2 Speech, Writing, Code
Three Worldviews

The Locus of Complexity

Speech, writing, code: these three major systems for creating signification interact with each other in millions of encounters every day. Each, as it has been theorized, comes with its own worldview, associated technologies, and user feedback loops. In the progression from speech to writing to code, each successor regime reinterprets the system(s) that came before, inscribing prior values into its own dynamics. Now that the information age is well advanced, we urgently need nuanced analyses of the overlaps and discontinuities of code with the legacy systems of speech and writing, so that we can understand how processes of signification change when speech and writing are coded into binary digits. Although speech and writing issuing from programmed media may still be recognizable as spoken utterances and print documents, they do not emerge unchanged by the encounter with code. Nor is the effect of code limited to individual texts. In a broader sense, our understanding of speech and writing in general is deeply influenced by the pervasive use of code (my deliberate situating of them as legacy systems above is intended as a provocation to suggest the perceptual shifts underway). This chapter will show how the worldview of computation sketched in chapter 1 manifests itself in the specific case of the digital computer. It will also indicate ways in which commonly accepted ideas about signification need to be reevaluated in the context of coding technologies. Finally, it will suggest terms for analysis that, although not absent in speech and writing, assume new importance with code and therefore lead to new theoretical emphases and foci of attention.

In drawing comparisons of code with speech and writing, one is faced with an embarrassment of riches. One thinks, for example, of the embodied views of speech explored by practitioners and theorists as diverse as Walter

Ong and Oliver Sacks; of such preeminent theorists of writing as Paul de Man and J. Hillis Miller. Out of many possibilities, I have chosen to focus on Ferdinand de Saussure's view of speech and Jacques Derrida's grammatological view of writing partly because these theorists take systemic approaches to their subjects that make clear the larger conceptual issues. Like them, I want to take a systemic approach by focusing on the conceptual system in which code is embedded, a perspective that immediately concerns programming for digital computers but also includes the metaphysical implications of the Regime of Computation. In addition, both Saussure and Derrida have been extremely influential in shaping contemporary views of speech and writing. By addressing their work in detail, I can by implication address the large number of related projects these two theorists have inspired. An additional advantage is Derrida's engagement with Saussure's theories of the speech system; Derrida's work has an intimate relation with Saussure's, which is richly documented in Derrida's writings, especially his early work. Moreover, one of Derrida's critical points is that writing exceeds speech and cannot simply be conceptualized as speech's written form. Similarly, I will argue that code exceeds both writing and speech, having characteristics that appear in neither of these legacy systems. This project, then, is not meant as a general comparison of code with structuralism and deconstruction but as a more narrowly focused inquiry that takes up specifically Saussure and Derrida.

Before turning to a systematic comparison of code with Saussure's speech system and Derrida's grammatology, I want to establish a general framework for my remarks. Derrida's remarkably supple and complex writing notwithstanding, much of his analysis derives from a characteristic of writing that would likely spring to mind if we were asked to identify the principal way in which writing differs from speech. Writing, unlike speech (before recording technologies), is not confined to the event of its making. It can be stored and transmitted, published in dozens of countries and hundreds of different editions, read immediately after its creation or a thousand years hence. In a sense it is no surprise that Derrida summarizes this difference between writing and speech by fusing "difference" with "defer," for the ability to defer indefinitely our encounter with writing leaps out as perhaps the most salient way in which it differs from speech. Derrida, of course, complicates and extends this commonsense idea by linking it with a powerful critique of the metaphysics of presence—but these complexities have their root in something most people would identify as a constitutive difference between speech and writing.

If we were to ask about the parallel characteristic that leaps to mind to distinguish code from speech and writing, an obvious contender would be the fact that code is addressed both to humans and intelligent machines. A further distinction is implied when we note that computers, although capable of performing diverse and complicated tasks, have at the base level of machine language only two symbols and a small number of logical operations with which to work. As Stephen Wolfram eloquently testifies in relation to his cellular automata, the amazing thing about them is that starting from an extremely simple base they can nevertheless produce complex patterns and behaviors.

This train of thought suggests the following question: Where does the complexity reside that makes code (or computers, or cellular automata) seem adequate to represent a complex world? In his critique of speech and the metaphysics of presence, Derrida makes clear that complexity was vested traditionally in the Logos, the originary point conceptualized as necessarily exceeding the world's complexity, since in this view the world derived from the Logos. For Derrida's grammatology, complexity is conceptually invested in the trace and by implication in the subtle analysis that detects the movement of the trace, which can never be found as a thing-in-itself. For code, complexity inheres neither in the origin nor in the operation of difference as such but in the labor of computation that again and again calculates differences to create complexity as an emergent property of computation. Humans, who have limited access to their own computational machinery (assuming that cognition includes computational elements, a proposition explored in chapter 9), create intelligent machines whose operations can be known in comprehensive detail (at least in theory). In turn, the existence of these machines, as many researchers have noted, suggests that the complexities we perceive and generate likewise emerge from a simple underlying base; these researchers hope that computers might show us, in Brian Cantwell Smith's phrase, "how a structured lump of clay can sit up and think."[1] The advantages of the computational view, for those who espouse it, is that emergence can be studied as a knowable and quantifiable phenomena, freed both from the mysteries of the Logos and the complexities of discursive explanations dense with ambiguities. One might observe, of course, that these characteristics also mark the limitations of a computational perspective.

It is not an accident that my analysis starts by inquiring about the locus of complexity, for the different strategies through which Saussurean linguistics, Derridean grammatology, and the Regime of Computation situate complexity have extensive implications for their respective worldviews. From this starting point I will develop several ways, not all of them obvious, in which code differs from speech and writing. My purpose is not to supplant these legacy systems and especially not to subordinate speech and writing to code (an important point, given Saussure's historic claim that writing must be subordinate to speech, and Derrida's insistence that, on the contrary, speech is subordinate to writing). Rather, for me the "juice" (as Rodney Brooks calls it) comes from the complex dynamics generated when code interacts with speech and writing, interactions that include their mundane daily transactions and also the larger stakes implicit in the conflict and cooperation of their worldviews.

Saussurean Signs and Material Matters

Let us begin, then, where Saussure began, with his assertion that the sign has no "natural" or inevitable relation to that which it refers; "the linguistic sign is arbitrary," he writes in *Course in General Linguistics.*[2] Partly for this reason, he excludes from consideration hieroglyphic and idiomatic writing. As he makes clear in a number of places, he regards speech as the true locus of the language system (*la langue*) and writing as merely derivative of speech. "A language and its written form constitute two separate systems of signs. The sole reason for the existence of the latter is to present the former. The object of study in linguistics is not a combination of the written word and the spoken word. The spoken word alone constitutes that object" (24–25). Objecting to the primacy Saussure accords to speech, Derrida sees in this hierarchy indications that Saussure remains bound to a metaphysics of presence and to the Logos with which speech has traditionally been associated; in an interview with Julia Kristeva, he remarks that "the concept of the sign belongs to metaphysics."[3]

Derrida marshals numerous arguments to insist that writing, far from being derivative as Saussure claims, in fact precedes speech; "the linguistic sign implies an original writing."[4] This counterintuitive idea, to which we will return, depends on his special understanding of writing as grammatology. In addition, he critiques Saussure's notion of the arbitrariness of the sign, asking if the "ultimate function" of this premise is to obscure "the rights of history, production, institutions etc., except in the form of the arbitrary and in the substance of naturalism" (*Of Grammatology*, 33). Of course he recognizes that the arbitrariness of the sign as Saussure posits it refers to the absence of a necessary connection between sign and referent, but his critique implies that Saussure's formulation tends to suppress the recognition that constraints of any kind might encumber the choice of sign. The productive role that constraints play in the Regime of Computation, functioning to eliminate possible choices until only a few remain, is conspicuously absent in Saussure's theory. Instead, meaning emerges and is stabilized by differential relations between signs. Jonathan Culler, writing his influential book on Saussure just as Saussure was becoming well known in the United States, makes explicit this implication: "The fact that the relation between signifier and signified is arbitrary means, then, that since there are no fixed universal concepts or fixed universal signifiers, the signified itself is arbitrary, and so is the signifier. . . . Both signifier and signified are purely relational or differential entities."[5]

Culler's interpretation helps explain why material constraints tend to drop out of Saussure's theory as appropriated by American poststructuralism, with a corresponding emphasis on differential relations and shifting uncertain ties to reference and, indeed, to the material conditions of production altogether (interpretations that have been contested by later commentators seeking to recuperate Saussure for various purposes).[6] Although Derrida's suggestion that Saussure had erased "the rights of history, production, institutions, etc." remains underdeveloped in his writing, the recognition that materiality imposes significant constraints becomes crucially important in code, and arguably in speech and writing as well. Why, for example, are there no words in English (and so far as I know, in any other language) that have one hundred or more syllables? Obviously, we have no such words because they would take too long to pronounce. In contrast to the erasure of materiality in Saussure, material constraints have historically been recognized as crucial in the development of computers, from John von Neumann in the 1950s agonizing over how to dissipate heat produced by vacuum tubes to present-day concerns that the limits of miniaturization are being approached with silicon-based chips. Moreover, material constraints have played a central role in favoring the shift to digital computers over analog.

To understand why digital computers have been favored, consider Transistor to Transistor Logic (TTL) chips, where the binary digit zero is represented by zero volts and the binary digit one by five volts. If a voltage fluctuation creates a signal of .5 volts, it is relatively easy to correct this voltage to zero, since .5 is much closer to zero than to five. Error control is much more complex with analog computers, where voltages vary continuously. For code, then, the assumption that the sign is arbitrary must be qualified by material constraints that limit the ranges within which signs can operate meaningfully and acquire significance. As we shall see, these qualifications are part of a larger picture that tie code more intimately to material conditions than is the case either for Saussure's speech system or Derrida's grammatological view of writing. In the worldview of code, materiality matters.

Culler was not wrong in emphasizing differential relations rather than material constraints, for it is clear that Saussure's view of the sign tended toward dematerialization. "The physical part of the [speech] circuit can be dismissed from consideration straight away," he says (13). Although he later acknowledges that "linguistic signs, although essentially psychological, are not abstractions," he sees their materiality as "realities localized in the brain" and distinguishes this mediated materiality from linguistic structure, where "there is only the sound pattern" (15). He argues that differences in pronunciation should be excluded because they "affect only the material substance of words" (18). Considering the sign to consist of signifier and signified, he insists that the signifier is not the acoustic sound itself but the "sound pattern" or "sound image," that is, an idealized version of the sound, whereas the signified is the concept associated with this image. The advantage of defining an immaterial pattern as the signifier is obvious; through this move, he dispenses with having to deal with variations in pronunciation, dialects, and so on (although he does recognize differences in inflection, a point that Johanna Drucker uses to excavate from his theory a more robust sense of the materiality of the sign).[7]

This is Saussure's way of coping with the noise of the world, whereby idealization plays a role similar to the function performed by discreteness in digital systems. It is worth reflecting on the differences between these two strategies. Rectifying voltage fluctuations could be compared to Saussure's "rectification" of actual sounds into idealized sound images. Importantly, however, rectification with code happens in the electronics rather than in the (idealized) system created by a human theorist. Thus it is a physical operation rather than a mental one, and it happens while the code is running rather than retrospectively in a theoretical model. These differences again illustrate that code is more intrinsically bound to materiality than Saussure's conception of *la langue*.

The disjunctions between Saussure's theory and the materially determined practices of code raise the question of whether it makes sense to use such legacy terms as "signifier" and "signified" with code. Many theorists concerned with electronic textuality are starting from new frameworks that do not rely on these traditional terms. In chapter 1, for example, I introduced Espen Aarseth's "textonomy," which sweeps the board clean and works from fundamental considerations to create a taxonomic scheme for analyzing ergodic literature. Similarly, a group of German researchers at the University of Siegen are working together in a project they call "Medienumbrüche" (Media Upheavals). They "regard the semiotic difference between strings and nets of signifiers as the foundation of a theory of 'net literature' which calls basic concepts such as 'author,' 'work,' and 'reader' into question."[8]

Although valuing these new theoretical frameworks as necessary interventions, I think it is important also to undertake a nuanced analysis of where code does and does not fit with traditional terms, especially for this project with its focus on intermediation. The exchanges, conflicts, and cooperations between the embedded assumptions of speech and writing in relation to code would be likely to slip unnoticed through a framework based solely on networked and programmable media, for the shift over to the new assumptions would tend to obscure the ways in which the older worldviews engage in continuing negotiations and intermediations with the new. For my purposes, then, the comparison is vital. Later I will perform the reverse operation of trying to fit the speech and writing systems into the worldview of code, and here too I expect the discontinuities to be as revealing as the continuities.

In the context of code, then, what would count as signifier and signified? Given the importance of the binary base, I suggest that the signifiers be considered as voltages—a suggestion already implicit in Friedrich Kittler's argument that ultimately everything in a digital computer reduces to changes in voltages.[9] The signifieds are then the interpretations that other layers of code give these voltages. Programming languages operating at higher levels translate this basic mechanic level of signification into commands that more closely resemble natural language. The translation from binary code into high-level languages, and from high-level languages back into binary code, must happen every time commands are compiled or interpreted, for voltages and the bit stream formed from them are all the machine can understand.[10] Thus voltages at the machine level function as signifiers for a higher level that interprets them, and these interpretations in turn become signifiers for a still higher level interfacing with them. Hence the different levels of code consist of interlocking chains of signifiers and signifieds, with signifieds on one level becoming signifiers on another. Because all these operations depend on the ability of the machine to recognize the difference between one and zero, Saussure's premise that differences between signs make signification possible fits well with computer architecture.

Derrida's *Différance* and the Clarity of Code

This continuity between computer architecture and Saussure's understanding of signification becomes a discontinuity, however, when Derrida transforms difference into *différance*, a neologism suggesting that meanings generated by differential relations are endlessly deferred. Speaking metaphorically, we may say that whereas Saussure focuses on two (or more) linguistic signs and infers a relationship connecting them, Derrida focuses on the gap as the important element, thus converting Saussure's presumption of presence into a generative force of absence: "*différance is not,* does not exist, is not a present-being (*on*) in any form. . . . It has neither existence nor essence. It derives from no category of being, whether present or absent."[11] To name this generative force, Derrida coins any number of terms in addition to *différance*: "trace," "arche-writing," "non-originary origin," and so on. Whatever the name, the important point is that the trace has no positive existence in itself and thus cannot be reified or recuperated back into a metaphysics of presence. "*Différance* is not only irreducible to any ontological or theological—ontotheological—reappropriation, but as the very opening of the space in which ontotheology—philosophy—produces its system and its history, it includes ontotheology, inscribing it and exceeding it without return" (*Margins*, 6). Always on the move, the trace resides everywhere and nowhere, functioning as the elusive and fecund force that makes possible all subsequent meaning. In this sense the trace, as the arche-writing that enables signification, precedes speech and also writing in the ordinary sense. The notoriously slippery nature of the trace has authorized the widely accepted idea, reinforced by thousands of deconstructive readings performed by those who followed in Derrida's footsteps, that meaning is always indeterminate and deferred.

Let us now consider how this claim for difference/deferral looks from the point of view of code. In the worldview of code, the generation of meaning happens in ways that scholars trained in the traditional humanities sometimes find difficult to understand and even more difficult to accept. At the level of binary code, the system can tolerate little if any ambiguity. For any physically embodied system, some noise and, therefore, possible ambiguities are always present. In the case of digital computers, noise enters the system (among other places) in the voltage trail-off errors discussed earlier, but these are rectified into unambiguous signals of one and zero before they enter the bit stream.[12] As the system builds up levels of programming languages such as compilers, interpreters, scripting languages, and so forth, they develop functionalities that permit increasingly greater ambiguities in the choices permitted or tolerated. The Microsoft Word spell checker is a good example. Given a letter string not in the program's dictionary, it looks for the closest matches and offers them as possibilities. No matter how sophisticated the program, however, all commands must be parsed as binary code to be intelligible to the machine.

In the context of digital computers, even less tenable than ambiguity is the proposition that a signifier could be meaningful without reference to a signified. In Derrida's view, Saussure's definition of the sign undercuts the metaphysics of presence in one sense and reinforces it in another. He argues that the very idea of a signified as conceptually distinct from the signifier (although for Saussure, indissoluble from it) gives credence to a transcendental signified, and to this extent reinscribes classical metaphysics. The distinction between signifier and signified, Derrida writes, "leaves open the possibility of thinking a *concept signified in and of itself*, a concept simply present for thought, independent of a relationship to language, that is of a relationship to a system of signifiers" (*Positions*, 19). At the same time, the distinction also opens the possibility that any signified, extracted from one context and embedded in another, could slide into the position of the signifier (for example, when one concept entails another). Since the idea of a transcendental signified implies that there is nothing above or beyond this originary point, the dynamic of signified-becoming-signifier threatens to undermine the absolute authority given to the transcendental signified. In this sense, Saussure's theory (as interpreted by Derrida) can be seen as working against the metaphysics of presence in which it otherwise remains complicit. This background helps to explain why, in deconstructive criticism, the focus tends to fall on the signifier rather than the signified. Indeed, I venture to guess that in contemporary critical theory, "signifier" is used thousands of times for every time "signified" appears.

In the worldview of code, it makes no sense to talk about signifiers without signifieds. Every voltage change must have a precise meaning in order to affect the behavior of the machine; without signifieds, code would have no efficacy. Similarly, it makes no sense to talk about floating signifiers (Lacan's adaptation of Derrida's sliding signifier) because every change in voltage must be given an unambiguous interpretation, or the program is likely not to function as intended.[13] Moreover, changes on one level of programming code must be exactly correlated with what is happening at all the other levels. If one tries to run a program designed for an older operating system on a newer one that no longer recognizes the code, the machine simply finds it unreadable, that is, unintelligible. For the machine, obsolete code is no longer a competent utterance.

Because it is a frequent point of confusion, I emphasize that these dynamics happen before (or after) any human interpretation of these messages. Whatever messages on screen may say or imply, they are themselves generated through a machine dynamics that has little tolerance for ambiguity, floating signifiers, or signifiers without corresponding signifieds. Although the computational worldview is similar to grammatology in not presuming the transcendental signified that Derrida detects in Saussure's speech system, it also does not tolerate the slippages Derrida sees as intrinsic to grammatology. Nor does code allow the infinite iterability and citation that Derrida associates with inscriptions, whereby any phrase, sentence, or paragraph can be lifted from one context and embedded in another. "A written sign carries with it a force that breaks with its context, that is, with the collectivity of presences organizing the moment of its inscription. This breaking force [*force de rupture*] is not an accidental predicate but the very structure of the written text."[14] Although Derrida asserts that this iterability is not limited to written language but "is to be found in all language" (*Limited Inc*, 10), this assertion does not hold true literally for code, where the contexts are precisely determined by the level and nature of the code. Code may be rendered unintelligible if transported into a different context—for example, into a different programming language or a different syntactic structure within the same language. Only at the high level of object-oriented languages such as C++ does code recuperate the advantages of citability and iterability (i.e., inheritance and polymorphism, in the discourse of programming language) and in this sense become "grammatological."[15]

Ellen Ullman, a software engineer who has been a pioneer in a field largely dominated by men, has written movingly about the different worldviews of code and natural language as they relate to ambiguity and slippage.[16] Asked in an interview with Scott Rosenberg if code is a language, she replied, "We can use English to invent poetry, to try to express things that are very hard to express. In programming you really can't. Finally, a computer program has only one meaning: what it does. It isn't a text for an academic to read. Its entire meaning is its function."[17] Emphasizing the unforgivingness of code, Ullman underscores its functionality. Code has become an important actor in the contemporary world because it has the power to change the behavior of digital computers, which in turn permeate nearly every kind of advanced technology. Code can set off missiles or regulate air traffic; control medical equipment or generate PET scans; model turbulent flow or help design innovative architecture. All of these tasks are built ultimately on a base of binary code and logic gates that are intolerant to error. Above all else, the digital computer is a logic machine, as Martin Davis shows elegantly in *The Universal Computer*, where he discusses the history of the logic on which the digital computer is based.

In *Close to the Machine*, Ullman illustrates vividly the contrast between the worldviews of code and human language when she discusses a software system she was commissioned to create that would help deliver information to AIDS patients. She recounts working in her San Francisco loft with a small group of programmers she had hired. They worked around the clock to meet the deadline, speaking in rapid-fire phrases about the structure of the program, the work-arounds they could devise, the flow charts that showed how the code would be processed. Junk food abounded, dress was disheveled, courtesy was a waste of precious minutes, and sleep became a distant memory as they subordinated all other concerns to the logic of the machine (1–17). Then, as the independent contractor responsible for the system, she met with the staff whose clients would be using the software. Suddenly the clear logic dissolved into an amorphous mass of half-articulated thoughts, messy needs and desires, fears and hopes of desperately ill people. Even as she tried to deal with the cloud of language in which these concerns were expressed, her mind raced to translate the concerns into a list of logical requirements to which her programmers could respond. Acting as the bridge arcing between the floating signifiers of natural language and the rigidity of machine code, she felt acutely the strain of trying to reconcile these two very different views of the world. "I had reduced the users' objections to a set of five system changes. I would like to use the word 'reduce' like a cook: something boiled down to its essence. But I was aware that the real human essence was already absent from the list I'd prepared. An item like 'How will we know if the clients have TB?'—the fear of sitting in a small, poorly ventilated room with someone who has medication-resistant TB, the normal and complicated biological urgency of that question—became a list of data elements to be added to the screens and database" (13–14).

One of the book's poignant scenes comes when Ullman, emotionally stressed by events in her life, decides to take her computer apart and put it back together, in a kind of somatic therapy that soothed by putting her physically in touch with the parts that functioned in such perfectly logical fashion (65–94). The scene illustrates another way in which the worldview of code differs from Saussure's dematerialized view of speech and Derrida's emphasis on linguistic indeterminacy. Although it is possible to view computer algorithms as logical structures that do not need to be instantiated to have meaning (the received view of many computer science departments), in practice any logical or formal system must run on some kind of platform to acquire meaning, whether a human brain or a digital computer. Without the ability to change the behavior of machines, code would remain a relatively esoteric interest of mathematicians working in areas such as the lambda calculus (where algorithms were used for research purposes in the 1930s, prior to the invention of the digital computer).[18] Code has become arguably as important as natural language because it causes things to happen, which requires that it be executed as commands the machine can run.

Code that runs on a machine is performative in a much stronger sense than that attributed to language. When language is said to be performative, the kinds of actions it "performs" happen in the minds of humans, as when someone says "I declare this legislative session open" or "I pronounce you husband and wife." Granted, these changes in minds can and do result in behavioral effects, but the performative force of language is nonetheless tied to the external changes through complex chains of mediation. By contrast, code running in a digital computer causes changes in machine behavior and, through networked ports and other interfaces, may initiate other changes, all implemented through transmission and execution of code. Although code originates with human writers and readers, once entered into the machine it has as its primary reader the machine itself. Before any screen display accessible to humans can be generated, the machine must first read the code and use its instructions to write messages humans can read. Regardless of what humans think of a piece of code, the machine is the final arbiter of whether the code is intelligible. If the machine cannot read the code or if the program does not work properly, then the code must be changed and corrected before the machine can make things happen. In *Protocol*, Alexander R. Galloway makes this point forcefully when he defines code as executable language. "But how can code be so different from mere writing?" he asks. "The answer to this lies in the unique nature of computer code. . . . Code is a language, but a very special kind of language. *Code is the only language that is executable*."[19]

This character of code stands in striking contrast to the communities that decide whether an act of speech or a piece of writing constitutes a legible and competent utterance. As Saussure observes, no one person can change the spoken language system. "A language, as a collective phenomenon, takes the form of a totality of imprints in everyone's brain. . . . Thus it is something which is in each individual, but is none the less common to all. At the same time it is out of reach of any deliberate interference by individuals" (*Course*, 19). It takes many individual adopters of a change and a (relatively) long time for changes in the speech system to occur. (To some extent this claim requires modification in light of mass media such as television and newspapers, where a single speaker or small group of speakers can change the system if given enough press coverage. Saussure's point remains valid, however, in the sense that only if the speech acts of such privileged speakers are widely adopted can they actually change the system.) For Derrida, writing differs from Saussure's view of the speech system because inscriptions can endure over centuries or millennia, and thus can be cited, and therefore embedded, in a potentially infinite number of different contexts.

Moreover, Derrida reads the qualities of iterability and citation back into speech, instancing precisely the phenomena (e.g., quotations and speeches performed by actors in a theatrical performance) that Austin excluded from his speech act theory because he viewed them as anomalous.[20] In Derrida's grammatology, the more or less coherent community of speakers that Saussure presumes fractures into historically and geographically diverse contexts of different writing (and speaking) practices, different communities of general or expert readers, and different criteria for what constitutes competence and legibility. As William R. Paulson, among others, has observed, Derrida's complex writing style itself performs this diversity, insofar as it creates a class of "priestly" interpreters who can understand his writing, in contrast to many nonspecialist readers who have found it unintelligible.[21]

Like esoteric theoretical writing, code is intelligible only to a specialized community of experts who understand its complexities and can read and write it with fluency. There is, however, a significant difference between the worldview of code, on one hand, and, on the other, the community of speakers Saussure presumes and the infinitely diverse inscription contexts Derrida invokes. With code, a (relatively) few experts can initiate changes in the system that are often so significant they render previous systems illegible, as when Microsoft creates a new operating system such as Windows XP, which is not backward compatible with Windows 95 and earlier versions. Moreover, in code the breaks are much sharper and more complete than with either speech or writing. Occasionally I meet people who are using hardware and software that have been obsolete for years. Although they can still produce documents using these versions, they are increasingly marooned on an island in time, unable to send readable files or to read files from anyone else. Whereas undergraduates can understand (with some help) the Middle English of Chaucer's *Canterbury Tales* or the Elizabethan English of Shakespeare, thus making a connection over the hundreds of years that separate them from these works, no such bridges can be built between Windows 95 and Windows XP (separated by a mere seven or eight years), without both massive recoding and fluency in the nastily complex code of Windows programming.

Although code may inherit little or no baggage from classical metaphysics, it is permeated throughout with the politics and economics of capitalism, along with the embedded assumptions, resistant practices, and hegemonic reinscriptions associated with them. The open source movement testifies eloquently to the centrality of capitalist dynamics in the marketplace of code, even as it works to create an intellectual commons that operates according to the very different dynamics of a gift economy.[22]

The Hierarchy of Code

As we have seen, code differs from speech and writing in that it exists in clearly differentiated versions that are executable in a process that includes hardware and software and that makes obsolete programs literally unplayable unless an emulator or archival machine is used. Moreover, the historical strata of code do not involve a troublesome metaphysical inheritance but a troublesome deep layer of assembler code that can be understood and reverse engineered only with great difficulty, as demonstrated by attempts to excavate it and correct the problems associated with the Y2K crisis. The ways in which the historical character of code influence its alterations is a subject that requires understanding the difference between change within a given slice of time (synchrony) and change across time (diachrony).

For Saussure, the proper object of study for the semiotics of the speech system is the synchronic orientation. He points out in the introduction to *Course in General Linguistics* that his analysis reorients the field of study from the historicist and philological emphasis it had in previous generations to an understanding that regards the language system as a (more or less) coherent synchronous structure (1–8).[23] For Derrida, the diachronic manifests itself as the (allegedly) inescapable influence of classical metaphysics, whereas the synchronic ceases to have the force it does for Saussure because inscriptions, unlike speech prior to the invention of recording technologies, can be transported into historically disparate contexts, and so exist as rock does: in historically stratified formations that stretch back for thousands of years. Derrida writes, "This citationality, this duplication or duplicity, this iterability of the mark is neither an accident nor an anomaly, it is that (normal/abnormal) without which a mark could not even have a function called 'normal.' What would a mark be that could not be cited? Or one whose origins would not get lost along with way?" (*Limited Inc*, 12).[24]

Along with these differences in conceptualizing the sign and delineating its operations across and through time go related differences in how signs work together to comprise a semiotic system. When Saussure argues that differential relations between signs constitute the engine that drives signification, he identifies two vectors along which these relations operate, working at multiple levels within the speech system. The syntagmatic vector points horizontally, for example, along the syntax of a sentence. By contrast, the paradigmatic vector operates vertically, for example, in the synonyms that might be used in place of a given word in an utterance. Derrida makes little use of these terms in his grammatology, focusing instead on the hierarchical relations between concepts by which a privileged term posits a stigmatized term as the "outside" to its "inside." For Derrida, perhaps the primary instance of this hierarchical relation, in the context of Saussure's theory, is Saussure's attempt to relegate writing to a purely derivative role. Derrida's deconstruction of this hierarchical arrangement is typical of his treatment of hierarchical dichotomies in general, for he shows that the privileged term must in fact contain and depend on what it tries to exclude. "The exteriority of the signifier is the exteriority of writing in general, and I shall try to show . . . that there is no linguistic sign before writing. Without that exteriority, the very idea of the sign falls into decay" (*Of Grammatology*, 14).

Like speech, coding structures make use of what might be called the syntagmatic and paradigmatic, but in inverse relation to how they operate in speech systems. As Lev Manovich observes in *The Language of New Media*, in speech or writing the syntagmatic is what appears on the page (or as patterned sound), whereas the paradigmatic (the alternative choices that could have been made) is virtually rather than actually present (229–33). In digital media using dynamic databases, this relationship is reversed. The paradigmatic alternatives are encoded into the database and in this sense actually exist, whereas the syntagmatic is dynamically generated on the fly as choices are made that determine which items in the database will be used. In this sense, the syntagmatic is virtual rather than actual. This insight opens onto further explorations of how databases and narratives interface together, especially in electronic literature and the more general question of literariness.

In *Reading Voices*, Garrett Stewart argues that literary language is literary in part because of its ability to mobilize "virtual" words—related by sound, sense, or usage to those actually on the page—that surround the printed text with a "blooming buzz" of variants enriching and extending the text's meanings. Although operating according to a different dynamic than envisioned by Saussure, the luminous fog created by these variants resembles the paradigmatic vector that Saussure theorized for the speech system. When electronic literature offers the user hypertextual choices that lead to multiple narrative pathways, the strategy of evoking "virtual" possibilities happens not only on the level of the individual word but at the narrative level where different strands, outcomes, and interpretations mutually resonate with one another. This richness is possible, of course, only because all these possibilities are stored in the computer, available to be rearranged, interpolated, followed or not. Somewhat paradoxically, then, the more data that are stored in computer memory, all of which are ordered according to specified addresses and called by executable commands, the more ambiguities are possible. Flexibility and the resulting mobilization of narrative ambiguities at a high level depend upon rigidity and precision at a low level. The lower the level, the closer the language comes to the reductive simplicity of ones and zeros, and yet it is precisely the ability to build up from this reductive base that enables high-level literariness to be achieved. In this sense, the interplay between the virtual syntagmatic sequences and the actual paradigmatic database resembles the dynamic that Wolfram, Fredkin, and Harold Morowitz envision for computer simulations, with high-level complexity emerging out of the brute simplicity of binary distinctions and a few logical relationships. Literariness, as it is manifested in the panoply of choices characteristic of hypertext literature, here converges with the Regime of Computation in using the "simple rules, complex behavior" characteristic of code to achieve complexity.

Along with the hierarchical nature of code goes a dynamic of concealing and revealing that operates in ways that have no parallels in speech and writing. Because computer languages become more English-like as they move higher in the "tower of languages" (Rita Raley's phrase),[25] concealing the "brute" lower levels carries considerable advantage. Knowing how to conceal code with which one is not immediately concerned is an essential practice in computer programming. One of the advantages of object-oriented languages is bundling code within an object, so that the object becomes a more or less autonomous unit that can be changed without affecting other objects in that domain. At the same time, revealing code when it is appropriate or desired also bestows significant advantage. The "reveal code" command in HTML documents, for example, allows users to see comments, formatting instructions, and other material that may illuminate the construction and intent of the work under study, a point Loss Pequeño Glazier makes effectively in *Dig[iT]al Poet(I)(c)s.* With programs such as Dreamweaver that make layers easy to construct, additional dynamics of concealing and revealing come into play through rollovers and the like, re-creating on the screen dynamics that both depend on and reflect the "tower of languages" essential to code.[26]

These practices of concealing and revealing offer fertile ground for aesthetic and artistic exploration. Layers that reveal themselves according to timed sequences, cursor movements, and other criteria have become an important technique for writers seeking to create richly dense works with multiple pathways for interaction. One such work is Talan Memmott's *Lexia to Perplexia*, a notoriously "nervous" digital production that responds to even minute cursor movements in ways a user typically does not expect and finds difficult to control. Layering in this work is arguably the principal way by which complex screen design, text, animation, and movement interact with one another. Another example is M. D. Coverley's *The Book of Going Forth* *by Day*, where the visual tropes of revealing and concealing resonate with the multiple personae, patterned after ancient Egyptian beliefs, that cohabit in one body. The layers are instrumental in creating a visual/verbal/sonic narrative in which the deep past and the present, modern skepticism and ancient rituals, hieroglyphs and electronic writing merge and blend with one another.

The "reveal code" dynamic helps to create expectations (conscious and preconscious) in which the layered hierarchical structure of the tower of languages reinforces and is reinforced by the worldview of computation. The more the concealing/revealing dynamic becomes an everyday part of life and a ubiquitous strategy in everything from commercial Web pages to digital artworks, the more plausible it makes the view that the universe generates reality though a similar hierarchical structure of correlated levels ceaselessly and forever processing code. Similarly, the more the worldview of code is accepted, the more "natural" the layered dynamics of revealing and concealing code seem. Since these dynamics do not exist in anything like the same way with speech and writing, the overall effect—no doubt subtle at first but growing in importance as digital cultures and technologies become increasingly pervasive and indispensable—is to validate code as the lingua franca of nature. Speech and writing then appear as evolutionary stepping stones necessary to ratchet up Homo sapiens to the point where humans can understand the computational nature of reality and use its principles to create technologies that simulate the simulations running on the Universal Computer. This, in effect, is the worldview Morowitz envisions when he writes about the fourth stage (after the evolution of the cosmos, life, and mind) as mind reflecting on mind.[27] The more "natural" code comes to seem, the more plausible it is to conceptualize human thought as emerging from a machinic base of computational processes, a proposition explored in chapter 7.

As I argue throughout, however, human cognition, although it may have computational elements, includes analog consciousness that cannot be understood simply or even primarily as digital computation. Speech and writing, in my view, should not be seen as predecessors to code that will wither away but as vital partners on many levels of scale in the evolution of complexity. As I said in chapter 1, not Wolfram *or* his cellular automata alone, but *both* together—hence my emphasis on narrative and subjectivity as these are intermediated with computation. It is not the triumph of the Regime of Computation that can best explain the complexities of the world and, especially, of human cultures but its interactions with the stories we tell and the media technologies instrumental in making, storing, and transmitting.

Making Discrete and the Interpenetration of Code and Language

Let us now shift from interpreting code through the worldviews of speech and writing to the inverse approach of interpreting speech and writing through the worldview of code. An operation scarcely mentioned by Saussure and Derrida but central to code is digitization, which I interpret here as the act of making something discrete rather than continuous, that is, digital rather than analog. The act of making discrete extends through multiple levels of scale, from the physical process of forming bit patterns up through a complex hierarchy in which programs are written to compile other programs. Understanding the practices through which this hierarchy is constructed, as well as the empowerments and limitations the hierarchy entails, is an important step in theorizing code in relation to speech and writing.

Let me make a claim that, in the interest of space, I will assert rather than substantiate: the world as we sense it on a human scale is basically analog. Over millennia, humans have developed biological modifications and technological prostheses to impose digitization on these analog processes, from the physiological evolution needed to produce speech to sophisticated digital computers. From a continuous stream of breath, speech introduces the discreteness of phonemes; writing carries digitization farther by adding artifacts to this physiological process, developing inscription technologies that represent phonemes with alphabetic letters. At every point, analog processes interpenetrate and cooperate with these digitizations. Experienced readers, for example, perceive words not as individual letters but as patterns perceived in a single glance. The synergy between the analog and digital capitalizes on the strengths distinctive to each. As we have seen, digitization allows fine-tuned error control and depth of coding, whereas analog processes tie in with highly evolved human capabilities of pattern processing. In addition, the analog function of morphological resemblance, that is, similarity of form, is the principal and indeed (so far as I know) the only way to convey information from one instantiated entity to a differently instantiated entity.

How do practices of making discrete work in the digital computer? We have already heard about the formation of the bit stream from changing voltages channeled through logic gates, a process that utilizes morphological resemblance. From the bit pattern bytes are formed, usually with each byte composed of eight bits—seven bits to represent the ASCII code, and an empty one that can be assigned special significance. At each of these stages, the technology can embody features that were once useful but have since become obsolete. For example, the ASCII code contains a seven-bit pattern corresponding to a bell ringing on a teletype. Although teletypes are no longer in use, the bit pattern remains because retrofitting the ASCII code to delete it would require far more labor than would be justified by the benefit. To some extent, then, the technology functions like a rock strata, with the lower layers bearing the fossilized marks of technologies now extinct.

In the progression from speech to writing to code, each successor regime introduces features not present in its predecessors. In *Of Grammatology*, Derrida repeatedly refers to the space between words in alphabetic writing to demonstrate his point that writing cannot be adequately understood simply as the transcription of speech patterns (39, passim). Writing, he argues, exceeds speech and thus cannot be encapsulated within this predecessor regime; "writing is at the same time more exterior to speech, not being its 'image' or its 'symbol,' and more interior to speech, which is already in itself a writing" (46). Not coincidentally, spaces play an important role in the digitization of writing by making the separation of one word from another visually clear, thus contributing to the evolution of the codex book as it increasingly realized its potential as a medium distinct from speech. Similarly, code has characteristics that occur neither in speech nor in writing—processes that, by exceeding these legacy systems, mark a disjunction.

To explore these characteristics, let us now jump to a high level in the hierarchy of code and consider object-oriented programming languages, such as the ubiquitous C++. (I leave out of this discussion the newer languages of Java and C#, for which similar arguments could be made). C++ commands are written in ASCII and then converted into machine language, so this high-level programming language, like everything that happens in the computer, builds on a binary base. Nevertheless, C++ instantiates a profound shift of perspective from machine language and also from the procedural languages like FORTRAN and BASIC that preceded it. Whereas procedural languages conceptualize the program as a flow of modularized procedures (often diagrammed with a flowchart) that function as commands to the machine, object-oriented languages are modeled after natural languages and create a syntax using the equivalent of nouns (that is, objects) and verbs (processes in the system design).

A significant advantage to this mode of conceptualization, as Bruce Eckel explains in *Thinking in C++*, is that it allows programmers to conceptualize the solution in the same terms used to describe the problem. In procedural languages, by contrast, the problem would be stated in real-world terms (Eckel's example is "put the grommet in the bin"), whereas the solution would have to be expressed in terms of behaviors the machine could execute ("set the bit in the chip that means that the relay will close"; 43). C++ reduces the conceptual overhead by allowing both the solution and the problem to be expressed in equivalent terms, with the language's structure performing the work of translating between machine behaviors and human perceptions.

The heart of this innovation is allowing the programmer to express her understanding of the problem by defining classes, or abstract data types, that have both characteristics (data elements) and behaviors (functionalities). From a class, a set of objects instantiate the general idea in specific variations—the nouns referred to above. For example, if a class is defined as "shape," then objects in that class might be triangle, circle, square, and so on (37–38). Moreover, an object contains not only data but also functions that operate on the data—that is, it contains constraints that define it as a unit, and it also has encapsulated within it behaviors appropriate to that unit. For example, each object in "shape" might inherit the capability to be moved, to be erased, to be made different sizes, and so on, but each object would give these class characteristics its own interpretation. This method allows maximum flexibility in the initial design and in the inevitable revisions, modifications, and maintenance that large systems demand. The "verbs" then become the processes through which objects can interact with each other and the system design.

New objects can be added to a class without requiring that previous objects be changed, and new classes and metaclasses can also be added. Moreover, new objects can be created through inheritance, using a preexisting object as a base and then adding additional behaviors or characteristics. Since the way the classes are defined in effect describes the problem, the need for documentation external to the program is reduced; to a much greater extent than with procedural languages, the program serves as its own description. Another significant advantage of C++ is its ability to "hide" data and functions within an object, allowing the object to be treated as a unit without concern for these elements. "Abstraction is selective ignorance," Andrew Koenig and Barbara E. Moo write in *Accelerated C++*, a potent aphorism that speaks to the importance in large systems of hiding details until they need to be known.[28] Abstraction (defining classes), encapsulation (hiding details within objects and, on a metalevel, within classes), and inheritance (deriving new objects by building on preexisting objects) are the strategies that give object-oriented programs their superior flexibility and ease of design.

We can now see that object-oriented programs achieve their usefulness principally through the ways they anatomize the problems they are created to solve—that is, the ways in which they cut up the world. Obviously a great deal of skill and intuition goes into the selection of the appropriate classes and objects; the trick is to state the problem so it achieves abstraction in an appropriate way. This often requires multiple revisions to get it right, so ease of revision is crucial.

Some of the strategies C++ uses to achieve its language-like flexibility illustrate how it makes use of properties that do not appear in speech or writing and are specific to coding systems. Procedural languages work by what is called "early binding," a process in which the compiler (the part of the code hierarchy that translates higher-level commands into the machine language) works with the linker to direct a function call (a message calling for a particular function to be run) to the absolute address of the code to be executed. At the time of compiling, early binding thus activates a direct link between the program, compiler, and address, joining these elements before the program is actually run. C++, by contrast, uses "late binding," in which the compiler ensures that the function exists and checks its form for accuracy, but the actual address of the code is not used until the program is run.[29] Late binding is part of what allows the objects to be self-contained with minimum interference with other objects.

The point of this rather technical discussion is simple: there is no parallel to compiling in speech or writing, much less a distinction between compiling and run-time. The closest analogy, perhaps, is the translation of speech sounds or graphic letter forms into synapses in the human brain, but even to suggest this analogy risks confusing the *production* of speech and writing with its *interpretation* by a human user. Like speech and writing, computer behaviors can be interpreted by human users at multiple levels and in diverse ways, but this activity comes after (or before) the computer activity of compiling code and running programs.

Compiling (and interpreting, for which similar arguments can be made) is part of the complex web of processes, events, and interfaces that mediate between humans and machines, and its structure bespeaks the needs of both parties involved in the transaction. The importance of compiling (and interpreting) to digital technologies underscores the fact that new emphases emerge with code that, although not unknown in speech and writing, operate in ways specific to networked and programmable media. At the heart of this difference is the need to mediate between the natural languages native to human intelligence and the binary code native to intelligent machines. As a consequence, code implies a partnership between humans and intelligent machines in which the linguistic practices of each influence and interpenetrate the other.[30]

The evolution of C++ grew from precisely this kind of interpenetration. C++ is consciously modeled after natural language; once it came into wide use, it also affected how natural language is understood. We can see this two-way flow at work in the following observation by Bruce Eckel, in which he constructs the computer as an extension of the human mind. He writes, "The genesis of the computer revolution was in a machine. The genesis of our programming languages thus tends to look like that machine. But the computer is not so much a machine as it is a mind amplification tool and a different kind of expressive medium. As a result, the tools are beginning to look less like machines and more like parts of our minds, and more like other expressive mediums like writing, painting, sculpture, animation or filmmaking. Object-oriented programming is part of this movement toward the computer as an expressive medium" (*Thinking in C++*, 35). As computers are increasingly understood (and modeled after) "expressive mediums" like writing, they begin to acquire the familiar and potent capability of writing not merely to express thought but actively to constitute it. As high-level computer languages move closer to natural languages, the processes of intermediation by which each affects the other accelerate and intensify. Rita Raley has written on the relation between the spread of Global English and the interpenetration of programming languages with English syntax, grammar, and lexicon.[31] In addition, the creative writing practices of "codework," practiced by such artists as MEZ, Talan Memmott, Alan Sondheim, and others, mingle code and English in a pastiche that, by analogy with two natural languages that similarly intermingle, might be called a creole.[32]

The vectors associated with these processes do not all point in the same direction. As explored in chapter 8, (mis)recognizing visualizations of computational simulations as creatures like us both anthropomorphizes the simulations and "computationalizes" the humans. Knowing that binary code underlies complex emergent processes reinforces the view that human consciousness emerges from similar machinic processes, as explored in chapter 7. Anxieties can arise when the operations of the computer are mystified to the extent that users lose sight of (or never know) how the software actually works, thus putting themselves at the mercy of predatory companies like Microsoft, which makes it easy (or inevitable) for users to accept at face value the metaphors the corporation spoon-feeds them, a concern explored in chapter 6. These dynamics make unmistakably clear that computers are no longer merely tools (if they ever were) but are complex systems that increasingly produce the conditions, ideologies, assumptions, and practices that help to constitute what we call reality.

The operations of "making discrete" highlighted by digital computers clearly have ideological implications. Indeed, Wendy Hui Kyong Chun goes so far as to say that *software is ideology*, instancing Althusser's definition of ideology as "the representation of the subject's imaginary relationship to his or her real conditions of existence."[33] As she points out, desktop metaphors such as folders, trash cans, and so on create an imaginary relationship of the user to the actual command core of the machine, that is, to the "real conditions of existence" that in fact determine the parameters within which the user's actions can be understood as legible. As is true for other forms of ideology, the interpolation of the user into the machinic system does not require his or her conscious recognition of how he or she is being disciplined by the machine to become a certain kind of subject. As we know, interpolation is most effective when it is largely unconscious.

This conclusion makes abundantly clear why we cannot afford to ignore code or allow it to remain the exclusive concern of computer programmers and engineers. Strategies can emerge from a deep understanding of code that can be used to resist and subvert hegemonic control by megacorporations;[34] ideological critiques can explore the implications of code for cultural processes, a project already evident in Matthew Fuller's call, seconded by Matthew Kirschenbaum, for critical software studies;[35] readings of seminal literary texts can explore the implications of code for human thought and agency, among other concerns. Code is not the enemy, any more than it is the savior. Rather code is increasingly positioned as language's pervasive partner. Implicit in the juxtaposition is the intermediation of human thought and machine intelligence, with all the dangers, possibilities, liberations, and complexities this implies.