language. As such, programming follows a particular grammar, encodes and conveys information, and is socially shaped and circulated. Programming as a symbolic system distinguishes it from physical technologies. It is a "media machine," such as Mark Poster describes. Media machines (such as programming and writing) "process texts, images, and sound" and are different from other forms of technology that act on natural objects such as wood and iron.[5] Unlike media that have predetermined physical capacities for recording events and ideas—such as the phonograph—programming and writing are both subject to constant interpretive acts that unfold as humans produce them, although they are clearly always subject to certain physical limitations as well. And unlike media that primarily record events and ideas—such as the phonograph—programming and writing are both interpretive; they can be used to construct new ideas.

This view of programming as a symbolic, sociomaterial system similar to writing cannot fully encompass it or describe it; no approach could do that alone. However, it is a productive lens on programming, one that fills a gap left by approaches through math, logic, cognitive science, and engineering, and which gives us ways of thinking about writing and programming as related and inextricable forms of composition. Considering programming and writing together also points to ways that people can, as individuals and societies, create knowledge and structure information. Although I intend the chapter to provide justification for some of the rhetoric that connects programming to writing, I also use it to extend the argument for coding literacy that sees programming expanding the mind and human capability. As I observed in chapter 1, this humanistic argument has been subsumed by connections between programming and economic advantage. I am reviving it here—idealistically, perhaps, but alloyed with the knowledge that ideologies of literacy, even if necessary, are not necessarily good. This chapter lays the theoretical groundwork for subsequent chapters on the historical and contemporary explorations of the ways writing and programming have shaped society.

Thinking with Technologies

Vannevar Bush's visionary 1945 article in the *Atlantic*, "As We May Think," outlined some of the ways that computers could help us organize

information and lead to new ways of thinking. Noting that Gottfried Leibniz and Charles Babbage were both stymied by manufacturing constraints in their efforts to create calculating machines, Bush writes: "The world has arrived at an age of cheap complex devices of great reliability; and something is bound to come of it."[6] That "something" he imagines through a technology he calls a "Memex," which he proposes as an extension of human thinking. Through better organization and access to the world's trove of knowledge, the Memex would allow people to create new "trails" through information, thus building new knowledge through those new connections. Reflecting on the piece in 1964, Martin Greenberger writes, "By 2000 AD man should have a much better comprehension of himself and his system, not because he will be innately any smarter than he is today, but because he will have learned to use imaginatively the most powerful amplifier of intelligence yet devised."[7] Bush revisits his argument in 1967, reminding readers that he had proposed a *personal* device, rather than the mainframes then in use. This personal device was to have "serve[d] a man's daily thoughts directly, fitting in with his normal thought processes, rather than just do chores for him."[8] J. C. R. Licklider envisioned a similar "symbiosis" in 1960, when he hoped that "in not too many years, human brains and computing machines will be coupled together very tightly, and that the resulting partnership will think as no human brain has ever thought and process data in a way not approached by the information-handling machines we know today."[9]

Although it took some time for this process to happen, we now appear to have arrived at the moment when our personal computational devices "fit in" with our thought processes; I argue that computers are augmenting our intellect. Programming, now that it is relatively accessible and broadly applicable, is the critical link between these cheap, ubiquitous devices and their function as extensions to our minds. To examine the intelligence amplification that Bush and Licklider and Greenberger imagined, we should move beyond a discussion of specific devices and look instead toward computation and the functions of programming.

What Makes Something an "Intelligence"?

In "The Question Concerning Technology," originally given as a lecture in 1949, Heidegger says that we tend to think of technology as a tool for greater efficiency, a better way to get things done. But this is wrong, he argues. To push beyond thinking of technology in only instrumental ways, Heidegger claims we must see that "the essence of technology is by no means anything technological."[10] Instead, the essence of technology is in its "way of revealing." Thinking of writing and programming as material

intelligences, we can see that they reveal knowledge that might be more difficult to access through other means. Much of what is done in programming could also be done in writing or speech or engineering, but programming unlocks properties of knowledge that might be obscured through those other methods.

The kind of knowledge revealed by programming concerns procedures. Programmers and computer scientists have often insisted that the programmed computer reveals ways of understanding and encoding procedural knowledge. Of course, procedural knowledge has always been present in human reasoning, but programming highlights it as a central way to understand and represent the world. In a discussion among technological luminaries at MIT in 1961, for example, John McCarthy described how programming reveals a knowledge process previously concealed: "Programming is the art of stating procedures. Prior to the development of digital computers, one did not have to state procedures precisely. Now we have a tool that will carry out any procedure, provided we can state this procedure sufficiently well."[11] Licklider responded to McCarthy to suggest that while programming was novel, what it reveals is something more fundamental: "computer programming [is] a way into the structure of ideas and into the understanding of intellectual processes that is just a new thing in this world."[12] More recently, Ian Bogost points out that although procedural representations can be found elsewhere, "the computer magnifies the ability to create representations of processes."[13] That programming highlights knowledge about procedures has prompted Bogost and others to claim that it contributes to "procedural literacy."

Some go so far as to altogether remove the computer in their characterization of programming as the human ability to express processes. In their seminal programming textbook, *Structure and Interpretation of Computer Programs*, Hal Abelson, Gerald Sussman, and Julie Sussman write:

> Underlying our approach to this subject is our conviction that "computer science" is not a science and that its significance has little to do with computers. The computer revolution is a revolution in the way we think and in the way we express what we think. The essence of this change is the emergence of what might best be called *procedural epistemology*—the study of the structure of knowledge from an imperative point of view, as opposed to the more declarative point of view taken by classical mathematical subjects. Mathematics provides a framework for dealing precisely with notions of "what is." Computation provides a framework for dealing precisely with notions of "how to."[14]

Abelson imagines future historians looking back at our current age with some amusement at our shortsightedness concerning computers, as we

fetishize the specific tool of the computer. He notes that just as geographers' work was once identified solely through their surveying equipment, the technological and material presence of the computer has overshadowed its ability to encapsulate how-to knowledge.[15] And, of course, this knowledge was present in the concept of programming before computers existed: Ada Lovelace's description of how one would program Charles Babbage's Analytical Engine walks carefully through the procedures to make the machine function.[16] Because we are forced to make the procedural, or how-to, knowledge highly explicit through code in order to communicate with the computer, this knowledge, which was present but tacit in many human activities prior to the computer, is laid bare in programming.

We can connect this procedural knowledge in programming to the claims that programming enables certain kinds of thinking, which were introduced in chapter 1. For example, Seymour Papert calls programming an "object to think with," likening it to the gears he played with as a child and which later allowed him to have a feel for engineering and physics.[17] A student of Jean Piaget, Papert argued that programming is a kind of "scaffold" for knowledge, a way to boost thinking. Programming is a technological version of teachers who help kids understand concepts just beyond their reach—concepts within their Piagetian "zone of proximal development." Through exercises where children "programmed" each other to navigate paths, Papert taught them more sophisticated spatial and logical reasoning than they were capable of on their own. Similarly, Papert designed his Logo programming language to be an "object to think with." Beginning with pilot projects in a handful of British schools in the 1970s, the Logo programming language expanded its reach into American classrooms through defense-related education spending in the 1980s. Scaled up and distributed across thousands of classrooms and teachers, Logo was less effective at teaching students the concepts that Papert was able to teach them in his pilots. As this educational experiment seems to suggest, the Logo programming language—by itself—was not an object to think with. Like any technology, it could not act independently to alter cognition. Among other things, Logo needed to be combined with good teachers and support to help kids think about procedures and problem solving.[18]

Programming fosters procedural knowledge, but as the example of Logo demonstrates, programming does not and cannot do this on its own. The social milieu and the notation in which code circulates affect the kinds of thinking it fosters, who does this thinking, and how widespread it can become. Programming languages matter. Teaching matters. These points were forcefully made by Roy Pea and Midian Kurland in their 1984 critique

of the contemporary hype about teaching Logo in elementary school classrooms.[19] Pea and Kurland note that the cognitive benefits of programming are the same as those that were once attributed to math, Latin, writing, and logic. Yet the claim that "spontaneous experience with a powerful symbolic system will have beneficial cognitive consequences" is merely "technoromanticism" and cannot be empirically verified.[20] The main problem, they say, is that these claims treat programming as a "unitary skill." However,

> Like reading, [programming] is comprised of a large number of abilities that interrelate with the organization of the learner's knowledge base, memory and processing capacities, repertoire of comprehension strategies, and general problem-solving abilities. … As reading is often equated with skill in decoding, "learning to program" in schools is often equated with learning the vocabulary and syntax of a programming language. But skilled programming, like reading, is complex and context-dependent.[21]

In other words, programming—like reading or writing—inextricably intertwines the technological, the social, and the cognitive. Following up on his work with Papert, diSessa appears to have taken some of Pea and Kurland's points into consideration in his concept of material intelligences.

A Sociomaterial Approach to Writing Technologies

Pea and Kurland's critique of the claims for Logo, as well as diSessa's (indirect) response, emerge from a broader context of debates about the effects of technologies on individual cognition and society. This has been tricky territory for anyone wanting to make claims for literacy and writing, as evidenced by the controversy around Goody, Ong, and McLuhan's claims. Although popular discourse still might accept those claims uncritically, it is very difficult to prove the effects of writing on thinking. My argument about the sociomateriality of programming and writing differs from these now passé and technologically deterministic theories about writing (and programming) because of its attention to social factors and its silence on cognitive "consequences." It is useful to review where these ideas came from, however, and how we got sociomaterial theories of literacy as a corrective.

Harold Innis argued in the 1950s that governments whose laws were written on paper had different characteristics than those whose laws were etched in stone tablets. There appeared to be different "biases" for these media. For example, tablets were more difficult to carry long distances than paper, and so paper governments had a larger geographic range.[22] His student, Marshall McLuhan, took the idea of "bias" in communication further,

asserting that materiality was everything, that "the medium is the message."[23] Walter Ong zeroed in on writing specifically, arguing that it is "a technology that restructures thought."[24] Jack Goody and Ian Watt published an (infamously) influential paper in 1963 on "The Consequences of Literacy," which credited literacy, specifically the ability to read and write the alphabet, with the development of Western civilization.[25] Although Goody and Watt had implicitly intended to counter the arguments that certain societies were racially superior to others by locating successes in their writing technologies rather than the inherent characteristics of their people, their argument about the superiority of alphabets has largely been discounted as ethnocentric, and Goody went back to revise these claims significantly.

These early claims for writing and literacy were decimated by the social turn in writing studies. Influenced by larger movements of postmodernism and poststructuralism, social theories of writing pointed out that communication is not fully determined by its technologies. The way writing is learned, culturally framed, and individually performed also shapes what might be written. Brian Street helped to knock down the so-called theoretical "great divide" between oral and literate societies with his "ideological model" of literacy, which posited literacies as multidimensional, continuous, and inflected with value systems.[26] In an extensive study of the Vai people in Liberia, Sylvia Scribner and Michael Cole demonstrated that only modest changes in thought could be attributed to literacy.[27] The work by "New Literacy Studies" (NLS) researchers such as Street, Scribner, and Cole moves literacy from the value-neutral, technical continuum of skills that culminates in a narrowly Western tradition—as depicted in Goody and Watt's account—and recovers the sociocultural aspects of literacy. Thus, after the work in NLS, we can no longer go so far as to mark the "literate man" as fundamentally different from "oral man," as McLuhan claimed. McLuhan and Ong and Goody emphasized technologies at the expense of attending to the ways that they were adopted and circulated in societies.

Yet, surely it makes a difference if we type on our cheap laptops with QWERTY keyboards instead of painstakingly handwrite with quills that need constant sharpening and expensive ink? There are bound to be differences in what we say in 140-character messages on smartphones and letters written longhand and mailed. These questions prompted by new technologies of reading and writing have led scholars to revisit Ong, McLuhan, and others.[28] As part of a larger "material turn" in the social sciences and humanities, studies of writing and communication are attending to the materials of composition: the codex format, the history of paper, the

surfaces of physical computer memory, the platform of the e-book.[29] But instead of resuscitating McLuhan's "electronic man" they ask: What might it mean for people to use digital modes to communicate? Deterministic histories of technologies such as McLuhan's are no longer tenable, but, as Bruno Latour has convincingly argued, neither are purely social histories.[30] In place of "the social" or "the consequences" of technology, we have *social* histories of technology, "new materialism," and "media archaeology."

However, a closer look at the work in NLS reveals that this material focus was there all along.[31] In other words, those interested in the nature of writing have long understood writing as a technology whose social effects depend on its material manifestations. The problem with determinist or "great divide" theories was not that they were wrong to attend to the technologies of literacy, but that they were not specific enough to be useful.[32] David Olson, whose early work was accused of reproducing the "great divide" between oral and literate cultures, later retracted his boldest claims and wrote, "The excitement of McLuhan's writing springs from its sheer scope; the particulars of his hypothesis regarding oral man, literate man, electronic man, and so on continue to be apt metaphors but have limited theoretical use. They fail, I believe, not because they are false, but because they do not indicate precisely how writing or printing could actually have produced those effects."[33] In Olson's revisions, we can see echoes of Pea and Kurland's argument that treating programming as a "unitary skill" dismisses so much of its complexity as to dismiss any claims for its benefits.

New Literacy Studies offers a nuanced and careful look at the ways writing intersects in complex ways with social situations. Work in NLS that attends more specifically to the technological as well as the social includes Deborah Brandt demonstrating the palimpsest of twentieth century literacy technologies in contemporary American lives; Niko Besnier charting the global circulation of English literacy as manifested in such material objects of capitalism as T-shirts worn by people on the island of Nukulaelae; Christina Haas describing differences in navigating texts on computer screens versus paper; Kate Vieira exploring the ways document technologies impinge on literacy practices in immigrant communities; and Timothy Laquintano revealing the strategies of e-book authors to restrict and expand the circulation of their work.[34] For these researchers, literacies circulate in technologies such as pens, papers, and dictionaries; in symbolic systems such as English, alphabets, and genres; and in networks such as workplaces, families, schools, and the global post. Literacies are plural, multidimensional, heavily inflected by orality, acquired along with value systems, *as well as* intertwined with the technologies in which they are enacted.

This sociomaterial perspective on literacy resonates with what social history of technology scholar Thomas Misa calls a "softer" approach to the influence of technologies on society. Rather than the "hard" determinism of McLuhan and rather than "consequences" of technologies, this "softer" approach uses the concept of technological affordances. It is represented by technology theorists such as Ruth Schwartz Cowan, who claim that technologies have certain politics and uses embedded in their design—what Innis might have called "biases."[35] Misa writes, "Accounts such as these are not content merely to explain the 'social' by the 'technical' (technology as a social force), or the 'technical' by the 'social' (technology as a social product). These accounts shift to an interpretive framework presenting technology at once as socially constructed and society-shaping. They view technology as a social process."[36] It is this approach that I see present in sociomaterial perspectives on literacy such as diSessa offers with his three-pillar model of material intelligences.

So, when I say that programming builds new knowledge, I do not mean that the technology of programming inevitably leads to certain cognitive or societal effects. As a technologically mediated symbolic system, programming makes some kinds of thinking more available than did previous technologies of communication. Through the ways it enables people to structure and express information, it uncovers certain kinds of tacit knowledge. Here, I follow Heidegger in his assertion that technologies can *reveal* certain things latent in their design and implementation. By looking simultaneously at the social and the technological aspects of programming, I draw on sociomaterial theories of literacy and attempt to avoid the determinism of McLuhan and the Toronto School. Programming and writing are both socially shaped and shaping technologies that have "become in a very real sense part of our thinking, remembering, and communicating."[37]

Programming Is/and Written Language

Some shared characteristics contribute to the ways that programming and writing both help us build knowledge. Below, I outline some of these shared characteristics and overlaps, beginning with the historical and technological process that turned programming into writing. (With the caveat, of course, that Lovelace and Turing both programmed in writing ahead of the machines that could run their code.) This transformation meant that programming could merge writing's capabilities of scale, replication, and distribution with engineering's capabilities to control machines. I then trace the evolution of programming languages further into the ways they

balance human and computer time and legibility. As programming languages that more closely resemble human language emerge, can we maintain our theoretical distinctions between programming, writing, and speech? Yes and no. Speech act theory helps to explain and complicate those boundaries.

A Brief and Incomplete History of Programming Languages

The earliest mechanical and electrical computers were not controlled by code at all: they relied on physical engineering rather than writing to program them. They were more closely related to ship navigation instruments or machines of industrial production such as the Jacquard loom. Then, intermediary systems were developed that allowed people to write something that resembled English yet still communicate to the computer in binary numbers. As programming languages and these intermediary translating systems grew more sophisticated, programming code became more abstract, more symbolic—less about the constraints of the machine, and more about the processes the programmer wanted to enact. This evolution allowed nonspecialists to harness the computational power of the digital computer. It also brought programming into a closer relationship with writing.

Computers in World War II such as the Mark I at Harvard, the ENIAC at the University of Pennsylvania, and the Colossus at Bletchley Park carried out their calculations on relays, wires, or vacuum tubes. These computers were used for massive mathematical calculations, such as breaking code or weapons ballistics, and for each new calculation, the machines had to be reconfigured. Complicated calculations had to be worked out using basic functions such as adding and subtracting, and these operations would be assigned specific sequences by a circuit designer. When working on the ENIAC, John von Neumann's team developed a concept of a "stored program," which would allow the computer to store its instructions along with its data.[38] This design meant that computers did not need to be rewired for each new calculation. Computers could then be general-purpose machines; that is, they could be used for many different calculations without being physically manipulated. The implications of this development were huge: "von Neumann architecture" (as it is now called) moved the concept of programming from physical engineering to symbolic representation. Media theorist Friedrich Kittler locates this transition earlier; he argues that when Konrad Zuse designed the first programmable computer in 1938, "the world of the symbolic really turned into the world of the machine."[39] With these developments, programming became the manipulation of code, a symbolic

text that was part of a writing system. Computers became technologies of writing as well as engineering.

Although many of the fundamental concepts of programming were in place at this time, there was no difference between source code and machine code. Computers then and now really only understand two commands: off and on, which are usually represented by 0 and 1. The concept of binary programming has roots in nineteenth-century mathematical logic by George Boole and Claude Shannon's 1937 adaptation of those theories to communications. Represented in ones and zeros, binary operation looks like this:

```
11000000 0110010111001010 1101001101100100
```

where "11000000" is the operation ADD and the next two numbers (represented in binary format) are what the computer adds together.[40] Binary code was a direct representation of the on/off states of the vacuum tubes translated into ones and zeroes. With the introduction of octal and hexadecimal numerical representations, programs became slightly more legible to programmers, but were still fairly arcane.[41] Then, in 1948, a group at Cambridge figured out that the computer could be made to understand letters as well as numbers. Letters could be made into numbers: human-readable source code could be translated into computer-readable machine code through an intermediary program they called an "assembler."[42] Following is a more contemporary assembly language procedure to determine whether a number is higher or lower than average[43]:

```
cmp edx,32h
jle wmain+32h
push offset string "Higher than average\n"
jmp wmain+37h
push offset string "Lower than average\n"
call dword ptr [__imp__printf]
```

Assembly programs are specific to the architecture of a particular machine. Although they are more readable than binary or hexadecimal code, they still require programmers to assign specific, physical memory locations to data (e.g., "wmain+32h") and are therefore determined by hardware architecture.

In the 1950s, compilers, a new generation of translation programs, allowed source code to be imbued with more semantic value by making the computer a more sophisticated reader—and writer—of code. They were part of an effort then called "automatic programming," which strove to

eliminate programmers by shifting more of the burden of the program to the computer. As Grace Hopper articulated in 1952, "It is the current aim to replace, as far as possible, the human brain by an electronic digital computer." (She later softened this stance, and instead outlined various roles and hierarchies for programmers.)[44] The legacy of automatic programming includes government funding from DARPA in the 1970s and 1980s and some design aspects of the Java programming language in the 1990s.[45]

Working on the UNIVAC computer, Hopper wrote the first compiler in 1951, A-0. She credited ENIAC programmer Betty Snyder (Holbertson), who wrote a sort-merge generator—the first program that wrote a program—as inspiration for her compiler.[46] Janet Abbate notes that women were disproportionately well-represented in these early efforts to streamline programming, perhaps because they were often the operators of computers in this era. The first successful higher-level language to work with a compiler—FORTRAN (now Fortran)—was released in 1957 by John Backus at IBM.[47] The language was quickly adopted by IBM programmers as it allowed them to write code much more quickly and concisely than assembly language did, and it had a significant influence on other programming language development. Here is a snippet of Fortran code:

```
A = (x * 2) / Y
```

Anyone familiar with mathematical notation can read this as *A* equals *x* times 2, divided by *Y*.[48] In Fortran, like most programming languages, the equal sign (=) refers to assignment rather than equation; therefore, this statement *assigns* the value of (*x* * 2) / *Y* to the variable *A*. In assembly language, the loading, storing, and dividing of the numbers would have been separate instructions, but Fortran allows the steps to be combined in a more familiar and legible notation. Fortran, which stands for "formula translation," points to the primary use of computers at the time, to calculate complex formulas, but it also gestures forward to an idea of programming less tied to the field of engineering or mathematics. Fortran's more legible notation meant that programming could be done by nonprofessionals; physicists, chemists, and other scientists could and did use the computer without learning specific assembly languages. In other words, Fortran allowed programming to move outside of the exclusive domain of computer specialists.

LISP (now Lisp) was developed shortly afterward by John McCarthy at MIT and has been the vehicle for many language developments, such as dynamic typing and conditional statements. On the heels of Lisp and Fortran was COBOL (Common Business Oriented Language), which was based

on Hopper's B-0 compiler and designed in 1959 by a committee of representatives from government and computer manufacturers to read with English-like syntax. The COBOL equivalent of the above Fortran statement is:

```
MULTIPLY X by 2 giving TEMP
DIVIDE TEMP by Y giving A
```

COBOL was friendlier to report layouts and large data structures and used more English words to represent orders, so it was more accessible to managers and businesses.[49] COBOL was also designed to be platform independent—that is, to work across a number of different manufacturers' machines—in order to streamline the local and idiosyncratic practices of programming that had cropped up around shared machines. COBOL pointed toward increasing standardization in programming but also suggested that the computer had become a more useful and versatile tool for many different fields, including business. COBOL became an industry standard not only because of its committee-based design but also because the U.S. Department of Defense announced in 1960 that it would not lease or purchase a computer without a COBOL compiler.[50] The standardization for COBOL process emphasized corporate and management interests over the interests of academics and individual programmers.[51] Registering their dissatisfaction with business-biased COBOL and IBM-controlled Fortran, the Association for Computer Machinery developed the more "elegant" language ALGOL in 1962.[52] ALGOL never achieved the popularity of its rivals, in part because of the Department of Defense order and other social pressures that encouraged adoption of COBOL.

In the 1960s, the social influence on the design of programming languages is visible in the ways that COBOL and ALGOL accommodated the increasingly collaborative work of programming. COBOL did this through its widespread adoption by Department of Defense fiat, and ALGOL fostered collaboration through its structured design. "Structured programming," pioneered in ALGOL, is a design that allows programmers to trace the path of a program's execution. For any given line of code, the programmer can tell where the program came from and where it flows next, making programs easier to follow and debug. Structured programming also lends itself well to modularity, so that programmers can divide a program into different sections of code on which they can work individually before reassembling them. Languages with standards and structure, such as COBOL and ALGOL, helped increasingly large programming teams to collaborate. Structured style was key to the professionalization of programmers in business.[53] However, this structure enforced rigidity around styles

of programming, which seemed to attract or enforce certain kinds of people working as programmers. Sherry Turkle argues that structured programming, especially in the 1970s and 1980s, emphasized "a male-dominated computer culture that took one style as the right and only way to program."[54]

Another important development in programming languages emerged from academic artificial intelligence research in the late 1950s, represented by Lisp. Lisp's innovation is that it is unusually flexible. Using logical principles from Alonzo Church's lambda calculus, it does not draw distinctions between data and procedures—roughly, the nouns and verbs of computer programming languages. Lisp, which stands for "list processor," uses linked lists to structure both data and processes.[55] The ways that Lisp handles data and procedures highlights the recursive and symbolic aspects of programming—which are also features of human language, of course. Lisp's features made it challenging to implement with the hardware at the time, but many of its ideas have been absorbed in subsequent languages. Versions of Lisp continue to be popular in education, artificial intelligence, and for scripting languages, and it can boast high-profile acolytes such as Paul Graham and Richard Gabriel, who writes: "Lisp is the language of loveliness. With it a great programmer can make a beautiful, operating thing, a thing organically created and formed through the interaction of a programmer/artist and a medium of expression that happens to execute on a computer."[56] Abelson, Sussman, and Sussman's popular *Structure and Interpretation of Computer Programs,* designed for the MIT introductory computer science course, uses a version of Lisp called Scheme.

As computers became cheaper to own and to run in the middle decades of the twentieth century, programmer time became more valuable than computer time, and languages were increasingly biased toward human comprehension over computational efficiency. This emphasis on human comprehension meant that programming language design has trended away from the materiality of the device and toward the abstraction of the processes that control it.[57] Defined more by its social achievements than its technical ones, the BASIC programming language developed in the 1960s by Dartmouth professors John Kemeny and Thomas Kurtz (discussed in chapter 1) represented this move toward greater human comprehension. BASIC represents the beginning of a lineage of programming languages geared toward novices, including languages such as Logo, Etoys, Alice, and Scratch. Using many different approaches—combinable text blocks, English language syntax, graphics—novice languages have aimed to make concepts of programming easy initially, although most of them are not meant

to be used in professionally or long term.[58] Other visual programming languages such as Blockly, Node-RED, Max/MSP, and Modkit make specific tasks easier, such as creating apps for mobile phones, programming the so-called Internet of Things, Arduino control, and music generation.[59]

A focus on human-friendly abstraction is illustrated in object-oriented languages such as Simula from the Norwegian Computing Center in the 1960s or Alan Kay's Smalltalk developed at Xerox PARC in the 1970s. Object-oriented languages "black-box" some of the details of programs in order to allow programmers to plug in "objects"—encapsulated packages of functions and data—to build even more complex programs.[60] In the 1990s, increasingly robust language libraries in object-oriented languages such as Java allowed individual programmers to string together long series of functions written by other programmers, creating complex programs that could run on almost any machine. This practice of writing with language libraries complied with the need for programming modularity in businesses of the era, but also has allowed less experienced programmers to write programs that build on the work of other, more specialized programmers.

The 2000s saw an upswing in the social circulation of languages such as Python, Javascript, and Ruby, with accompanying frameworks such as Ruby on Rails or Django that allow programmers to do specific tasks in extremely streamlined ways. Continuing the trend of valuing programmer time above computer time, these languages allow programmers to write and run code very quickly, although they tend to run less efficiently. These languages have grown in popularity because they are well suited for Web applications, where time for network or database access rather than code-running efficiency is the limiting factor.[61] These languages thrive in the communication bazaar of the Web, collecting communities and code libraries that enrich them. They also interact with the markup language of the Web, HTML (Hypertext Markup Language). Their ease of use and implementation in addition to their applicability to the Web and interaction with HTML means they are often learned by nonprofessional programmers in more casual contexts.

Programming languages make different trade-offs between freedom and constraint and offer different affordances for creativity, safety, standard means of expression, and reliability to programmers working in various contexts. These trade-offs are often related to the legibility of code for its dual audiences: the computer and the programmer. For example, the legibility and ease of learning for novices are reasons for the BASIC programming language's popularity, although it is not a particularly fast or efficient

language for the computer to execute. These trade-offs are visible in the evolution of programming languages I discussed earlier, but these trade-offs also apply across the thousands of languages currently available. Software running in Ruby might be easier to write, but it tends to run more slowly than software in the C programming language, in part because C offers fewer "safety nets" for programmers. The C and C++ programming languages offer features such as direct memory manipulation, operator overloading, and the ability to redefine key words in a preprocessor that can help optimize programs for specific contexts, but these features can make it difficult to check errors and can decrease legibility if a programmer isn't careful. Some of these useful but "dangerous" features are not available in languages like Java. The software environment in which a person programs—often called integrated development environments, or IDEs—also make a huge difference in what is possible to say or even for a programmer to learn what it is possible to say in a language. Some IDEs highlight syntax and automatically tab and organize code in ways that are easier for programmers to read. Thousands of code languages, frameworks, libraries, and support software have features tailored to specific programming contexts and are shaped by unique technical factors. Specificities of tools matter for programming, just as they do for writing.

The technical affordances of programming languages intersect in complicated ways with the communities that use the languages and the composition spaces in which they circulate. As their trade-offs with memory manipulation and safety might suggest, C and C++ are used in memory and graphics-heavy contexts such as game programming, where efficiency is critical and programmers are highly specialized. In contrast, Ruby is useful for development of Web and mobile software applications. Java has an organizational structure that can mimic a top-down bureaucracy, similar to some of its commercial contexts for use. Niche languages such as Objective C for the iPhone and Processing for art and design circulate among specialized communities and are imprinted by the communication and composition practices of those communities. Another social factor shaping programming languages is their popularity: texts written in more common computer languages have a larger pool in which to circulate, just as texts written in more common human languages do. If a software program needs to circulate widely on the Web, it will need to be written in a language such as Javascript, PHP, or Ruby, but if a person is writing a small program for their personal use or if a development team doesn't need to circulate their code widely, they can implement their program in a less common language such as Scala or Lua or Haskell.

Any linear history of programming languages tells only part of the story. Programming languages have evolved along many paths, which alternately merge, run parallel to each other, or diverge. And popular programming languages never go away: COBOL and Fortran still have millions of lines of code running in legacy business and manufacturing software. We can still, however, make a few generalities about this evolution: over the past 60 years, designers of programming languages have attempted to make more writer-friendly languages that increase the semantic value of source code and release writers from needing to know details about the computer's hardware. Some important changes along this path in programming language design include the use of words rather than numbers, automatic memory management, structured program organization, code libraries, code comments, and the development of programming environments to enhance the legibility of code. The historically persistent trade-offs between the roles of hardware and the many layers of software point to the difficulties of drawing lines between the physical engineering of computational machines and the many different languages we use to control them. Moreover, the rise and fall and rise again of languages such as Lisp suggest that language specifications inevitably intersect with social and technical factors. Lisp was an academic language impractical for production software until computer processors could run it more efficiently. I've concentrated here on the evolution of programming languages rather than computer hardware, but as the popularity of Lisp demonstrates, the two threads are inseparable.

While programming languages have continued to evolve toward greater abstraction, they still have a long way to go to be "natural language." As the syntax of computer code has grown to resemble human language (especially English[62]), the requirements for precise expression in programming have shifted but not disappeared. Highly readable languages such as Python, Ruby, and Javascript still require logical thinking and attention to explicit expressions of procedures. This persistent requirement of precision is key to our exploration of programming as a tool for thought and to our next discussion about the relationship between programming and human language.

Language in Action

Programming became a kind of writing when it moved from physical wiring and electromechanics to a system of symbolic representation. But there are significant differences between writing and programming as symbolic systems. Programming *is* writing because it is symbols inscribed on a

surface and designed to be read, but programming is also designed to executed by the computer. Because it represents procedures to be executed by a computer, programming is a type of *action* as well as a type of writing.

The capability of code to make things happen through language has led some to present it as fundamentally different from writing and speech, as when Alexander Galloway claims, "Code is the only language that does what it says."[63] However, this distinction is as false as the "great divide" scholars have tried to make between writing and speech. Contrary to Galloway's claim, theories of language as symbolic action in rhetoric from Kenneth Burke and theories of speech acts from J. L. Austin suggest that we cannot definitively distinguish between code and traditional writing through their abilities to perform actions in language.[64] This is most clearly seen in the written language of the law and the language of verbal promises. But, as Burke argued, "All acts of language materialize a certain reality."[65] Programmer and essayist Richard Gabriel would agree: "Artists who create by writing or producing other representations of what the world is or could be are also laying out a map for how the world could become."[66] Textual writing and code represent as well as construct the world.

However, the ways that code can enact things in the world are quite different from the ways that text or human language can. Speech act theory can help us characterize those differences: code and text both have audiences, intent, and effects, which play out very differently between the two symbolic writing systems. In contrast to prior theories that focused on language as description, Austin noted that statements such as "I bet you a dollar that he makes the next shot" or "I now pronounce you married" or "Bring me my shoes" are not descriptive, nor are they true or false; instead, they *enact* something in the world. Or, rather, they are *attempting* to enact something in the world. Many factors can intervene to sabotage their actions; if someone does not accept the bet, if the speaker lacks the authority to marry a couple, or if the intended audience fails to hear that the speaker wants her shoes, then the implied action does not happen. Austin calls these failed statements "infelicitous." To be "felicitous," statements like these must be uttered in the right context and according to the conventional procedure they invoke, they must be stated by a person with the right authority, and they must be heard by the person to whom they are directed. Austin begins his lectures in *How to Do Things with Words* by naming such statements "performative," but he ultimately fails to find a basis to separate performative statements from descriptive statements. By the end of the lecture series, he proposes that *all* language implies a kind of action.[67]

Instead of thinking of code or human language as purely performative or descriptive, then, we can think of them as being performative and descriptive to different degrees.[68] To characterize the degrees to which language is performative and descriptive, Austin introduces three forces: locutionary, illocutionary, and perlocutionary. An act with locutionary force describes something. Illocutionary force refers to the speaker's intention in uttering the act. Perlocutionary force "is *the achieving* of certain *effects* by saying something," such as persuading or warning someone.[69] All speech is both locutionary and illocutionary, although the achievement of effects—the perlocutionary force—depends on social context.[70] Extending Austin, John Searle argued that we must consider language rules in social context in order to understand speech acts.[71] Searle draws an analogy: although baseball is governed by rules, a study limited to its formal rules would not tell us much about the game. To study the speech act is to study the whole game of baseball—not just its formal rules, but its context, culture, and participants.[72] For baseball as well as speech, context shapes what it means to perform within a rule system.

In the same way, we must look at programming as more than just formal rules. As the history of programming language use and development above suggested, programming is also a sociolinguistic system. This is, in part, because human programmers inevitably operate in social contexts. Yasmin Kafai and Quinn Burke argue that programming draws its power and worth from its circulation in social networks and communities.[73] Computers are also socially contextualized; they are objects that are both controlled by language and can be used to manipulate linguistic symbols. Exploring the connections between computer and human language, Terry Winograd and Fernando Flores argue that "computers do not exist … outside of language."[74] Their power as well as their social context is derived from the human language on which programming builds. For Winograd and Flores, the computer is controlled by code language in social contexts and can be used to produce socially embedded text. These linguistic objects are speech acts, they claim: code language makes commitments and changes in the world just as human speech does. And just as human speech has both locutionary and illocutionary forces, so does computer code.[75]

Yet computer code does not collapse completely into human language. Code has separate audiences for its descriptive and performative functions: programmers must imagine the procedure's performance to read and write the description of a procedure, but the computer actually performs it. While writers can and do describe procedures to human readers in documents such as operation manuals, procedures are one of several "relatively

nonverbal representations [that are] difficult, but sometime necessary, to capture in words," according to writing researchers Linda Flower and John Hayes.[76] Part of the difficulty in capturing procedures in prose lies in the author's need to assume what the audience already knows and what they must learn to enact the procedure. If the reader does not have enough information or does not want to be persuaded, the perlocutionary effects of speech or text, as Austin reminds us, can be infelicitous. The social properties of language can cause infelicitous interactions, but they also facilitate literature and human communication as we know it. However, in programming, the computer only "knows" what it is told; it has a predictable perlocutionary force. As Ada Lovelace wrote in her notes to Charles Babbage's protocomputer, "The Analytical Engine has no pretensions whatever to *originate* anything. It can do whatever we *know how to order it* to perform."[77] With a perfect explanation of procedures, a program is automatically felicitous—nebulous social factors cannot sabotage it.

While this can eliminate some of the confusion about what the computer needs to be told to enact procedures, it presents a different challenge to writers: describing a procedure perfectly. Describing procedures perfectly is nontrivial, and as programs grow in complexity, the capacity for a person or team to envision the whole system and scenario for software becomes impossible. Invoking Ada Lovelace's statement about the computer's predictability, Alan Turing winked at this complexity and declared that computers did not infallibly do what programmers tell them to do; on the contrary, he wrote, "Machines take me by surprise with great frequency."[78] Code's perfect *technical* perlocutionary affordance is no guarantee of results in software.

This perlocutionary affordance of programming is both an advantage and a disadvantage of code relative to writing. In the next section, we'll explore more of these trade-offs and interactions between code and human language as forms of communication.

Affordances of Form in Code

As Wittgenstein observed, human language works not through explicit definitions or explanation, but through use and exchange. Human language, in other words, works in a bottom-up and social fashion: it gathers meaning gradually as it circulates among its users.[79] Walter Ong contrasts this property of human language with programming:

> Computer "languages," … resemble human languages … in some ways but are forever totally unlike human languages in that they do not grow out of the un-

> conscious but directly out of consciousness. Computer language rules ("grammar") are stated first and thereafter used. The "rules" of grammar in natural human languages are used first and can be abstracted from usage and stated explicitly in words only with difficulty and never completely.[80]

Programming works in a formal and top-down fashion: it gathers meaning only as terms and procedures are explicitly defined by its writers. Code must adhere strictly to set formats to be understood by computers, whereas human language can convey a "gist" of an idea even if the details are fuzzy. Along these lines, Jay David Bolter draws on Charles Peirce to claim that computer programming is an "exercise in applied semiotics," a sign system in which everything is defined in relation to another sign—a closed system.[81]

The fact that code relies on explicit definitions and form is sometimes a source of critique. Programming is misrepresented as strictly technical practice, a right-or-wrong activity with little creative potential. For example, composition scholar Joel Haefner argues that code cannot capture the nuances, double meanings, and richness of human language and experience. After translating Hamlet's "To be or not to be" statement into a yes/no Boolean statement in the C programming language, he laments that "the simultaneous dichotomy Shakespeare demands—to consider being and nonbeing—cannot exist in the text of code."[82] Although Haefner admits his translation is "contrived," it is worse than that; it is misleading. In fact, a computer program is capable of offering far more complex representations than Haefner proposes. Defending the computer against accusations of inflexibility such as the one Haefner wages, Ian Bogost writes, "We think of computers as frustrating, limiting and simplistic not because they execute processes, but because they are frequently programmed to execute simplistic processes."[83] A more inventive way to translate the speech would be to use complex fuzzy logic common in artificial intelligence programming. But even better would be to leave poor Hamlet alone, with the understanding that neither programming nor writing can serve as a direct translation of the other. We might say that critiques of computers chopping up the world too cleanly or not responding thoughtfully to their users are modern versions of Socrates's complaint about writing as being a dull child who travels too far from its parent, the author. While it is true that Socrates's dull child cannot directly engage in meaningful discourse, Walter Ong points out that the only way we know about Socrates is through Plato's writing.[84] It might be nice to have seen Socrates in live discourse, but Socrates in writing is the Socrates that was able to survive for centuries.

Neither speech nor writing perfectly represents the other; they have different possibilities of scale, audience, and duration. Similarly, code allows for a form of representation different from writing. To measure programming by the standards of writing is to call it mechanistic and unnuanced. To measure writing by the standards of programming is to call it sloppy and unclear. From a rhetorical standpoint, we might think of the ways speech, writing, and code represent and convey information as their Aristotelian "available means." From a design standpoint, we can think of their various affordances. Although the arguments above indicate that it is difficult to distinguish between speech, writing, and programming at their borders, we still must attempt to take code on its own terms, rather than as a poor substitute for speech or writing. Below, I point to some of the affordances of code's form for representing and scaling up as well as for creative purposes.

Discreteness and Abstraction, Fidelity and Scale

Although it may not serve well to represent the nuances of Hamlet's indecision, the discrete nature of code gives it power to represent a different form of complexity. Code can be used to build complex, chained, and multilayered procedures because of its discreteness. In working code, each process is represented unambiguously from the perspective of the computer, and has a concrete location and description. The computer will interpret procedures as precisely as one writes them (not, as Turing pointed out, as one *means* them).[85] Because of their discreteness and precision, processes can be piled on each other, making complex and interconnected systems. Through loops, a program can perform the same operation millions of times in a row; through recursion, a process can call itself, nesting those loops. The discrete representation of code also enables one to copy a procedure from one program and paste an exact copy into another program or to use code libraries. The exact same results of the code are not guaranteed in this new context, but this perfect transfer of process is something to which written instruction manuals can only aspire. This perfect transfer is even more difficult in speech, as the game of "Telephone" attests. Code can convey and transfer information about procedures across space and time on a larger scale than speech or writing.

Of course, writing is also discrete. In written English, 26 letters plus punctuation are routinely used to convey complex ideas on a large scale. Attending a performance of *Hamlet*'s original staging is no longer possible, but we know about *Hamlet*'s text because its text can travel across space and time. To travel in this way, the written text of *Hamlet* necessarily eliminates

the gestural, vocal, nuanced complexity of *Hamlet*'s staging. And, as any letter writer or Emily Dickinson scholar will attest, the translation of handwritten text to print also sacrifices meaning. With this sacrifice, print enables multiple and *identical* copies of a text to circulate beyond a writer's immediate contexts.[86] The availability of photocopiers in the 1960s meant that handwritten text could also circulate in facsimile editions, although it may say something about written language that few print publications call for this fidelity.[87]

The trade-offs between fidelity and scale in continuous and discrete representations of processes go beyond just speech and writing and have been central to philosophical discourse about industrial processes since Karl Marx. Max Weber, drawing on Marx and commenting on industrial capitalism's reduction of information and action in order to scale up production, feared that this "rationalization" of practice would result in a dampening of human spirit. Marx frames this separation of worker from knowledge about processes as political and material; Weber pinned it to the faceless grind of industry. While machines discretize human gesture, writing does the same for *logos*, philosopher Bernard Stiegler claims.[88] Stiegler's concept of *grammatization* takes the concept of rationalization from theories of industrial processes to symbolic representations such as programming and writing. He draws on Jacques Derrida's concept of grammatology, Sylvan Auroux's "grammatisation," and Heidegger's theories of technics to claim that grammatization "concerns all processes of discretization of the continuous."[89] He describes the grammatization of industrial machines—machines that discretized the gestures of human factory workers—alongside the grammatization of writing. Writing, like these machines, discretizes language and thought, breaks it up into words and sentences. In this way, writing throws out gesture, nuance, and tone of speech. But in exchange for this fidelity, it can scale; writing can be perfectly replicated in other contexts.[90]

Stiegler draws this grammatization process forward from writing to digital technologies and points out its benefits and drawbacks. To explore the trajectory and implications of digital grammatization, he takes a cue from Weber and traces rationalization back to the Enlightenment. He observes that the regularization of science led to repeatable experiments and an explosion of knowledge. But there was another side to this rationalization. As Adorno and Horkheimer argued, the Holocaust was another result of "the rationalization of the world," with radically different implications than the Enlightenment.[91] The Holocaust parsed humanity into "rational" and discrete categories and allowed for a devastating othering of certain human categories. Just as Weber feared a loss of humanity through

industrialization, Stiegler argues that digital technologies threaten our mind and memory through the way they externalize it, discretize it, and displace "the infinities" that make human life possible or worth living.[92] His connection to Plato's anxiety about writing is obvious and intentional. And, indeed, by way of Derrida's explication of Plato, Stiegler calls the rationalization of information through digital technology a *pharmakon*,[93] something that is both poison and medicine. Recalling Marx, Stiegler argues that the relentless and blind forces of capitalism drive rationalization toward poison.

But to focus only on the information lost in the process of rationalization is to tell only the poison side of the *pharmakon* story. While digital technologies "rationalize" certain information and knowledge, they also reveal and enable the creation of new kinds of knowledge. In a broad, historical analysis of the Industrial Revolution, James Beniger argues that crises of control drove the rationalization of human action, which then led to new ways of organizing information. Rationalizing technologies such as the clock, factory, and train—the technologies that Weber feared were stealing our souls—allowed Western society to scale up governance of an increasing population and respond to increasingly complex demands on organization. In Beniger's terms, the "control revolution" led to the "information society." To put a finer point on these historical arguments: saying that rationalizing technologies have *affordances* is altogether different from saying they lead to either genocide or progress. Rationalizing technologies make certain things possible that were not possible before, at least not in the same way.

Like machine manufacture over individual craftsmanship, there are complicated trade-offs between the symbolic representation systems of writing over speech or programming over writing. Although analog, continuous representations such as speech or handwriting allow for infinite variation, their information does not scale well. Both writing and programming cut out some of the contextual information of their lower-order representations—they rationalize or grammatize it—in order to scale up. Programming and writing both sacrifice richer context and nuance for precision, scalability, interactivity, and widespread dissemination. These trade-offs are key to understanding their affordances. Even technologies that circulate analog representations—the photocopier, the telephone, or the tape recorder, for instance—carry only some of the nuances of their originals. But digital information scales and can travel: Daniel Kohanski notes, "Digital technology trades perfect representation of the original data for a perfect preservation of the representation."[94] This is the difference between

the bootleg concert tape, which is muddied after too many copies, and the digital MP3. Although it can never fully represent nuanced experiences of live performances, the discreteness of the digital is where the power of modern computing lies.

But code and writing do more than just *reduce* information from writing and speech; they transform it. They spatialize temporal information. In a transcribed speech, Stiegler argues,

> The grammatisation of a type of behaviour consists in a spatialisation of time, given that behaviour is above all a form of time (a meaningful sequence of words, an operational sequence of gestures, a perceptual flow of sensations, and so on). Spatialising time means, for example, transforming the *temporal flow* of a speech such as the one I am delivering to you here and now into a *textual space*, a de-temporalised form of this speech: it is thus to make a spatial object. And this is what is going on from alphabetic writing to digital technology, as shown by Walter Ong: "Writing … initiated what print and computers only continue, the reduction of dynamic sound to quiescent space."[95]

Ong uses the word *reduction* in the sentence that Stiegler highlights here, although space is not simply a reduction of time. It is something different altogether. Code, as a digital technology, follows this same process of grammatization that Stiegler describes for speech's transformation to text. But, additionally, code can *retemporalize* information. When code is written, it translates processes into text. When it runs, it turns text into process, albeit a process that is digitized, rationalized, grammatized. This transubstantiation of code is captured in its multifaceted manifestations in the law, as text *and* machine, or, as description *and* process.[96]

Jacques Derrida provides a rich way of looking at this translation of information across modes. For Derrida, writing does not reduce speech; it exceeds it. Exploding Saussure's differentiation of the word as *signifier* from the concept of being *signified*, Derrida argued that, without writing, we would not have organizing concepts at all. Derrida's surprising move in *Of Grammatology* was to posit that writing rather than speech was primary.[97] N. Katherine Hayles takes this argument one step further by arguing that code exceeds rather than reduces writing. The way that code represents processes makes us look at the world in a new way, a worldview she calls "The Regime of Computation." This new worldview sees processes as potentially discrete in the same way that the worldview we got from writing sees concepts as discrete.[98] Like Derrida and Stiegler, Hayles sees a complicated and reciprocal relationship across modes. Even if we aren't writing with the computer, and even if we cannot program, she observes that we are always working in *language plus code*: code has forever changed writing in the same way that

writing has forever changed speech.[99] The existence of multiple modes changes our rhetorical choices and how we think about them.

Derrida, Hayles, and Stiegler all posit a hierarchy of abstraction from speech to writing, with vectors of influence pointing both directions; Hayles and Stiegler take the hierarchy already established between speech and writing and elevate it to programming. If writing is a second-order semiotic system that builds on speech, programming is a third-order semiotic system that builds on writing. This hierarchy of abstraction does not imply a hierarchy of value, however. Because of the complicated relationship between these modes, we cannot convey equivalent information across them. Through the grammatization of sound, writing can excise tonal nuance and dialogism from speech, and through the grammatization of processes, programming can excise ambiguity and affect from writing or other source ideas. Both can attempt to account for this loss—through punctuation, typography, or style in writing or through fuzzy logic or human input in programming, for example. But, as this section has argued, grammatization is a transformation that provides certain affordances. We might choose to write rather than speak if we are making a complicated argument that we'd like to reach a large audience at different times, as I am doing in this book. We might choose to program if we want to simulate a complicated system that we would like for many others to engage with, as computer game makers do. The way that code grammatizes processes means that it facilitates procedural ways of thinking and allows new kinds of information to circulate in new ways. The availability of speech, writing, and code for composition makes for complex rhetorical choices: none is now a default, and all offer different, intermediated affordances. Complex literacies—which include the ability to engage with speech, writing, and code—may now be necessary to make these rhetorical choices.

Creativity with Code

Although code is often written primarily for its functional value (as read by the computer) rather than its aesthetic value (as read by other programmers), its dual audience introduces creative possibilities. Creative exploits in code, including what is sometimes called "codework," esoteric languages, and "pseudocode" poetry-code hybrids, demonstrate that it can accomplish some of the imaginative work of writing and speech. However, code's dual human and machine audience and its unique affordances of form allow for different kinds of creativity than these other modes. Significant work in critical code studies and software studies has been dedicated to these practices, in part because the creative affordances of code vis-à-vis writing can

be most clearly illustrated through these literary-leaning examples. Creative code plays with the textual form of code and the machine interpretations of its processes, highlighting the tension between code's form and function as well as its audiences.

While these creative software and coding practices are only small slices of the landscape of contemporary composition in code, everyday coding is also a creative endeavor, a process by which people learn to think in new ways—although it's not always framed that way. Richard Gabriel complains, "Writing and programming are creative acts, yet we've tried to label programming as engineering (as if engineering weren't creative)."[100] As an example of creativity possible in coding style, Crista Videira Lopes's *Exercises in Programming Style,* inspired by Raymond Queneau's *Exercises in Style*, presents one common programming problem with 33 different constraints, each generating a different solution in code.[101] Creative code thus points to a much broader phenomenon about the affordances of code, the value of different constraints, and programming's complicated relation to human and machine audiences. As Mateas and Montfort argued, the play inherent in obfuscated and weird languages "refutes the idea that the programmer's task is automatic, value-neutral, and disconnected from the meanings of words in the world."[102] This connection of code to "words in the world" reminds us that programming of all kinds is imbricated in the social world where programmers learn and practice their craft.

As Videira Lopes suggests, code, like poetry, gains some of its expressive capabilities from the fact that its form is restricted.[103] For human readers of poetry and code, the structure helps them to interpret and appreciate the writer's creativity within the form. Referring to the parallels between the restricted forms of code and art, former IBM programmer and manager Frederick Brooks argues that "form is liberating."[104] Constraints engender creative solutions, and programs and poetry both offer flexible tools with which to work creatively within those forms. In an often-quoted passage, Brooks writes, "There is a delight in working in such a tractable medium. The programmer, like the poet, works only slightly removed from pure thought-stuff. … Few media of creation are so flexible, so easy to polish and rework, so readily capable of realizing grand conceptual structures."[105] Brooks echoes the concept of programming as knowledge-building in his vision of it building grand conceptual structures. He goes on to distinguish the programmer from the poet in that the programmer can create tangible things: "The program construct, unlike the poet's words, is real in the sense that it moves and works, producing visible outputs separate from the construct itself."[106] But as we saw in our exploration of speech act theory and

poststructuralist theories of language, we cannot draw lines between programming and writing so cleanly.

For standard programming environments, the objective is generally to produce code that is clear and consistent for both the computer and the human reader, to omit unnecessary processes for the computer and make active processes transparent to human readers. But even standard code can be poetic—"elegant"—when meaning and form merge in aesthetically pleasing ways, as Brooks suggests. Emphasizing the aesthetic value of code for human audiences, the influential computer scientist Donald Knuth famously conceived of "literate programming," arguing "Literature of the program genre is performable by machines, but that is not its main purpose. The computer programs that are truly beautiful, useful and profitable must be readable by people."[107] Through the idea of literate programming, Knuth has focused on human readers of code, promoting the idea of programming as an aesthetic as well as functional activity.

In their article on "obfuscation, weird languages and code aesthetics," Michael Mateas and Nick Montfort focus on these trade-offs between human and computer legibility in order to highlight some of the potentials of code as a form of creative expression. Mateas and Montfort argue that traditional values in programming—elegance and clarity—must be expanded to include the alternative aesthetic values and rhetorical purposes of playful code exemplified in "weird languages." Weird languages are not generally useful for business or production, though the ways they communicate different messages to computers and humans highlight some of programming's affordances. Pulling from theories of aesthetics and communication, Mateas and Montfort argue that "All coding inevitably involves double-coding. 'Good' code simultaneously specifies a mechanical process and talks about this mechanical process to a human reader."[108] They draw parallels from computational double-coding to words that have meanings in more than one language (e.g., *dire* in both French and English); we could extend their parallel to slang words and puns, which can push on double meanings to index different audiences.

The Shakespeare Programming Language (SPL) draws particular attention to the double-coding of programs for humans and computers. Here is a program, or "scene," from the language[109]:

```
Act II: Determining divisibility.
Scene I: A private conversation.
Juliet: Art thou more cunning than the Ghost?
Romeo: If so, let us proceed to scene V.
```

```
[Exit Romeo]
[Enter Hamlet]
Juliet: You are as villainous as the square root of Romeo!
Hamlet: You are as lovely as a red rose.
```

This function, or "scene," in SPL may not be any more poetic than Haefner's "To be or not to be" Boolean speech in C, but it is more inventive. In SPL, variable names are Shakespearean characters—in this case, Romeo, Juliet, and Hamlet—and constants are nouns such as "rose" and "codpiece." SPL's compiler is built with a list of nouns designated 1 or –1. Nouns with a negative connotation are assigned the value of –1 and prefixing them with adjectives doubles the number; "sorry little codpiece" then has the value of –4.[110] The SPL language allows for conditional jumps and value comparisons, as we can see in this example; scene V is a specific place elsewhere in the program, and "art thou more cunning …" is a value comparison. The Chef programming language is another example of double-coding: Chef programs are simultaneously recipes and machine instructions. The point of SPL, Chef, and other weird languages is not to provide ways to solve problems effectively in code, but to play with the borders between human and computational interpretations of code. As Geoff Cox and Alex McLean argue, these languages show that "rendering speech or code as mere written words fails to articulate the richness of human-machine expression."[111]

Like weird languages, so-called obfuscating or esoteric code languages are another creative play with the form of code. Rather than resembling human genres like plays or recipes, they double-code by being very difficult for programmers yet perfectly interpretable by the computer. Languages such as Brainfuck remove the niceties of human legibility, such as meaningful variable and function names and white space, to produce code that is nearly impossible for people to read. A Brainfuck "hello world" program—a program that simply prints "hello world" on the screen—looks like this[112]:

```
++++++++++[>+++++++>++++++++++>+++>+<<<<>++.>+.++
+++++..+++.>++.<<+++++++++++++++.>.+++.------.---
-----.>+.>.
```

Malbolge, another obfuscating language, plays with the convention that source code is a static text. It is self-modifying, which means that not just the data is manipulated through the course of the program, but the code also changes on the basis of its own instructions.[113] Here is a Malbolge "HELLO WORld" [sic] program[114]:

```
(=<`$9]7<5YXz7wT.3,+O/o'K%$H"'~D|#z@b=`{^Lx8%$Xmr
kpohm-kNi;gsedcba`_^]\[ZYXWVUTSRQPONMLKJIHGFEDCBA
@?>=<;:9876543s+O<oLm
```

Although the language was specified in 1998, it took 2 years and complex cryptography and artificial intelligence techniques to write a program in it.[115]

Software performances like algoraves and the demoscene also speak simultaneously to the human and machine audiences for code. In algoraves, people perform by coding live on stage, generating visuals and sound.[116] The audience may not understand the code that drives the music, but the code—projected onto screens on the stage—is part of the performance. Sometimes even the live-coders don't understand why the algorithms generate particular audiovisual output, but the glitch and the bugs are part of the aesthetic.[117] Demoscene code is also performative at the nexus of code, visuals, and sound, but its code is honed and revised. Demoscene programs have highly compacted source code—often only a few kilobytes of data—that result in very sophisticated audiovisual "demonstrations." Demosceners show their skills by pushing archaic computer processors such as that of the Commodore 64 to their limit with graphical demos, usually in complex animations and audio. The aesthetic values of the demoscene are dependent not only on the resulting visual and audio display but also on the cleverness of the code tricks and processor used to create the display. Algorave and demoscene performances demand that audiences acknowledge both the human-accessible output and the computational processes that drive it. The code isn't just driving the art; it is inseparable from the art.

Attention to form is essential to working with any creative object, but this attention is especially critical for reading or analyzing code-based creative objects. Without this attention, Mateas and Wardrip-Fruin both argue, we cannot fully account for the interactive aspects of a piece of creative software.[118] Espen Aarseth's early work with electronic literature provides one analytical framework that accounts for a text's code as well as its output; he named the two different writings that result in dynamic, digital compositions "scriptons" and "textons." Scriptons are essentially the code that runs the creative works, and textons are the visible, human-accessible outputs of the scriptons. Aarseth calls the combination of these writings "ergotic literature."[119] Similarly, in the field of composition, David Rieder argues that we need an approach that "recognizes interplay and upholds the distinction between the two species [of code and text]."[120] N. Katherine

Hayles calls phenomena like this interplay of code and text a "creolization" of new media and offers "media specific analysis" as way to analyze the objects that result from this interplay.[121]

Because writing and critiquing these kinds of works—ergotic literature, weird languages, games, demos, algorave coding, and so forth—requires some understanding of both the code and its output, Mateas argues that "procedural literacy" is essential for any new media practitioner or critic. Mateas goes further to say that procedural literacy is important for people beyond just those associated with creative new media projects—it's an essential skill for everyone.[122] Echoing the arguments for coding literacy that we saw in the past chapter, Mateas brings us full circle, connecting creative coding with the code that we see structuring so much of our world now. Although they circulate in different spheres, "everyday code" can be inflected by the same creative play that literary code highlights. And both operate among the combined social and technical audiences of humans and machines. What's more, these examples show that coding and writing can no longer be fully separated from each other—further proof that they are intertwining in a new iteration of literacy.

Social Coding

In his extended description of code in *Bloomberg Business*, Paul Ford lucidly explains many of the technical aspects of programming, but he also attends to who writes it, how they think, and what it means for the type of software that gets written. Understanding how code works, according to Ford, includes understanding some of its social aspects. Programming is often projected as a pure meritocracy—if the code runs, you're good; if you contribute good patches to open-source software, you're good.[123] While programmers will sometimes talk about the power they accrue from working alone and communicating directly with their machine, this asocial vision of code is a myth. Networks of mentorships, work habits, and word of mouth about techniques and opportunities are just some of the social structures that surround and support programming. The machines that run software are not social in the same way that humans are, but the ways that software gets written and circulates and taken up are still shaped by social factors.[124]

In this section, I use contemporary literacy studies to think through some of the social processes that impinge on programming as a form of writing. The intersecting social and material forces in programming are different from those in writing, but theories of literacy can help us understand

how they work and how they shape the practices of programming, as well as the populations who do it.

Collaborative Coding

Ultimately, all programming is collaborative—although it is often asynchronously so. Even if they aren't working alongside other programmers physically or online, programmers work with languages, machines, and programming environments designed by others. They work with libraries of procedures or codebases or frameworks programmed by teams of other programmers. Social influences in programming are particularly apparent for a programmer entering a new coding community and learning to communicate effectively within that space. She is often faced with *legacy code*—code that constitutes an ongoing software project. This is a common situation in workplaces where software is used and written—the project may be 5 or 20 or even 50 years old or it must interact with preexisting software or hardware. Even with perfect knowledge of a particular language, a programmer must learn to work within the existing system that has been established by previous teams of programmers. Nathan Ensmenger explains how social and historical forces are at work in legacy code maintenance:

> When charged with maintaining a so-called legacy system, the programmer is working not with a blank slate but a palimpsest. Computer code is indeed a kind of writing, and software development a form of literary production. But the ease with which computer code can be written, modified, and deleted belies the durability of the underlying document. Because software is a tangible record, not only of the intentions of the original designer but of the social, technological, and organization context in which it was developed, it cannot be easily modified.[125]

Entering into a preexisting software project, a programmer must work within the history of the legacy code and the "palimpsest" of human influences on the code in order to successfully update it. Frederick Brooks likens this kind of programming to the building of cathedrals: they are built over many generations, and to maintain their structural integrity, subsequent generations of builders must bow to the original designer's grand vision.[126] The writing of code is shaped by previous practices of organization, and for this reason, large commercial codebases often change slowly.

These code palimpsests and communities recall both James Paul Gee's concept of "Discourse" and Mikhail Bakhtin's theory of speech genres, both of which have been foundational to social theories of literacy. Bakhtin argued that all speech occurs in socially shaped genres and that all

"utterances" are refractions of words from previous speakers. In the same way that Bakhtin portrays every act of communication as building on previous ones, code is always written by refracting previous code, or "utterances" from other programmers, either in the form of palimpsistic codebases or programming languages. No speaker is "after all, the first speaker, the one who disturbs the entire silence of the universe," Bakhtin writes.[127] No programmer codes as if she were the first programmer.

Even before joining a team of programmers or an established software project, programmers learn to code by following examples from others who have come before them. Programmers shape their coding styles on the basis of what they see in particular language communities or in workplaces where they first learned to work with a large body of code. Every coding community has conventions of style, and a programmer joining a new community must learn how to communicate successfully not only with the computer, but also within that community. We might think of this learning process for programming through Gee's description of learning secondary "Discourses." For Gee, people travel within and among different Discourse communities, each of which is loosely marked by common sets of practices. These practices are constantly shifting and layered with established conventions.[128] Discourses are not formally defined, but instead live within their communities, specifically within the people who inhabit these communities: "The individual instantiates, gives body to a Discourse every time he or she acts or speaks, and thus carries it, and ultimately changes it, through time."[129] Our home languages and cultures, the ones we acquire effortlessly, make up our primary Discourse communities. The Discourses we explicitly learn—such as any form of literacy—are secondary.[130]

We can see each new programmer as adding her discourse performance to the palimpsest of code and coding practices. Programmers may prefer to rewrite code in order to enact their own style and because reading other people's code can sometimes be more challenging than writing it from scratch. But because code is written in communities and programmers must collaborate across time and space to maintain it, a fresh programmer coming to a project is often constrained by the local discourse of the project. As programmers learn and work in different project discourses, they are shaped by these discourses in turn. In this way, individual programmers are also palimpsests—layered with coding practices learned in school, on the Web, and from a history of workplaces. This is also, of course, true of individual literates: we are affected by our personal history of education and reading and writing. As Web-based applications and communities that organize themselves through the Web grow to dominate the contexts in which

people learn to code, this social impact on learning may even be accentuated.[131]

A look at a couple of specific collaborative practices will help to illustrate some of the ways code and programmers are shaped by discourse and the code "utterances" that precede their work. We look here at "code review" and "pair programming," both of which are central to the software development style called "agile." Agile development has been a popular response to "waterfall" development, which is top-down software design that outlines an entire project, parses it out into separately coded subsections, and then assembles the project once these subsections are completed. The complexities of software can foil waterfall's trickle-down process, whereas agile development acknowledges the inevitability of these complexities with frequent testing and code review. Agile methodologies attempt to work *with* social factors in coding.[132]

Code review, which follows some of the same patterns as peer review in writing, is a common practice on larger software projects. Programmers put their code out to be reviewed by their coworkers, some of whom may even work in different countries. Rather than testing whether the code works, they read its text, asking questions similar to those asked by peer reviewers of writing, but with an eye toward the collaborative model of software writing. A reviewer of code, then, acts as an authentic human reader, evaluating not whether the code can be executed by the computer, but whether the code works for its intended purposes and whether other programmers will be able to read and understand how the code works.

For instance, the Mozilla Development Center notes that one of the issues a code reviewer should consider is maintainability: "Code which is unreadable is impossible to maintain. If the reviewer has to ask questions about the purpose of a piece of code, then it is probably not documented well enough. Does the code follow the coding style guide?"[133] Mozilla outlines its style explicitly in a "coding style guide" in part because it follows an open-source development paradigm, but the conventions of style are implicit in many other workplaces and coding communities, just as they are in discourse communities.[134] Here is an excerpt of the Mozilla Style Guide that enforces the community-based standards of code review and introduces it to newcomers:

> This document attempts to explain the basic styles and patterns that are used in the Mozilla codebase. New code should try to conform to these standards so that it is as easy to maintain as existing code. Of course every rule has an exception, but it's important to know the rules nonetheless!

> This is particularly directed at people new to the Mozilla codebase, who are in the process of getting their code reviewed. Before getting a review, please read over this document and make sure your code conforms to the recommendations here.[135]

Through enforcing style rules and good coding practices, code review helps to keep the code written within development communities intelligible to all participants in the community. It enforces the discourse of the community. But code review also influences the programmers who participate in the project. Programmers cite feedback from the open-source community as a primary motivator for their participation because it helps them improve their skills.[136]

In pair programming, another common social design practice in agile development, the time between code writing and reviewing is collapsed: pairs of programmers write code together. This practice not only helps programmers anticipate bugs but also helps programmers share knowledge of the particular codebase, local programming style, and general knowledge about programming. Pair-programming code is then collectively rather than individually owned, like the coauthorship common in many professional writing contexts. The pair has to understand the code together, which forces them to communicate.[137] Because of its social aspects, pair programming and its enforcement of social rules of programming can be stifling for some programmers.[138] However, many people learn efficient or good programming practices through pair programming. In a debate about the benefits of pair programming on Jeff Atwood's popular programming blog, Bryan Arendt writes, "Although it feels like running with weighted shoes at times I have been able to quickly pick up some quick tips, shortcut keys and other little tricks that I would have never known about. Also when I am back to programming all by myself it feels like I have removed the weights from my shoes. I can run that much faster after practicing with the weights."[139]

Both code review and pair programming highlight the fact that code is *simultaneously* technical and social. The computer's interpretation of code is only one factor in good programming. The legibility of code and its processes for other human readers is also critical, and the values for code reading are socially shaped.

Social Organization of Code in an Online Coding Community

Open-source software communities, in particular, demonstrate the ways that coding is influenced by social as well as technological factors. Large open-source software projects such as the Linux operating system and the

Firefox browser (developed by Mozilla, discussed earlier) have thousands of active contributors who program on their own time, on their companies' time (both sanctioned and in secret), and for school projects.[140] These communities have an array of formal and informal requirements for membership that are enforced in both the code-based permission structure and the socially situated discourse of the project. As Karim Lakhani and Robert Wolf report from a study of 684 software developers in 287 free or open-source software projects, community identity is very strong and cohesive in these projects and structures the participation of members of the community.[141] Anthropologist Gabriella Coleman describes the open-source community of Debian as being held together not only by their shared coding project but also by "a set of moral precepts—transparency, openness, accountability, and non-discrimination—that are used to establish correct procedures for technological production, licensing, and social organization."[142]

To see this in practice, we can look at the community surrounding the FreeBSD operating system, an open-source programming project that has been actively maintained for more than 20 years. FreeBSD is a UNIX-based operating system focused on security and stability, so it is often used for mission-critical servers and embedded computing in commercial devices. NASA and other areas of the U.S. government as well as corporations such as Apple take advantage of the fact that FreeBSD's lenient license does not require them to give back to the community as the more common open-source GNU General Public License (GPL) does. In the FreeBSD community and many other open-source development communities, the membership and boundaries of the community are as much about social protocol as about writing code.

In a personal interview, Mike Silbersack, a FreeBSD developer who has been part of the community for more than fifteen years, offers advice to programmers looking to join an open-source project: "The important thing is finding something you're interested in and a community you feel comfortable with. And, just like with any group, you've got to join their mailing lists, and just absorb it. Don't try to join in the conversation for the first month. You've got to, you know, you've got to understand what's going on around you before you jump in."[143] In other words, membership in open-source communities such as FreeBSD depends not simply on the quality of code written but also on how fully someone masters the discourse of the particular community. The FreeBSD project has an elaborate structure for participation involving several hundred "committers," a nine-person core team that loosely oversees the project, and the FreeBSD Foundation, whose goal is to support the project.[144] Committers can mentor new members

interested in working on FreeBSD and joining their ranks. As Mike describes it, the mentorship process requires a significant commitment from both parties but ultimately results in tighter social ties, a stronger community, and a better codebase.

The process of membership in FreeBSD and other open-source programming communities evokes Jean Lave and Etienne Wenger's theory of "communities of practice." A "community of practice" is shaped by discourse and contains masters, apprentices, and people along a whole continuum of participation levels.[145] Open-source projects often rely on a handful of strong personalities to keep them going, but successful communities have socially enforced structures that ensure new members can join as others lose interest.[146] In this way, as Lave and Wenger write, the "mastery resides not in the master, but in the organization of the community of practice of which the master is a part."[147] Mike explains, "There's no person who's in charge of FreeBSD." He goes on to describe the decentralized organization of the FreeBSD project: "Basically, once you—if you—go through the process and become a FreeBSD committer, you can go work on whatever you want, as long as people aren't objecting to it. So of course, usually you would want to build a consensus among other people in your subject area." A good community member, then, is highly cognizant of his audience—the other programmers working on FreeBSD and the end users of the software.

Being aware of both audiences involves a sophisticated understanding of the codebase as well as the use cases for it. In response to my question about what makes a good contributor to FreeBSD, Mike replied:

> First of all, if they're adding something to the system, they have to make sure—well, if it's going to be disruptive, it has to be useful to a large number of people. Socially, I think, of course they have to be aware of whoever else cares about that area they want to work on, and try to be preemptive talking to them about the changes they're going to make. And on the other hand, if they put the change in, and they get some sort of flak back from it, they have to be prepared to back the change out, which early on seems like a big deal to you. … And learn who's giving you a reasonable response, and what the consensus is, because you can't always make everybody happy.

Successful contribution of code depends on the committer's understanding of how significant a change will be, and that understanding is based on a deep awareness of the range of users for FreeBSD. In other words, a rhetorical awareness of the community's range of responses is critical to successful participation in it. Coleman notes a similar phenomenon in another open-source community, where the "technical rhetoric" that accompanies a commit can affect its status: technical rhetoric "includes a presentation of the

code, a corollary written statement, or a justification as to why no change should be made."[148]

For Mike, success in the FreeBSD project has translated into a sophisticated social and procedural awareness of audience: "Certainly it's taught me how to work with a large piece of code with lots of different developers interacting. And made me aware of how changes I make interact with other people." When he fails to fully account for one of these audiences, the community will police the infraction in writing on the collective listserv. Enforcement of shared procedural discourse norms can contribute to the sense of community. Mike explains, "I never got my hand slapped too badly. Occasionally, I did. You know I really felt like I was part of a community." In this statement, we can see how Mike connects the enforcement of community norms to the identity of the community itself.

This feeling of being part of a community is central to programmers' enjoyment of free or open-source software (F/OSS) participation, according to Lakhani and Wolf. Models of extrinsic motivation, in terms of cost-benefit calculations, have been unable to fully account for the intense participation patterns of F/OSS; Lakhani and Wolf indicate instead that the joy of doing work in a challenging and supportive community justifies many programmers' participation in F/OSS.[149] The Ruby language community exhibits a similarly strong collective identity. Its leader, Yukihiro Matsumoto, who goes by "Matz" online, praises the community for its cohesion: "It's surprising that people in the Ruby community, having such a wide range of variety and given its size, still continue to be nice to each other. I believe the community is Ruby's greatest strength." Matz also sees the community surrounding Ruby as essential to the popularity of the language.[150] Open-source programming projects provide a particularly resonant example of discourse communities for programming. Through them, we can see how the writing of code can be socially as well as technically influenced.

Coding Literacy Is "Leaky" Knowledge

Literacy learned in one context can spill over into others as it becomes part of an individual's identity and repertoire of practices. Deborah Brandt describes literacy as "leaky" knowledge; literacy cannot be reckoned like other business assets because corporations cannot own the knowledge housed in their personnel. Corporations can, however, attempt to cultivate and control the knowledge circulating within them.[151] They can do this by regulating writing practices and by "embed[ding] knowledge deeply within organizational routines and structures so that it does not belong to any one person."[152] Conversely, the knowledge employees gain at their jobs is also

portable.[153] Brandt tells the stories of people who have learned accounting or letter writing in their workplaces and have then transferred their skills into their personal and family lives as exemplars of literacy leveraging. The Americans she studies use the literacy skills they learn in their jobs to enrich their home lives through better communication and organization.[154] Similarly, Sylvia Scribner describes ways the Vai people of Liberia transfer their literacy knowledge into writing for private communications, documenting dreams, charting family history, composing tales for children, and for keeping track of feasts and crops.[155] When people have literacy, they use it in all sorts of ways.

Like other forms of knowledge, coding literacy travels with people and spills into new contexts. As we saw in the examples of the Mozilla and FreeBSD projects, open-source software communities use committer privileges and boundary-policing of house style and values to regulate coding practices and control the product. Yet open-source projects have an incentive to encourage leaky knowledge among their participants: as noted above, a significant motivator for F/OSS participation is that skills learned in F/OSS projects can be leveraged elsewhere. The decentralization tactic of FreeBSD—Mark's assertion that no one really is in charge—can make room for what Brandt describes as "self-sponsored" learning projects. Self-sponsored projects may benefit from mentorship but they are initiated by learners and can allow learners to respond to a market putting ever-increasing demands on their literacy skills.[156]

Self-sponsored learning for coding often leans on contemporary social structures like Web forums for mentorship. On the popular Web forum Slashdot, advice about self-sponsorship was given to an Indian computer science student who wrote in looking to join an F/OSS project: "Could you suggest a road map, links to essential tools or a few projects, for people like me, who would want to improve their skills by contributing FOSS?"[157] Respondents offered him tips on how to augment his university education by pointing to particular F/OSS projects and by telling their own stories of mentorship from their project's community. One Firefox developer wrote of his first experience wanting to "patch" (contribute to) the codebase:

> I got on IRC [Internet Relay Chat] and asked for help, and a couple very patient developers helped me understand where the code was that needed to be patched, and how to fix the issue. As I found other things that were missing, or things I didn't like, I wrote more and more patches, each time with less help - probably 99% of the lines of code in my early patches were written over IRC by more experienced devs, and pasted into a text editor by me:-).[158]

The social practices and information-sharing described here enable code-learning in these spaces, but they also set the stage for coders' ability to leverage their programming skills in other contexts such as workplaces and school. Lakhani and Wolf write, "Delayed benefits to participation [in F/OSS] include career advancement … and improving programming skills (human capital)."[159] In a sector where companies frequently fold and employment is volatile, the social and technical knowledge and skills gained through an open-source project acts as a kind of professional insurance.

In traditional workplaces, there is sometimes a conflict of interest between programmers and management over how much energy should be spent on investing in new coding skills. Brandt describes the tension that can arise when corporations seeking to control the leaky knowledge limit the opportunities for employees seeking to improve their own literacy skills through self-sponsored projects—some of them on company time. We can see the same tension with coding literacy. Jason, a game designer who has worked at several major computer game companies, said that in order to stay responsive to the ever-increasing demands on their skills, he and other programmers often invented self-improvement projects that they then pitched to management as necessary features of the software. Jason noted that this process becomes so naturalized in a fast-moving field like game programming that some programmers began to believe their self-improvement projects were really in the company's interest. He speculated that a programmer who did not follow this pattern of skill development would be perpetually stuck in the same job and unable to leverage his skills in the programming marketplace.[160] New dynamics of work, some of which I described in chapter 1, necessitate these literacy self-improvement projects. In the workplace, programmers trade on their literacy skills and must be ready to revise and augment these skills for new workplaces and contexts.

The "leakiness" we see for literacy is true for any kind of knowledge investment that a company makes in its workers or that an employee makes in herself. But literacy knowledge, because of its centrality in communication and commerce and because it can travel across information networks, can have greater value in a global marketplace than other knowledge that is confined to local materials, such as knowledge of specific company policies. The incentives to control literacy—for both individuals and groups—are high. Literacy's leaky tendencies make it possible for individuals to wrest some control over their own career trajectories despite the pressure often exerted by companies.[161] The availability of online self-sponsored learning

opportunities for coding literacy makes this personal control possible. But how these self-sponsored opportunities are designed, how people learn to pitch their own literacy-improving projects to management, and how welcoming online communities are to those learning the terms of discourse are all social properties of coding literacy. The examples in this section illustrate some of the complicated ways in which literacy inheres in individuals yet is socially shaped and traded.

Conclusion

Creative programming encompasses not only the technical skills of reading and writing code on specific hardware but also social communities such as the algorave or demoscene. The F/OSS community trades and builds knowledge about code and software in complex online communities. Human audiences for code are highlighted in these creative examples, but *all* code is embedded in human social contexts. Few programmers work alone, communicating only with their computer, and many of them program in agile environments where their work is subject to regular group review or to the instant collaborative work of pair programming. Even those few programmers who do work alone use programming languages that have been shaped by the human history of programming and devices embedded in social histories. Programming—like writing—is a complex, social, expressive activity within a symbolic and technological system. In this chapter, theories of language have helped me to explain some of the ways code functions and what its affordances allow code writers to do. Affordances of both programming and writing include their creative form, discreteness, and scalability. These affordances structure how these modes can be used to create and represent knowledge. But these technical aspects of code interoperate with its social contexts and circulation. In part because of its attention to the intertwined social and technological factors of writing, diSessa's concept of material intelligences as well as New Literacy Studies theories provide a theoretically productive way of looking at code.

It is misleading, however, to consider programming and writing as separate technologies—in many practical cases, they are inseparable. As this chapter demonstrates, code, language, and culture intersect all the time. Since the advent of text-based programming languages, code is a form of writing as well as an enactment of procedures. Now that most of our writing is done with computational devices, it is no longer possible to fully extricate writing from programming, just as it is impossible to untangle writing from speech, or literacy from orality. Exploring the nature of

language, action, and expression through programming allows us to think about the relationship between writing and speech differently and also to consider the ways in which technologies can combine with and foster human abilities. Computational and textual literacy are not simply parallel abilities, but intersectional, part of a new and larger version of literacy. With this framework of material intelligences and literacy, we can see both programming and writing within a longer history of technology and communication. In the next two chapters, I explore that history in greater depth.