Understanding Computer Programming as a Literacy

Annette Vee University of Pittsburgh

ABSTRACT

Since the 1960s, computer scientists and enthusiasts have paralleled computer programming to literacy, arguing it is a generalizable skill that should be more widely taught and held. Launching from that premise, this article leverages historical and social findings from literacy studies to frame computer programming as "computational literacy." I argue that programming and writing have followed similar historical trajectories as material technologies and explain how they are intertwined in contemporary composition environments. A concept of "computational literacy" helps us to better understand the social, technical and cultural dynamics of programming, but it also enriches our vision of twenty-first century composition.

KEYWORDS

literacy, computers, computer programming, history, code, multimodal composition, digital media

> We compare mass ability to read and write software with mass literacy, and predict equally pervasive changes to society. Hardware is now sufficiently fast and cheap to make mass computer education possible: the next big change will happen when most computer users have the knowledge and power to create and modify software.
> - Guido van Rossum, from a 1999 DARPA grant application to support the teaching of computer programming.

In his DARPA grant application, Guido van Rossum, the designer of the Python programming language,[1] almost certainly hoped to tap into the positive cultural associations of literacy in order to secure funding for his project. While invoking "literacy" is rhetorically opportunistic because it is a trigger for funding, says Cynthia Selfe, the comparison between programming and literacy has been echoed so frequently that it is more than just rhetorical flourish. This parallel between programming and literacy began almost as soon as programmable computers were invented. Since the 1960s, computer enthusiasts have employed the concept of "literacy" to underscore the importance, flexibility, and power of writing for and with computers. Computer scientist Alan Perlis argued in 1961 that all undergraduates should be taught programming, just as they are taught writing in first year composition courses. At Dartmouth University in the 1960s, mathematicians John Kemeny and Thomas Kurtz designed the Basic programming language for students and non-specialists. Later, Kemeny wrote: "Someday computer literacy will be a condition for employment, possibly for survival, because the computer illiterate will be cut off from most sources of information" (216).

The parallel between programming and literacy has now made its way into popular commentary: Douglas Rushkoff says that learning programming gives people "access to the control panel of civilization" (1) and Marc Prensky argues "[a]s programming becomes more important, it will leave the back room and become a key skill and attribute of our top intellectual and social classes, just as reading and writing did in the past." Code.org, a non-profit started up in 2013 and supported by Mark Zuckerberg and Bill Gates, showcases on their website a litany of quotes from educators, technologists and public figures claiming that learning to code is a issue of "civil rights," the "4th literacy," and a way to "[c]ontrol your destiny, help your family, your community, and your country."

The promotion of computer programming as a type of writing appropriate for the masses is present in many more places than I have listed here.[2] But, unfortunately, when "literacy" is connected to programming, it is often in unsophisticated ways: literacy as limited to reading and writing text; literacy divorced from social or historical context; literacy as an unmitigated form of progress. Despite these anemic uses of the concept of literacy, however, I argue that these computer specialists are on to something. What does this persistent linking of programming to writing mean for literacy specialists?

Computer programming has a lot in common with textual literacy—historical trajectory, social shaping, affordances for communication, and connections to civic discourse. In this article, I argue that the refrain of "literacy" in reference to computer programming is not only apt because of these parallels, but that our definitions of literacy must shift to accommodate this new form of digital writing. Whether or not computer programming will be a mass literacy remains to be seen. But as code and computers have become central to our daily lives, programming has certainly become a powerful mode of written communication. Literacy studies may help us to better understand the social, technical and cultural dynamics of this important composition technology.

My approach to the link between programming and literacy moves beyond these brief, comparative gestures and leverages these historical and social findings from literacy studies:

- Historically, literacy became important when text became important to governance (Clanchy) and literacy became infrastructural when everyday life depended on it (Gilmore).
- Literacy is not simply the technical processes of reading and writing but is also shaped by social factors and ideologies (Street; diSessa).
- Which kinds of literate identities are available to people can shape how they learn literacy (Heath; Purcell-Gates; Banks).

These tenets of contemporary literacy studies provide useful perspectives on the ways that computer programming has become central to our communication practices over the last 60 years, and what that might mean for 21st century writing.

I first define and review claims about "computational literacy" and outline some salient features of code, including how it became a kind of writing. I then draw on precedents in the history of literacy in order to help us consider the future of literacy as intersecting with the writing of code. Next, I provide an overview of some of the social contexts of computer programming that bear a resemblance to the social contexts of literacy. Finally, I argue that literacy educators might want to help shape the values and approaches to programming because of programming's central role in our contemporary writing environments. Values espoused in computer science, while productive in professional contexts, are too narrow for a future where programming might become a generalized rather than specialized practice—a *literacy*. Given these stakes, I contend that literacy scholars must cultivate a deeper understanding of the complex relationship between textual and computational literacy.

COMPUTATIONAL LITERACY

Arguments for what we should consider a "literacy" have proliferated over the last two decades. Beyond the literacy of reading and writing text, scholars have proposed visual literacy (Kress and van Leeuwen), design literacy (Cope and Kalantzis), quantitative literacy (Wolfe), and video game literacy (Squire), among many other kinds of "literacies." Frameworks offer to help teachers manage the demand of integrating these new literacies into composition classes (Selber; Cope and Kalantzis; DeVoss, McKee and Selfe). This raft of new literacies and pedagogical approaches suggests the power of "literacy" as a descriptive term that implies urgency, but it also gestures toward the increasing complexity of contemporary information representation and communication. Fearing a dilution of literacy's explanatory power, Anne Frances Wysocki and Johndan Johnson-Eilola warned us against using the term to describe any and all systems of skills. Consequently, I want to be cautious about overusing the term *literacy* as well as piling on yet another skill considered essential to 21st century composition. So, what exactly is literacy? And why might it be useful to stretch its conceptual apparatus to describe computer programming?

Defining "computational literacy"

I define "literacy" as a human facility with a symbolic and infrastructural technology—such as a textual writing system—that can be used for creative, communicative and rhetorical purposes. Literacy enables people to represent their ideas in texts that can travel away from immediate, interpersonal contexts (to write) and also to interpret texts produced by others (to read).

The critical difference between a *literacy* and a system of technology-dependent communicative skills—what Andrea diSessa called a "material intelligence"—is in the positioning of the technologies that those skills employ: the technologies undergirding literacies are more central to life than those for material intelligences (5). While people *benefit* from material intelligences, they *need* literacies to negotiate their world. For a material intelligence to become a literacy, then, a material component or technology must first become central, or *infrastructural*, to a society's communication practices.[3] Next, the ability to interpret or compose with that technology must become central and widespread, which critically depends on the technology's ease of use (diSessa). In the case of textual literacy, the timing of these two events varies considerably with the society under discussion. In Britain, for instance, written texts became central to society centuries before the ability to read and write them did. And in more isolated societies, the ability to interpret texts may not yet be a literacy because it is not widespread or because texts are not important.

"Extending diSessa's schema, I propose that: a determination of whether or not a system of skills is a literacy depends on its societal context. One can be skilled at leveraging specific technologies to communicate, but a literacy leverages infrastructural symbolic technologies and is necessary for everyday life. Although many material intelligences are dubbed "literacies," this definition of literacy narrows its field and allows the term to retain its potency."

Extending diSessa's schema, I propose that: *a determination of whether or not a system of skills is a literacy depends on its societal context*. One can be skilled at leveraging specific technologies to communicate, but a *literacy* leverages *infrastructural* symbolic technologies and is necessary for everyday life. Although many material intelligences are dubbed "literacies," this definition of literacy narrows its field and allows the term to retain its potency.

Like textual literacy, computer programming is also a human facility with a symbolic technology—code—that allows people to represent and interpret ideas at a distance. Throughout much of the world, code is now infrastructural. Layered over and under the technology of writing, computer code now structures much of our contemporary communications, including word processing, email, the World Wide Web, social networking, digital video production, and mobile phone technology. Our

employment, health records, and citizenship status—once recorded solely in text—are catalogued in computer code databases. But while the technology of code is now infrastructural to our society, the ability to read and write it is not yet widespread. By the definition above, then, computer programming is a *material intelligence*, not yet a *literacy*. However, as code becomes more infrastructural and as more people learn to write it, computer programming is looking more and more like a literacy. As I describe in more detail below, programming is leaving the exclusive domain of computer science and becoming more central to professions like journalism, biology, design—and, through the digital humanities, even the study of literature and history.

The way that programming is flowing out of the specialized domain of computer science and into other fields suggests that it is becoming a literacy—that is, a widely held ability to compose and interpret symbolic and communicative texts in an infrastructural medium. Computer science and education scholars have used several terms to describe the form of literacy that programming might represent, including "procedural literacy" (Bogost; Mateas), "computational literacy" (diSessa), and "computational thinking" (Wing). In the field of rhetoric and composition, the terms "source literacy" (Stolley) and "proceduracy" (Vee) have also been employed. Here, I use the term "computational literacy" because it links the theoretical apparatus of literacy with the computation that is central to computer programming; however, I depart from these scholars in several key ways.

"Procedural literacy," for Ian Bogost, "entails the ability to reconfigure concepts and rules to understand processes, not just on the computer, but in general" (245). In this definition, Bogost equates literacy more with reading than writing—the "understand[ing]" of processes rather than the representation of them. He ascribes the "authoring [of] arguments through processes" to the concept of "procedural rhetoric" (29). For Bogost, understanding the procedures of digital artifacts or recombining blocks of meaning can be procedural literacy, too—not just the learning of programming (257). I differ from Bogost in my greater interest in writing as central to literacy practices, but also in my focus on social contexts and programming itself. While I agree that digital artifacts such as games can offer a window on the processes that underwrite software, I believe this turn away from programming sidesteps the powerful social and historical dynamics of composing code.

In another discussion of "procedural literacy," computer scientist Michael Mateas focuses on new media practitioners and attends to code-writing more specifically; however, his treatment of "literacy" is brief and leaves room for more exploration. Jeannette Wing focuses on her field of computer science (CS) rather than programming per se, arguing that "computational thinking … is a fundamental skill for everyone, not just for computer scientists" (33). She explicitly relates her concept of "computational thinking" to reading, writing and arithmetic. I differ from Wing in that I think CS is but one—albeit important—guide to thinking about this new form of potential literacy. As programming moves beyond CS, we must broaden the conceptual apparatus we use to understand its functions in the world.

Education scholar Andrea diSessa's model of the social, cognitive and material "pillars" that support literacy is compatible with my definition of computational literacy. His term and concept

of "computational literacy" acknowledges the computer as the material basis of the literacy, yet also breaks with the skills-based term "computer literacy" that limited educational theory in the 1980s and 1990s. Computation is, of course, the core function of the computer. But as computation becomes more deeply embedded in digital devices, programming is diversifying beyond what we might traditionally consider a "computer." This literacy will change as computing changes, just as textual literacy has changed with the affordances of new inscription technologies. Therefore, we need a concept of programming-as-literacy that abstracts it away from its current technologies. I favor diSessa's term "computational literacy" in this discussion because it points to the underlying mechanisms of the literacy of computer programming—computation—and yet also gestures beyond any specific instrument.

Influenced by Bogost, Mateas, Wing and diSessa, I define "computational literacy" as the constellation of abilities to break a complex process down into small procedures and then express—or "write"—those procedures using the technology of code that may be "read" by a non-human entity such as a computer. In order to write code, a person must be able to express a process in terms and procedures that can be evaluated by recourse to explicit rules. In order to read code, a person must be able to translate those hyper-explicit directions into a working model of what the computer is doing. My use of the term "literacy" here is strategic; by the definition of "literacy" above, this ability is still just a "material intelligence." However, "literacy" is suggestive of the role that this ability will take in the future, it evokes the frequent parallels made between programming and writing, and it opens up access to theories of literacy.

Computational literacy builds on textual literacy because it entails textual writing and reading, but it is also quite distinct from textual literacy. In programming, one must build structures out of explicitly defined components. As Wittgenstein argued, human language works differently: not through explicit definitions, but through use and exchange. This property of language-in-use facilitates literature and human communication as we know it. But it also makes language susceptible to failure: as JL Austin reminds us, a reader or listener can have too little information or may not want to be persuaded, which renders an action in speech "infelicitous." The explicitness required in programming is a source of critique from scholars in the humanities because it forces discrete definitions (e.g., Haefner). But code's discreteness also enables one to build complex and chained procedures with the confidence that the computer will interpret them as precisely as one writes them. Code can scale up and perform the same operation millions of times in a row—a perfect perlocutionary affordance that is impossible in human language. For these reasons, computational literacy is not simply a literacy *practice*—a subset of textual literacy. It is instead a (potential) literacy on its own, with a complex relationship to textual literacy.

The evolution of programming from engineering to writing

Programming has a complex relationship with writing; it *is* writing, but its connection to the

technology of code and computers[4] also distinguishes it from textual writing. At the same time, the writing system of code distinguishes computers from other infrastructural technologies, such as cars.[5] This merging of text and technology was not always the case for computers, however; the earliest mechanical and electrical computers relied on engineering rather than writing to program them. To name one example: Harvard's Mark I, completed in 1944, was programmed by switching circuits or physically plugging wires into vacuum tubes. Each new calculation required rewiring the machine, essentially making the computer a special purpose machine for each new situation. With the development in 1945 of the "stored program concept,"[6] the computer's program could be stored in memory in the same way that it stored its data. While simple in hindsight, this design was revelatory—it moved the concept of "programming" from physical engineering to symbolic representation. Programming became the manipulation of *code*, a symbolic text that was part of a writing system. In this way, computers became technologies of writing as well as engineering.

In subsequent years, control of the computer through code has continued to trend away from the materiality of the device and towards the abstraction of the processes that control it (Graham). To illustrate: each new revision of Digital Equipment Corporation's popular PDP computer in the 1960s required a new programming language because the hardware had changed, but by the 1990s, the Java programming language's "virtual machine" offered an effectively platform-independent programming environment. Over the last 60 years, many designers of programming languages have attempted to make more writer-friendly languages that increase the semantic value of 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 comments; and the development of programming environments to enhance the legibility of code. As the syntax of computer code has grown to resemble human language (especially English), the requirements for precise expression in programming have been changed—but not eliminated.

These language developments have led many to believe that programming will soon be obsolete—that is, once the computer can respond to natural human language, there will be no need to write code. As early as 1961, Peter Elias claimed that training in programming languages would soon cease because "undergraduates will face the console with such a natural keyboard and such a natural language that there will be little left, if anything, to the teaching of programming. [At this point, we] should hope that it would have disappeared from the curricula of all but a moderate group of specialists" (qtd. in Perlis 203). At first glance, Elias's claim appears to be supported by modern interfaces such as the iPad's. Thousands of apps, menus, and interfaces promise to deliver the power of programming to those who do not know how to write code. Collectively, they suggest that we can drag and drop our way to problem solving in software.

Elias's argument is perhaps the most persuasive against the idea that programming will become a literacy: computer interfaces and languages will evolve to be so sophisticated that very few people will need to know how to compose code. But, at least so far, that hasn't happened. While programming

> ***As a technology supporting a "material intelligence" becomes easier to master, that ability becomes more important to the workplace and more integrated into everyday life; it becomes more like a literacy.***

languages have continued to evolve since 1961, they still have a long way to go to be "natural language." Highly-readable languages such as Python, Ruby and Javascript still require logical thinking and attention to explicit expressions of procedures. Stripped-down interfaces and templates such as the iPad's can accommodate only limited design choices. They are built for the consumption rather than production of software. This means that the programmers and software designers (or the companies they work for) still call the shots.

Thus, the historical trajectory of programming language development I've outlined here suggests that the central importance of programming is unlikely to dwindle with the increasing sophistication of computer languages. In fact, if the history of literacy is any model—and in the next section I argue it is—then the development of more accessible programming languages might *increase* rather than decrease pressures on computational literacy. More sophisticated and more widely distributed writing technologies actually seem to have put more pressure on individual literacy, ratcheting up the level of skill needed for one to be considered "literate" (diSessa; Brandt). As a technology supporting a "material intelligence" becomes easier to master, that ability becomes more important to the workplace and more integrated into everyday life; it becomes more like a *literacy*. In this same way, as computers have become more accessible and languages easier to learn and use, programming appears to be moving *away* from the domain of specialists—contrary to Elias's hope.

HISTORIES OF TEXTUAL AND COMPUTATIONAL LITERACY

As Guido van Rossum suggested in his application for funding from DARPA (quoted above), we may be able to understand some of the ways that programming will function in society by turning to the history of literacy. In this section, we visit two key transitional periods in the history of literacy: the first is *when texts became infrastructural* to people's everyday lives, and the second is *when literacy began*. During the first transition, which we visit medieval England to observe, texts became central to people's lives because they aided developing institutions—government bureaucracy, written contract law and the enterprise of publishing—to scale up and accommodate population and information growth. In the second transition—the long nineteenth century in the West—institutions such as the postal service, written tax bills, public signage and mass education were built on the assumption that a majority of citizens could read and write. Below, I provide a broad-brush comparative history of these transitions to illustrate what we might learn about the trajectory of computational literacy.

When texts became infrastructural

We can see a similar historical pattern in the ways that the literacy technologies of text and code have spread throughout society: first emerging from central government initiatives, they expanded to other large institutions and businesses, and finally rippled out to restructure domestic life. Although texts had been present for ages, during the 11th-13th centuries in England, texts became commonplace in government, social organization, and commerce, enacting a gradual but profound change in the everyday life of average people. Writing evolved from an occasional tool into a highly useful and infrastructural practice for the communication and recording of information. As historian Brian Stock writes, during this time "people began to live texts" (4). This transition put a premium on skills that were once possessed only by scribes and clerks, and those who could read and write began to acquire a special status apart from other craftspeople. In a similar way, American government initiatives in computational technology prefigured the use of computers in business and education and domestic life.

For both text and code, the shift into societal infrastructure began with the central government's struggle to manage a sharp increase in information. In 11th century England, the Norman invaders to England struggled to control a vast and strange land and, consequently, the new ruler ordered a census to be taken—what became known as the "Domesday Book." Although the Domesday Book never became a comprehensive census, it required local authorities to produce written text in response to the crown's request, which encouraged and forced the adoption of text to record information in the provinces (Clanchy). In the same way that the Domesday Book attempted to catalogue the newly conquered English population, the late eighteenth century American census helped to recruit soldiers and tax citizens of the new United States. But just as memory could no longer catalogue early medieval England, human-implemented writing and mathematics reached their limit in late nineteenth century America. As the United States grew in population and the census grew in ambition, a new way of tabulating information was required. Herman Hollerith, a Census Office mechanical engineer and statistician, devised an analog, electronic computer anticipating the 1890 census information. Variations of the "Hollerith machine" were used until the 1950 census, which was the first to use a digital computer—the UNIVAC I (U.S. Census Bureau). Once again, the census was an impetus for a more sophisticated literacy technology.

After these and other centralized initiatives, both text and code made their way into other large information management projects. The ENIAC computer, created to produce firing tables for Americans in World War II, was finished too late to help the war effort; however, this research paved the way for computers to be taken up by large-scale industries and institutions such as airlines and universities in the 1950s and 1960s (Campbell-Kelly and Aspray). In a similar way, the English government prompted the adoption of writing in the provinces through new laws and policies. By the late 13th century, land laws had begun to favor written contracts over personal witnesses. To participate in this new documentary society, individuals needed to be able to sign their names or

use seals to indicate their acquiescence to the contracts (Clanchy). The spread of the texts from the central government to the provinces is echoed in the way that the programmers who cut their teeth on major government-funded software projects then circulated out into smaller industries, disseminating their knowledge of code writing further. Computer historian Martin Campbell-Kelly suggests that American central government projects were essentially training grounds for programmers and incubators for computer technology, both of which soon became central to areas such as banking, airline travel, and office work. We might be reminded of what historian Michael Clanchy calls the shift "from memory to written record"—in the 1960s and 1970s, the United States experienced a shift from written to *computational* record.

Not until the 1980s did computers became cheap enough for most people to become familiar with them. At this point, the tipping point with computers was reached, "so that people of ordinary skill would be able to use them and want to use them" (Campbell-Kelly and Aspray 231). Here we see the final stage in the spread of text and code—its expansion from centralized government and commerce into domestic life. Prior to the 11th century, when writing was only occasional and not powerfully central to business or legal transactions, the ability to read and write was a craft not so different from the ability to carve wood or make pottery; the concept of "literacy" did not exist because knowing how to read and write were specialized skills, or "material intelligences" in diSessa's terms. But as the technology of writing became infrastructural, that is, when it became central to institutions such as government and commerce, the ability to manipulate that technology could no longer be relegated to a specialized class. Laypeople living in England in the 13th century became familiar with the ways texts could record actions, could make promises, and could define their place in society (Clanchy). At this time, those who could not read text began to be recast as "illiterate" and power began to shift towards those who could.

> ***"Although code is everywhere, few—too few—can read or write it. What if those few became many? What if we're not short programmers, but instead people who can program?"***

In the same way that the technology of text began to impinge on everyday life in early medieval England, people's lives are now being circumscribed by code. To borrow Brian Stock's phrase, we have begun to "live code." Many centers of commercial and economic power are connected to code—the founders of Google, Microsoft and Facebook are just several of the people who have restructured our work and personal lives through their ability to program computers. In smaller but aggregately profound ways, our course management systems, mobile phone apps and productivity software shape the way we now teach, communicate, and even understand ourselves. Behind all of this software are programmers—people who do this work as a profession. Since the beginning of software, there have never been enough programmers to satisfy society's need for them. We are in a perpetual "software crisis," as computer historian Nathan Ensmenger notes, and as the recent Code.org promotional

video reminds us. Although code is everywhere, few—too few—can read or write it. What if those few became many? What if we're not short *programmers*, but instead *people who can program*?

When literacy began

As computers and code become more central to how we are constructed as citizens and to our communication, education, and commercial practices, computer programming is moving from a specialized to a generalized skill, or from a "material intelligence" to a literacy. We can see parallels to this moment in the ways that reading and writing became central to employment and citizenship in the nineteenth and twentieth centuries.

Beginning in the late eighteenth century United States, the ability to write and especially to read grew more common as a result of mass literacy campaigns, the rhetoric of building the new republic, vigorous economic activity, and personal motivation. The dramatic rise in literacy levels in the nineteenth century was tied to the increased importance of texts—newspapers that catalogued both local and global events, almanacs that offered advice to farmers, letters circulated by the post, and accounts that kept track of debts. Texts became a central conduit for culture and knowledge among certain groups in nineteenth century rural Vermont, for example, such that "reading became a necessity of life," according to historian William Gilmore. The ubiquity of text also affected democratic governance as printed ballots and changes in contract law put pressure on literacy (Stevens). These shifts in governance helped to justify the campaigns for mass schooling in the nineteenth-century United States (Soltow and Stevens) and rhetorically framed the work of mothers to pass literacy on to their children as citizens of the new republic (Gilmore 49). In the nineteenth and twentieth centuries the need for literacy accelerated: as literacy became common, it became more necessary and therefore became even more prevalent. Historian Lawrence Cremin commented on this period: "in an expanding literacy environment, literacy tends to create a demand for more literacy" (493).

In this same way, as computer code and the ability to write it becomes more prevalent, it is becoming an essential skill in professions outside of computer science. Clay Shirky describes this general pressure of programming on employment as "downsourcing," or the generalizing of this formerly specialized practice: "though all the attention is going to outsourcing, there's also a lot of downsourcing going on, the movement of programming from a job description to a more widely practiced skill." Although the need to *use* software has permeated almost all job description lists, trailing behind it is the need to program computers. Currently, scientists, economists, statisticians, media producers or journalists who know something about programming can streamline or enrich their research and production.

The pressure of computational literacy on the field of journalism merits a more detailed sketch because it illustrates some of the most interesting ways that writing is permeated with programming. Composition for online journalism—whether on blogs or traditional news organizations' websites—now involves the integration of visual, audio and programmatic elements. Alongside traditional

writing, we see interactive graphics and information displays on websites such as the *New York Times*, *OK Cupid* and *Five-Thirty-Eight.* These ubiquitous multimodal compositions are leading the way toward a code-based approach to conveying the news. The press industry, anxiously experiencing as well as reporting on their own state of affairs, has picked up on this shift in information conveyance from alphabetic text to code-based digital media. A writer for the Web magazine *Gawker* describes the "Rise of the Journalist Programmer:"

> Your typical professional blogger might juggle tasks requiring functional knowledge of HTML, Photoshop, video recording, video editing, video capture, podcasting, and CSS, all to complete tasks that used to be other people's problems, if they existed at all […] Coding is the logical next step down this road […] You don't have to look far to see how programming can grow naturally out of writing. (Tate)

In other words, the tasks that once belonged to other people's job descriptions have now been "downsourced" into the daily routines of today's typical journalist. The compositions *Gawker* lists differ in their technical requirements (e.g., HTML is "mark-up language," rather than a full programming language) but they all press on computational skills in some way. Responding to this shift in the profession, journalism schools have focused attention on training a new crop of journalists to be writers of code as well as text. For example, Columbia University recently announced a new Master of Science Program in Computer Science and Journalism that would integrate their traditional journalism program with computer programming (van Buskirk) and Northwestern's Medill School of Journalism has been offering scholarships to master's students with computer science or programming backgrounds for several years (Medill).

Several recent examples of computational literacy leveraged for civic applications also illustrate how it is bumping into writing, as well as into the traditional concerns of literacy educators. In "crisis camps" set up in major world cities after the 2010 earthquake in Haiti, teams of programmers used geographical data available from Google maps and NASA to write a Craigslist-style database that would match donations with needs and help locate missing persons (American Public Media). Launched in 2009, the organization Code for America uses the Teach for America model to embed programmers within local city governments to help streamline some of their specific bureaucratic processes ("About," *Code for America*). At the community level, Michele Simmons and Jeff Grabill present a case study of a citizen action group's website and database that reveal the dangers of PCBs in a local water supply, which demonstrates how community groups can struggle and succeed with code-based technology to get their messages out. Because of its centrality to civic rhetoric, Simmons and Grabill claimed this kind of programmatic database manipulation can no longer be relegated to technical disciplines: "writing at and through complex computer interfaces is a required literacy for citizenship in the twenty-first century" (441). Most of these civic activities do not require extensive skills in programming, but still draw on basic concepts of database construction and code-based computation, what might be considered basic computational literacy. This knowledge allows a writer to know when and where programming is best integrated, even if the writer does not

compose the program herself.

In these civic spaces, programming supports writing that can make a difference in the world. Perhaps for these reasons, justifications for teaching programming as a generalized skill are often pronounced along civic lines, rather than the moral and religious forces behind textual literacy campaigns in the nineteenth century. Bonnie Nardi argues that it is important for end users to know how to program "so that the many decisions a democratic society faces about the use of computers, including difficult issues of privacy, freedom of speech, and civil liberties, can be approached by ordinary citizens from a more knowledgeable standpoint" (3-4). In moments like the Congressional debates on anti-spam laws for email in the mid 1990s (Graham) and the proposed Stop Online Piracy Act (SOPA) of 2012, we saw what happens when United States public officials do not have the general knowledge Nardi argued for. In those cases, fundamental misunderstandings of computer programming obscured the terms of debate and nearly led to crippling or unenforceable laws.

In this burgeoning need for journalists, everyday citizens and public officials to know something about programming, we can see a layering of literacy technologies such as Deborah Brandt described in her ethnographic study of literacy practices in twentieth century America. Somewhat paradoxically, the increased importance of literacy accompanied an increased *complexity* of literacy. Brandt's interviewees saw their workplaces change to require more sophisticated written communication, extensive legal knowledge, and the ability to compose with computers. As new literacy technologies became more accessible and prevalent, they were folded into previously established communication practices, thereby ratcheting up the complexity of required literate practices. It appears that the increased ease of use of digital technologies has multiplied literacy again: programming is now in that complex workplace literacy mix.

COMPUTATIONAL LITERACY FROM THE PERSPECTIVE OF LITERACY STUDIES

The historical trajectory I outlined above suggests that literacy in the twenty-first century is an increasingly complex phenomenon that includes skills with both textual and computational technologies. Although apps and templates can help individuals and organizations pursue their interests in software without needing to know how to program, the specific information and communication requirements of businesses as well as governmental and social organizations are pushing software to be more customized. Consumer-focused services are often not flexible enough to accommodate local concerns, such as those that Simmons and Grabill describe. Additionally, leaving important decisions about software design up to the small (and relatively homogenous) population who can program disempowers those who only consume rather than produce software.

While Harvey Graff's historical findings indicate that the possession of literacy does not, independent of other factors, empower people or lift them out of lower incomes or social classes, *illiteracy* can be an impediment in a world where text and literacy is infrastructural to everyday life. In the same way, it appears that people who are not *computationally literate* must, in growing

numbers of cases, rely on others to help them navigate their professional, civic and personal lives. In computation as well as text, the illiterate person is "less the maker of his destiny than the literate person," as Edward Stevens observes about colonial New England (64). As more communication, social organization, government functions and commerce are being conducted through code—and as computational literacy becomes more infrastructural—the power balance is once again shifting toward those who are skilled in this new literacy technology.

This shifting power balance should alert socially attuned educators to the importance of integrating computational literacy practices into their writing and rhetoric courses. These courses are already overburdened by teaching the surfeit of literacies I mentioned at the beginning of this article. However, we can begin to think about how our writing classes might incorporate computational understanding and expression. A specific design for how this could work is beyond the scope of this article, but web design and programming in composition classes is a good start. To be clear, teaching some aspects of computational literacy in composition classes does not mean that English departments should be teaching computer science. Just as computer scientists often stress that programming is just one aspect of their discipline (Wing; Denning), we can think of computer science as an important but incomplete perspective on computational literacy. Below, I offer some perspectives on computational literacy as a social phenomenon *outside* of computer science and argue that literacy educators can provide valuable pedagogical perspectives on programming.

Social aspects of computational literacy

Emphasizing the social factors of literacy that intersect with and exceed its technological affordances, Brian Street writes: "literacy, of course, is more than just the 'technology' in which it is manifest. No one material feature serves to define literacy itself. It is a social process, in which particular socially constructed technologies are used within particular institutional frameworks for specific social purposes" (97). Street's "ideological model" synthesizes the technological and social aspects of literacy and reminds us of their complex interactions. This techno-social lens from literacy studies can help us understand the material affordances of code and computers as well as the ways that programming's social values, contexts and communities shape practices of computational literacy and the identities associated with those practices.

Because the computer is a technological object and because programming requires explicitness in a way that human communication generally does not, computer programming is often portrayed as asocial, or purely technological. As work in the history of technology has demonstrated, however, computers are social technologies in their design and deployment (Ensmenger; Campbell-Kelly and Aspray). Programming languages are written by people, and programmers write code not only for computers but also for other programmers. Although code is *often* written *primarily* for its function (as read by the computer) rather than its aesthetic value (as read by other programmers), the dual audiences for code introduce a tension in values surrounding its composition. The computer requires

precise expression, but human programmers need legibility and want aesthetically pleasing code. Emphasizing the aesthetic value of code for human audiences, the influential computer scientist Donald Knuth famously conceived of "literate programming," arguing "[l]iterature 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" (ix). Knuth's concept of "literate programming" is only possible because programming is done in social spaces with human audiences.

To understand some of these social influences on programming, we must disentangle them from the real and technical demands of the computer. Strictures such as how to control the program flow, how to name variables, how long functions should be, and how much code to write per line are established socially to help programmers work together, especially in very large teams, but they matter little to the computer. In other words, there are ways of organizing code that the computer understands perfectly well, but that are eschewed by certain human value systems in programming. Denigrated aspects of code are sometimes described with the affective term "code smells" (Atwood, "Code Smells"), which highlights the tension between code's human and computer audiences.[7]

We might think of the fallacy of right-or-wrong code as similar to that of literacy's mechanistic misrepresentation—that reading and writing are simply a matter of proper grammar and accurate decoding. Indeed, textual writing generally requires some adherence to standards in order to facilitate its reading. While concision, clear transitions and active verbs may constitute good style in certain contexts for writing, these values are socially shaped (Prendergast). They depend greatly on genre, audience and context, and a description of writing as *merely* adherence to standards ignores the complex social spaces in which it is produced and interpreted. Programming requires adherence to more explicit standards than textual writing, but also cannot be reduced to them or removed from its social contexts. For example, a programmer working by himself on a smallscale app need not attend to "proper" commenting, code formatting and variable naming, just as Strunk and White-defined "proper grammar" is often inappropriate for non-academic or creative contexts. In other words, value systems for code can fail when applied outside of the social contexts in which they developed.

As programming becomes more relevant to fields outside of computer science and software engineering, we can see the unfolding of this tension between the values for code written in those traditional contexts and values for code written outside of them. In the sciences, where code and algorithms have enabled researchers to process massive and complex datasets, this tension of what is "proper code" is quite marked. For example, a recent *Scientific American* story reported that code is not being released along with the rest of the methods used in scientific experiments, in part because scientists may be "embarrassed by the 'ugly' code they write for their own research" (Hsu). A discussion of the article on *Hacker News*, a popular online forum for programmers, encapsulates some of the key tensions in applying software engineering values to scientific code. As one commenter argued, the context for which code is written matters: "There's a huge difference between the disposable one-off code produced by a scientist trying to test a hypothesis, and production code produced by an engineer to serve in a commercial capacity" (jordanb, *Hacker News*). Code that might be fine for

a one-off experiment—that contains, say, some overly-long functions, duplication, or other kinds of "code smells"—might not be appropriate for commercial software that is, say, composed by a large team of programmers or maintained for decades across multiple operating systems. Although dominant values of programming may denigrate it, it could be just fine for a scientific context..

The practice of "obfuscating" code—rendering code illegible to humans while still parsable to the computer—also highlights the values that different contexts bring to bear on programming. Programmers will obfuscate code when they want to release working software but do not want people to read (and potentially copy or modify) its code. Obfuscation can also be used in playful ways, such as in the "Underhanded C Contest," where people write code that is deliberately deceptive—it appears (to humans) to perform one function, but actually (to the computer) performs another (Mateas and Montfort). Underhanded C contests have asked programmers to write code that misroutes luggage or mis-tallies votes while appearing to verify them. Another context for obfuscated code is in "weird languages," which are often meant to comment ironically on language design and implementation. Although they could technically be used to write software, these languages are intended more for play than use. For example, the aptly named Brainfuck language plays with obfuscation of code by taking away white space and using only a few symbolic characters rather than letters (Mateas and Montfort). If we think about code as written in social contexts and for other programmers, it makes sense that code can be creative, even playful, for the benefit of that human audience. Obfuscated code and weird languages suggest that the aesthetic value of programming varies with its context—just as it does for writing.

The specific forms and history of programming language technologies shape the value and uses of computational literacy, just as Street claims they do for textual literacy (96). Indeed, as Mateas and Montfort argue regarding obfuscated and weird languages, their inherent "play refutes the idea that the programmer's task is automatic, value-neutral, and disconnected from the meanings of words in the world." This connection of code to "words in the world" suggests that its attendant literacy is imbricated in the world where programmers learn and practice their craft. Put another way, computational literacy encompasses not only the technical skills of reading and writing code, but is best understood as coupled with its social contexts.

Literate identities

Because reading is an interpretive act that draws on knowledge acquired in specific social contexts, Street argues, "the acquisition of literacy is, in fact, a socialisation process rather than a technical process" (180). In Street's ideological model of literacy, someone who has acquired literacy in one context may not be functionally literate in another context because literacy cannot be extricated from its ideology. According to Shirley Brice Heath's canonical ethnographic study of literacy, who literacy-learners see using and valuing literacy can impact the way they take it up, or if they take it up. Children growing up in environments where text is absent and literacy is marginalized have few

ways to assimilate literacy into their lives, as suggested by Victoria Purcell-Gates's work with cycles of low literacy.

The impact of available identities on the development of literacy practices also appears to hold true for computational literacy. As demonstrated in the examples above about scientific and creative programming, the identities that computer science makes available for programming are too limited if it is to become computational literacy—a generalized rather than specialized skill. Problematically, historically disadvantaged groups in the domain of textual literacy are also finding themselves disadvantaged in computational literacy. For instance, in the 2011 account from the Bureau of Labor Statistics, only 20.8% of computer programmers were women.[8] Although programming was initially a female dominated field, it tipped toward male domination when it became more powerful and complex (Ensmenger), and has resisted a more general trend of increased participation rates of women evidenced in previously male-dominated fields such as law and medicine. Because programming is a potentially generalizable and powerful form of writing, who programs and who is computationally literate should be a concern of literacy educators.

Stereotypes for programmers appear to have been baked into the profession early on: Ensmenger notes that personality profiling was used in the 1960s to select for "anti-social, mathematically inclined male" programmers (79). Although it is no longer practiced explicitly, this personality profiling still influences the perception of programmers as stereotypically white, male, and socially awkward (Ensmenger). Even recent publications by professional organizations such as the ACM (Association for Computing Machinery) feature sexist images: a stylized illustration accompanying Peter Denning's 2008 article about the many facets of computer science (CS) shows five (seemingly white) males representing programming, engineering, math, etc. alongside one (seemingly white) female representing a computer *user*. High-profile sexism exhibited at tech conferences and fast-paced start-ups now appears to be compounding the problem—although countless men and women in tech have spoken out against it (Raja). The recent use of the term "brogrammer," associated with start-up culture only partially in jest, suggests a new kind of identity for programmers—as "bros," or, young, male, highly social and risk-taking fratboys (Raja). The so-called "rise of the brogrammer" suggests that programming *can* accommodate a broader set of identities, but these identities are still severely limited.

In addition to the narrow and lingering stereotypes of computer programmers, confining programming to its profession can constrain the styles and contexts of "acceptable" programming and discourage new learners. We can see this in the discussion about "ugly" scientific code above. The conflation of professional programming with the more generalized skill of programming is also evident in such recent critiques of the "learn to code" movement as Jeff Atwood's. Atwood, the co-founder of the popular online programming forum *Stack Overflow* and a prominent blogger, claimed we do not need a new crop of people who think they can code professional software —people such as New York City Mayor Bloomberg, who pledged in 2012 to participate in Codecademy's weekly learn-to-code emails: "To those who argue programming is an essential skill we should be

teaching our children, right up there with reading, writing, and arithmetic: **can you explain to me how Michael Bloomberg would be better at his day to day job of leading the largest city in the USA if he woke up one morning as a crack Java coder?"** ("Please Don't," emphasis in original)[9]. As several of Atwood's commenters pointed out, his argument presents programming as a tool only for the profession, and discounts the potential benefits of knowing aspects of programming in other professions or activities. In paradigms such as Atwood's, programming would be necessarily and problematically limited to the types of people already welcome in a professional context.

In contrast to these narrow perceptions of who should program, a concept of computational literacy teaches us that just as writing can be useful to those who are not professional writers, programming can be useful and enriching to many different groups of people. We can think about what Adam Banks calls "transformative access" to computational literacy—the access that allows people "*to both* change the interfaces of that system *and* fundamentally change the codes that determine how the system works" (45, emphasis in original). Changing the "interface" of programming might entail more widespread education of programming, perhaps even in our composition classes.

But changing "how the system works" would move beyond material access to education and into a critical examination of the ideologies embedded in that education. Programming as defined by computer science or software engineering is bound to echo the ideologies of those contexts. Peeling programming away from these ideologies reveals that the webmaster, gamemaker, tinkerer, scientist and citizen activist can also benefit from programming as a means to achieve their goals. Countless recent initiatives at colleges such as Harvey Mudd (Alvarado and Dodds), websites such as Codecademy.com and Code.org, and local organizations like Girl Develop It aim to teach programming in new contexts, as a more generalized skill than CS courses normally encourage. We might say they are all working toward "transformative access" (Banks) to programming. One major payoff of a concept of computational literacy is that it frames programming as a literacy practice with diverse applications rather than as a profession defined by a limited set of values.

CONCLUSION

Just as writing gradually worked its way into government and social infrastructures in the West, programming is moving into many of the domains previously dominated by writing. Similar patterns in these trajectories suggest that programming could eventually become the foundation of a new, computational literacy. But regardless of programming's future path, it is already a material intelligence and a powerful form of composition. Because of code's central role in governance, education, business and citizenship—because code is infrastructural—its writing practices concern literacy educators. This is the reactionary argument for paying attention to computational literacy—as Douglas

> ***"Programming is not replacing writing, but is rather interlacing with it, augmenting it."***

Rushkoff says, "program or be programmed." This is also the logic behind some of the calls to teach programming to elementary school kids: my learning the Logo programming language in the United States in the 1980s was supposed to help us beat those Russians, just as programmer training in the 1960s had successfully done.

But a concept of computational literacy offers us more than Cold War technology training strategies. It also helps us to understand the ways in which composition is changing. Programming is not replacing writing, but is rather interlacing with it, augmenting it. Programming plays a supportive role in traditional writing (including for this essay, composed on a computer with word processing software), and facilitates new forms of written communication such as tweets, texts, Facebook posts, emails and instant messages. Examples I have named above from journalism, literary work and civic applications demonstrate some of the changes this new hybrid writing technology has brought. Looking at the writing of code through a concept of computational literacy allows us to focus on the writing practices that undergird our complex, contemporary composition environments. It enables us to more critically engage with our software because it highlights the people who write it as well as the historical patterns that precede it.

Finally, understanding computer programming as computational literacy leads the way forward towards a more comprehensive and inclusive writing pedagogy. It is important to widen access to programming because of its power and diversity of applications, which means that programming cannot be relegated to the exclusive domain of computer science. It is also important to open up our concepts of writing to include programming. Together, images, sound and other modes of composition have already shifted the way we communicate and how we can express and process information. Consequently, literacy scholars have added these modes of writing to our concept of literacy, and have debated how to incorporate them in composition classrooms. But programming enables *all* forms of digital composition. We must now shift our models of literacy to account for it.[10]

NOTES

[1] Python is a popular language engineered to be broadly accessible and used often in education.

[2] For a more comprehensive list, see my annotated bibliography, available here: http://www.scribd.com/doc/96309313/Computer-Programming-and-Literacy-An-Annotated-Bibliography

[3] Here, I am drawing on Susan Leigh Star's framework, which describes infrastructure as embedded, transparent until it breaks down, has broad reach, is shaped by standards, and is difficult to change (381-2).

[4] When I write "computers," I mean the general class of machines that can perform computations, not simply mainframes, laptops, etc.

[5] Most modern cars contain computers, so, in some ways, I have artificially separated these two technologies. However, the car seems to be the most commonly invoked infrastructural technology to refute the unique importance of computers, and so it is the technology I choose to engage with here.

[6] This is sometimes referred to as "von Neumann architecture," after John von Neumann, a member of the ENIAC team at the University of Pennsylvania, and the named author of the groundbreaking "First Draft of a Report on the EDVAC." Because the origin of this design was collaborative and complicated, I do not refer to it as "von Neumann architecture" here.

[7] Another example of commonly denigrated code is the *goto* statement, famously derided and "considered harmful" by computer scientist Edsger Dijkstra because it allows programs to jump out of sequence ad hoc and therefore violates rules of clean program flow. Although few modern languages still technically support the *goto* statement's ability to circumvent formal program structure, it remains a touchstone for this clash in values.

[8] In software development and applications, 19% were women, and in web development, 38.6% were women.

[9] Atwood's invocation of Java—the language that dominates professional software contexts—is another indication of his assumptions that the "'everyone should learn programming' meme," as he calls it, is referring to professional and not casual programming contexts.

[10] I would like to thank Tim Laquintano, Kate Vieira, Deborah Brandt, Steve Carr, Don Bialostosky, Michael Bernard-Donals and the anonymous peer reviewers for helpful comments on drafts of this article.

WORKS CITED

"About." *Code for America*. Code for America Labs, Inc., n.d. Web. 30 Jun. 2012.

Alvarado, Christine, and Zachary Dodds. "Women in CS: An Evaluation of Three Promising Practices." *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, 2010. 57–61. Web. *ACM Digital Library*. 30 Jun. 2012.

American Public Media. "Devising Aid Programs on Their Laptops." *Marketplace.* 18 Jan. 2010. Web. 31 Jan. 2010.

Atwood, Jeff. "Code Smells" *Coding Horror Blog.* 18 May 2006. Web. 20 Jun. 2012.

-----. "Please Don't Learn to Code." *Coding Horror Blog.* 15 May 2012. Web. 15 May 2012.

Austin, John Langshaw. *How to Do Things with Words*. Cambridge: Harvard UP, 1962. Print.

Banks, Adam. *Race, Rhetoric, and Technology*. Mahwah: Lawrence Erlbaum, 2006. Print.

Bogost, Ian. *Persuasive Games: The Expressive Power of Videogames*. Cambridge: MIT P, 2007. Print.

Brandt, Deborah. *Literacy in American Lives*. Cambridge, UK: Cambridge UP, 2001. Print.

Bureau of Labor Statistics, United States Department of Labor. "Labor Force Statistics from the Current Population Survey." 1 Mar. 2012. Web. 26 Jun. 2012. <http://www.bls.gov/cps/cpsaat11.htm>

Campbell-Kelly, Martin. *From Airline Reservations to Sonic the Hedgehog: A History of the Software Industry*. Cambridge: MIT P, 2004. Print.

Campbell-Kelly, Martin, and William Aspray. *Computer: A History of the Information Machine*. The Sloan Technology Series. 2nd ed. Boulder: Westview P, 2004. Print.

Clanchy, Michael. *From Memory to Written Record: England 1066 - 1307*. Malden: Blackwell Publishing, 1993. Print.

Code.org. Code.org. n.d. Web. 7 April 2013. <http://www.code.org/>

Codecademy. Codecademy. n.d. Web. 7 April 2013. <http://www.codecademy.com>

Cope, Bill, and Mary Kalantzis, eds. *Multiliteracies: Literacy Learning and the Design of Social Futures*. London: Routledge, 2000. Print.

Cremin, Lawrence. *American Education: The National Experience, 1783-1876*. New York: Harper and Row, 1982. Print.

Denning, Peter J. "The Profession of IT: Voices of Computing." *Communications of the ACM* 51.8 (2008): 19-21. *ACM Digital Portal*. Web. 10 Apr. 2013.

DeVoss, Danielle Nicole, Heidi A. McKee, and Richard (Dickie) Selfe. *Technological Ecologies and Sustainability.* Logan: Computers and Composition Digital P. 2009. Web. 17 Jul. 2013.

Dijkstra, Edsger. "Go to Statement Considered Harmful." *Communications of the ACM* 11.3 (1968): 147-48. *ACM Digital Portal*. Web. 23 Apr. 2010.

diSessa, Andrea. *Changing Minds: Computers, Learning and Literacy*. Cambridge: MIT P, 2000. Print.

Ensmenger, Nathan. *The Computer Boys Take Over: Computers, Programmers, and the Politics of Technical Expertise*. Cambridge: MIT P, 2010. Print.

Gilmore, William J. *Reading Becomes a Necessity of Life: Material and Cultural Life in Rural New England, 1780-1835*. Knoxville: U of Tennessee P, 1989. Print.

Graff, Harvey. *The Literacy Myth: Cultural Integration and Social Structure in the Nineteenth Century*. 1979. New Brunswick: Transaction Publishers, 1991. Print.

Graham, Paul. *Hackers & Painters: Big Ideas from the Computer Age*. Sebastopol: O'Reilly, 2004. Print.

Hacker News. "Secret Computer Code Threatens Science (scientificamerican)" 13 Apr. 2012. Web. 17 Apr. 2012. < https://news.ycombinator.com/item?id=3844910>

Haefner, Joel. "The Politics of the Code." *Computers and Composition* 16.3 (1999): 325-39. *ScienceDirect.* Web. 19 Aug. 2008.

Heath, Shirley Brice. *Ways with Words: Language, Life and Work in Communities and Classroom*. New York: Cambridge UP, 1983. Print.

Hsu, Jeremy, and Innovation News Daily. "Secret Computer Code Threatens Science." *Scientific American*. Scientific American, Inc.,13 Apr. 2012. Web. 17 Apr. 2012.

Kemeny, John. "The Case for Computer Literacy." *Daedalus* 112.2 (1983): 211-30. *JStor.* Web. 13 Jan. 2013.

Knuth, Donald. *Literate Programming*. CSLI Lecture Notes. United States: Center for the Study of Language and Information, 1992. Print.

Kress, Gunther, and Theo van Leeuwen. *Reading Images: The Grammar of Visual Design*. 1996. 2nd ed. London: Routledge, 2006. Print.

Mateas, Michael. "Procedural Literacy: Educating the New Media Practitioner." *Future of Games, Simulations and Interactive Media in Learning Contexts.* Spec. issue of *On The Horizon* 13.1 (2005): 110-11. Print.

Mateas, Michael, and Nick Montfort. "A Box, Darkly: Obfuscation, Weird Languages, and Code Aesthetics." *Proceedings of the 6th Digital Arts and Culture Conference, IT University of Copenhagen* (2005): 144-53. *NickM.com.* Web. 1 Mar. 2009.

Medill School of Journalism. "Admissions." Northwestern U, n.d. Web. 24 Apr. 2010.

Nardi, Bonnie. *A Small Matter of Programming: Perspectives on End User Computing*. Cambridge: MIT P, 1993. Print.

Perlis, Alan. "The Computer and the University." *Computers and the World of the Future.* Ed. Martin Greenberger. Cambridge: MIT P, 1964. Print.

Prendergast, Catherine. "The Fighting Style: Reading the Unabomber's Strunk and White." *College English* 71.1 (2009): 10-28. Print.

Prensky, Marc. "Programming is the New Literacy." *Edutopia.* 2008. Web. 23 Apr. 2010. <http://www.edutopia.org/literacy-computer-programming>

Purcell-Gates, Victoria. *Other People's Words: The Cycle of Low Literacy*. 1995. First paperback ed. Cambridge: Harvard UP, 1997. Print.

Raja, Tasneem. "'Gangbang Interviews' and 'Bikini Shots': Silicon Valley's Brogrammer Problem." *Mother Jones* (26 Apr. 2012). Web. 18 Jun. 2012.

Rushkoff, Douglas. *Program or Be Programmed: Ten Commands for a Digital Age.* N.p.: OR Books, 2010. Kindle file.

Selber, Stuart A. *Multiliteracies for a Digital Age*. Carbondale: Southern Illinois UP, 2004. Print. Studies in Writing and Rhetoric Series.

Selfe, Cynthia. *Technology and Literacy in the 21st Century: The Importance of Paying Attention*. Carbondale: Southern Illinois UP, 1999. Print. Studies in Writing and Rhetoric Series.

Shirky, Clay. "Situated Software." *Clay Shirky's Writings about the Internet*. 2004. Web. 15 Jan 2010.

Simmons, W. Michelle, and Jeffrey Grabill. "Toward a Civic Rhetoric for Technologically and Scientifically Complex Places: Invention, Performance, and Participation." *College Composition and Communication* 58.3 (2007): 419-48. Print.

Soltow, Lee and Edward Stevens. *The Rise of Literacy and the Common School in the United States: A Socioeconomic Analysis to 1870*. Chicago: U of Chicago P, 1981. Print.

Squire, Kurt D. "Video Game Literacy: A Literacy of Expertise." *Handbook of Research on New Literacies*. Eds. Julie Coiro, Michele Knobel, Colin Lankshear, and Donald J. Leu. New York: Lawrence Erlbaum, 2008. 635-70. Print.

Star, Susan Leigh. "The Ethnography of Infrastructure." *American Behavioral Scientist* 43.3 (1999): 377–91. *Sage Journals*. Web. 3 Jul. 2013.

Stevens, Edward W., Jr. *Literacy, Law, and Social Order*. DeKalb: Northern Illinois UP, 1988. Print.

Stock, Brian. *The Implications of Literacy*. Princeton: Princeton UP, 1983. Print.

Stolley, Karl. "Source Literacy: A Vision of Craft." *Enculturation*. 10 Oct. 2012. Web. 11 Apr. 2013. < http://

enculturation.gmu.edu/node/5271>
Street, Brian. *Literacy in Theory and Practice*. Cambridge, UK: Cambridge UP, 1984. Print.
Tate, Ryan. "Hack to Hacker: Rise of the Journalist-Programmer." *Gawker: Valleywag.* 14 Jan. 2010. Web. 20 Jan. 2010.
U.S. Census Bureau. "History: Overview," "History: Univac I." 10 Dec. 2009. Web. 5 Jan. 2010.
van Buskirk, Eliot. "Will Columbia-Trained, Code-Savvy Journalists Bridge the Media/Tech Divide?" *Wired.* 7 Apr. 2010. Web. 20 Apr. 2010.
van Rossum, Guido. "Computer Programming for Everybody (Revised Proposal)." Corporation for National Research Initiatives, Jul 1999. Web. 23 Apr. 2010.
Vee, Annette. "Ideologies of a New Mass Literacy." Conference on College Composition and Communication Convention. The Riviera, Las Vegas. 14 Mar. 2013. Address.
Wing, Jeannette. "Computational Thinking." *Communications of the ACM* 49.3 (2006): 33-35. *ACM Digital Library.* Web. 7 Apr. 2007.
Wittgenstein, Ludwig. *Philosophical Investigations*. Trans. G.E.M. Anscombe. 2nd ed. Cambridge: Blackwell, 1997. Print.
Wolfe, Joanna. "Rhetorical Numbers: A Case for Quantitative Writing in the Composition Classroom." *College Composition and Communication* 61.3 (2010): 452-75. Print.
Wysocki, Anne Frances, and Johndan Johnson-Eilola. "Blinded by the Letter: Why Are We Using Literacy as a Metaphor for Everything Else?" *Passions, Pedagogies, and Twenty-First Century Technologies.* Ed. Gail Hawisher and Cynthia Selfe. Logan: Utah State UP, 1999. 349-68. Print.