1 On Sourcery and Source Codes

The spirit speaks! I see how it must read,
And boldly write: "In the beginning was the Deed!"
—Johann Wolfgang Goethe[1]

Software emerged as a thing—as an iterable textual program—through a process of commercialization and commodification that has made code *logos*: code as source, code as true representation of action, indeed, code as conflated with, and substituting for, action.[2] Now, in the beginning, is the word, the instruction. Software as logos turns *program* into a noun—it turns process in time into process in (text) space. In other words, Manfred Broy's software "pioneers," by making software easier to visualize, not only sought to make the implicit explicit, they also created a system in which the intangible and implicit drives the explicit. They thus obfuscated the machine and the process of execution, making software the end all and be all of computation and putting in place a powerful logic of sourcery that makes source code—which tellingly was first called pseudocode—a fetish.[3]

This chapter investigates the implications of code as logos and the ways in which this simultaneous conflation and separation of instruction from execution, itself a software effect, is constantly constructed and undone, historically and theoretically. This separation is crucial to understanding the power and thrill of programming, in particular the nostalgic fantasy of an all-powerful programmer, a sovereign neoliberal subject who magically transforms words into things. It is also key to addressing the nagging doubts and frustrations experienced by programmers: the sense that we are slaves, rather than masters, clerks rather than managers—that, because "code is law," the code, rather than the programmer, rules. These anxieties have paradoxically led to the romanticization and recuperation of early female operators of the 1946 Electronic Numerical Integrator and Computer (ENIAC) as the first programmers, for they, unlike us, had intimate contact with and knowledge of the machine. They did not even need code: they engaged in what is now called "direct programming," wiring connections

and setting values. Back then, however, the "master programmer" was part of the machine (it controlled the sequence of calculation); computers, in contrast, were human. Rather than making programmers and users either masters or slaves, code as logos establishes a perpetual oscillation between the two positions: every move to empower also estranges.

This chapter, however, does not call for a return to direct programming or hardware algorithms, which, as I argue in chapter 4, also embody logos. It also does not endorse such a call because the desire for a "return" to a simpler map of power drives source code as logos. The point is not to break free from this sourcery, but rather to play with the ways in which logos also invokes "spellbinding powers of enchantment, mesmerizing fascination, and alchemical transformation."[4] The point is to make our computers more productively spectral by exploiting the unexpected possibilities of source code as fetish. As a fetish, source code produces surprisingly "deviant" pleasures that do not end where they should. Framed as a re-source, it can help us think through the machinic and human rituals that help us imagine our technologies and their executions. The point is also to understand how the surprising emergence of code as logos shifts early and still-lingering debates in new media studies over electronic writing's relation to poststructuralism, debates that the move to software studies has to some extent sought to foreclose.[5] Rather than seeing technology as simply fulfilling or killing theory, this chapter outlines how the alleged "convergence" between theory and technology challenges what we thought we knew about logos. Relatedly, engaging source code as fetish does not mean condemning software as immaterial; rather, it means realizing the extent to which software, as an "immaterial" relation become thing, is linked to changes in the nature of subject-object relations more generally. Software as thing can help us link together minute machinations and larger flows of power, but only if we respect its ability to surprise and to move.

Source Code as Logos

To exaggerate slightly, software has recently been posited as the essence of new media and knowing software a form of enlightenment. Lev Manovich, in his groundbreaking *The Language of New Media*, for instance, asserts: "New media may look like media, but this is only the surface. . . . To understand the logic of new media, we need to turn to computer science. It is there that we may expect to find the new terms, categories, and operations that characterize media that become programmable. *From media studies, we move to something that can be called 'software studies'—from media theory to software theory*."[6] This turn to software—to the logic of what lies beneath—has offered a solid ground to new media studies, allowing it, as Manovich argues, to engage presently existing technologies and to banish so-called "vapor theory"—theory that fails to distinguish between demo and product, fiction and reality—to the margins.[7]

This call to banish vapor theory, made by Geert Lovink and Alexander Galloway among others, has been crucial to the rigorous study of new media, but this rush away from what is vapory—undefined, set in motion—is also troubling because vaporiness is not accidental but rather essential to new media and, more broadly, to software. Indeed, one of this book's central arguments is that a rigorous engagement with software makes new media studies more, rather than less, vapory. Software, after all, is ephemeral, information ghostly, and new media projects that have never, or barely, materialized are among the most valorized and cited.[8] (Also, if you take the technical definition of information seriously, information increases with vapor, with entropy). This turn to computer science also threatens to reify knowing software as truth, an experience that is arguably impossible: we all know some software, some programming languages, but does anyone really "know" software? What could this knowing even mean? Regardless, from myths of all-powerful hackers who "speak the language of computers as one does a mother tongue"[9] or who produce abstractions that release the virtual[10] to perhaps more mundane claims made about the radicality of open source, knowing (or using the right) software has been made analogous to man's release from his self-incurred tutelage.[11] As advocates of free and open source software make clear, this critique aims at political, as well as epistemological, emancipation. As a form of enlightenment, it is a stance of how not to be governed like that, an assertion of an essential freedom that can only be curtailed at great cost.[12]

Knowing software, however, does not simply enable us to fight domination or rescue software from "evil-doers" such as Microsoft. Software, free or not, is embedded and participates in structures of knowledge-power. For instance, using free software does not mean escaping from power, but rather engaging it differently, for free and open source software profoundly privatizes the public domain: GNU copyleft—which allows one to use, modify, and redistribute source code and derived programs, but only if the original distribution terms are maintained—seeks to fight copyright by spreading licences everywhere.[13] More subtly, the free software movement, by linking freedom and freely accessible source code, amplifies the power of source code both politically and technically. It erases the vicissitudes of execution and the institutional and technical structures needed to ensure the coincidence of source code and its execution. This amplification of the power of source code also dominates critical analyses of code, and the valorization of software as a "driving layer" conceptually constructs software as neatly layered.

Programmers, computer scientists, and critical theorists have reduced software to a recipe, a set of instructions, substituting space/text for time/process. The current common-sense definition of *software* as a "set of instructions that direct a computer to do a specific task" and the OED definition of software as "the programs and procedures required to enable a computer to perform a specific task, as opposed to the physical components of the system" both posit software as cause, as what drives

computation. Similarly, Alexander Galloway argues, "code draws a line between what is material and what is active, in essence saying that writing (hardware) cannot *do* anything, but must be transformed into code (software) to be effective. . . . Code is a language, but a very special kind of language. *Code is the only language that is executable* . . . code is the first language that actually does what it says."[14] This view of software as "actually doing what it *says*" (emphasis added) both separates instruction from, and makes software substitute for, execution. It assumes no difference between source code and execution, between instruction and result. That is, Galloway takes the principles of executable layers (application on top of operating system, etc.) and grafts it onto the system of compilation or translation, in which higher-level languages are transformed into executable codes that are then executed line by line. By doing what it "says," code is surprisingly logos. Like the King's speech in Plato's *Phaedrus*, it does not pronounce knowledge or demonstrate it—it transparently pronounces itself.[15] The hidden signified—meaning—shines through and transforms itself into action. Like Faust's translation of logos as "deed," code is action, so that "in the beginning was the Word, and the Word was with God, and the Word was God."[16]

Not surprisingly, many scholars critically studying code have theorized code as performative. Drawing in part from Galloway, N. Katherine Hayles in *My Mother Was a Computer: Digital Subjects and Literary Texts* distinguishes between the linguistic performative and the machinic performative, arguing:

> 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 reach 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.[17]

The independence of machine action—this autonomy, or automatic executability of code—is, according to Galloway, its material essence: "The material substrate of code, which must always exist as an amalgam of electrical signals and logical operations in silicon, however large or small, demonstrates that code exists first and foremost as commands issued to a machine. Code essentially has no other reason for being than instructing some machine in how to act. One cannot say the same for the natural languages."[18] Galloway thus concludes in "Language Wants to Be Overlooked: On Software and Ideology," "to see code as subjectively performative or enunciative is to anthropomorphize it, to project it onto the rubric of psychology, rather than to understand it through its own logic of 'calculation' or 'command.'"[19]

To what extent, however, can source code be understood outside of anthropomorphization? Does understanding voltages stored in memory as commands/code not

already anthropomorphize the machine? The title of Galloway's article, "Language *Wants* to Be Overlooked" (emphasis mine), inadvertently reveals the inevitability of this anthropomorphization. How can code/language want—or most revealingly *say*—anything? How exactly does code "cause" changes in machine behavior? What mediations are necessary for this insightful yet limiting notion of code as inherently executable, as conflating meaning and action?

Crafty Sources

To make the argument that code is automatically executable, the process of execution itself not only must be erased, but source code must also be conflated with its executable version. This is possible, Galloway argues, because the two "layers" of code can be reduced to each other: "uncompiled source code is *logically* equivalent to that same code compiled into assembly language and/or linked into machine code. For example, it is absurd to claim that a certain value expressed as a hexadecimal (base 16) number is more or less fundamental than that same value expressed as binary (base 2) number. They are simply two expressions of the same value."[20] He later elaborates on this point by drawing an analogy between quadratic equations and software layers:

> One should never understand this "higher" symbolic machine as anything empirically different from the "lower" symbolic interactions of voltages through logic gates. They are complex aggregates yes, but it is foolish to think that writing an "if/then" control structure in eight lines of assembly code is any more or less machinic than doing it in one line of C, just as the same quadratic equation may swell with any number of multipliers and still remain balanced. The relationship between the two is *technical*.[21]

According to Galloway's quadratic equation analogy, the difference between a compact line of higher-level programming code and eight lines written in assembler equals the difference between two equations, in which one contains coefficients that are multiples of the other. The solution to both equations is the same: one equation is the same as the other.

This reduction, however, does not capture the difference between the various instantiations of code, let alone the empirical difference between the higher symbolic machine and the lower interactions of voltages (the question here is: where does one make the empirical observation?). To state the obvious, one cannot run source code: it must be compiled or interpreted. This compilation or interpretation—this making executable of code—is not a trivial action; the compilation of code is not the same as translating a decimal number into a binary one. Rather, it involves instruction explosion and the translation of symbolic into real addresses. Consider, for example, the instructions needed for adding two numbers in PowerPC assembly language, which is one level higher than machine language:

```
li      r3,1          *load the number 1 into register 3
li      r4,2          *load the number 2 into register 4
add     r5,r4,r3      *add r3 to r4 and store the result in r5
stw     r5,sum(rtoc)  *store the contents of r5 (i.e., 3) into the memory location
                      *called "sum" (where sum is defined elsewhere)
blr                   *end of this snippet of code[22]
```

This explosion is not equivalent to multiplying both sides of a quadratic equation by the same coefficient or to the difference between E and 15. It is, instead, a breakdown of the steps needed to perform a simple arithmetic calculation; it focuses on the movement of data within the machine. The relationship between executable and higher-level code is not that of mathematical identity but rather logical equivalence, which can involve a leap of faith. This is clearest in the use of numerical methods to turn integration—a function performed fluidly in analog computers—into a series of simpler, repetitive arithmetical steps.

This translation from source code to executable is arguably as involved as the execution of any command, and it depends on the action (human or otherwise) of compiling/interpreting and executing. Also, some programs may be executable, but not all compiled code within that program is executed; rather, lines are read in as necessary. Software is "layered" in other words, not only because source is different from object, but also because object code is embedded within an operating system.

So, to spin Galloway's argument differently, a technical relation is far more complex than a numerical one. Rhetoric was considered a *technê* in antiquity. Drawing on this Paul Ricoeur explains, "*technê* is something more refined than a routine or an empirical practice and in spite of its focus on production, it contains a speculative element."[23] A technical relation engages art or craft. A technical person is one "skilled in or practically conversant with some particular art or subject."[24] Code does not always or automatically do what it says, but it does so in a crafty, speculative manner in which meaning and action are both created. It carries with it the possibility of deviousness: our belief that compilers simply expand higher-level commands—rather than alter or insert other behaviors—is simply that, a belief, one of the many that sustain computing as such. This belief glosses over the fact that *source code only becomes a source after the fact*. Execution, and a whole series of executions, belatedly makes some piece of code a source, which is again why source code, among other things, was initially called pseudocode.

Source code is more accurately a *re-source*, rather than a source. Source code becomes the source of an action only after it—or more precisely its executable substitute—expands to include software libraries, after its executable version merges with code burned into silicon chips; and after all these signals are carefully monitored, timed,

and rectified. Source code becomes a source only through its destruction, through its simultaneous nonpresence and presence.[25] (Thus, to return to the historical difficulties of analyzing software outlined by Mahoney, every software run is to some extent a reconstruction.) Source code as *technê*, as a generalized writing, is spectral. It is neither dead repetition nor living speech; nor is it a machine that erases the difference between the two. It, rather, puts in place a "relation between life and death, between present and representation, between two apparatuses."[26] As I elaborate throughout this book, information—through its capture in memory—is undead.

Source Code, after the Fact

Early on, the difficulties of code as source were obvious. Herman H. Goldstine and John von Neumann emphasized the dynamic nature of code in their "Planning and Coding of Problems for an Electronic Computing Instrument." In it, they argued that coding, despite the name, is not simply the static translation of "a meaningful text (the instructions that govern solving the problem under consideration) from one language (the language of mathematics, in which the planner will have conceived the problem, or rather the numerical procedure by which he has decided to solve the problem) into another language (that of our code)."[27] Because code does not unfold linearly, because its value depends on intermediate results, and because code can be modified as it is run (self-modifying code), "it will not be possible in general to foresee in advance and completely the actual course of C [the sequence of codes]." Therefore, "coding is . . . the technique of providing a dynamic background to control the automatic evolution of a meaning."[28] Code as "dead repetition," in other words, has always been regenerative and interactive; every iteration alters its meaning. Even given the limits to iterability that Hayles has presciently outlined in *My Mother Was a Computer*—limits due to software as axiomatic—coding still means producing a mark, a writing, open to alteration/iteration rather than an airtight anchor.[29]

Much disciplinary effort has been required to make source code readable as the source. Structured programming, which I examine in more detail later, sought to rein in "goto crazy" programmers and self-modifying code. A response to the much-discussed "software crisis" of the late 1960s, its goal was to move programming from a craft to a standardized industrial practice by creating disciplined programmers who dealt with abstractions rather than numerical processes.[30]

Making code the source also entails reducing hardware to memory and thus erasing the existence and possibility of hardware algorithms. Code is also not always the source because hardware does not need software to "do something." One can build algorithms using hardware. Figure 1.1, for instance, is the logical statement: if notB and notA, do CMD1 (state P); if notB and notA and notZ OR B and A (state Q) then command 2.

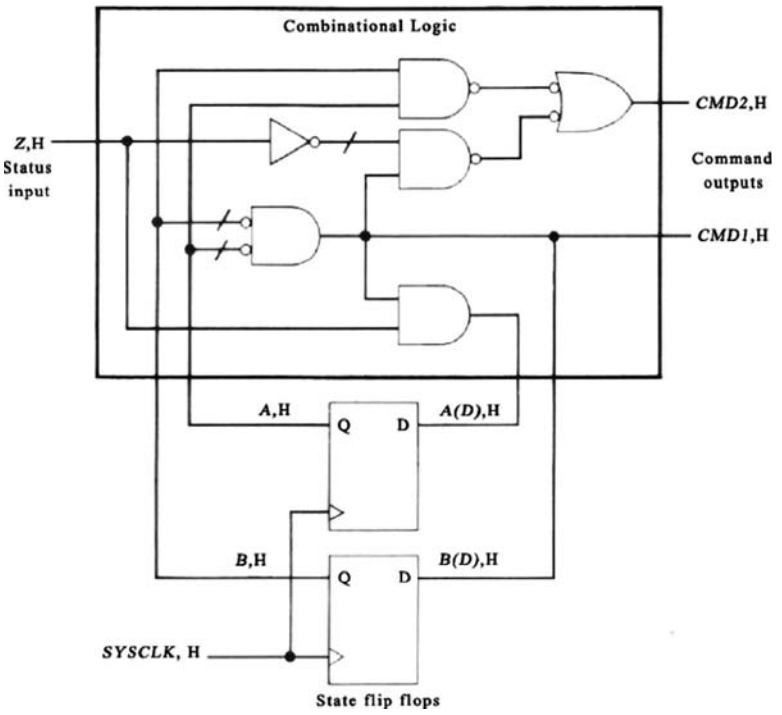

Figure 1.1
Logic diagram for a hardware algorithm

To be clear, I am not valorizing hardware over software, as though hardware naturally escapes this drive to make space signify time. Crucially, this schematic is itself an abstraction. Logic gates can only operate "logically"—as logos—if they are carefully timed. As Philip Agre has emphasized, the digital abstraction erases the fact that gates have "directionality in both space (listening to its inputs, driving its outputs) and in time (always moving toward a logically consistent relation between these inputs and outputs)."[31] When a value suddenly changes, there is a brief period in which a gate will give a false value. In addition, because signals propagate in time over space, they produce a magnetic field that can corrupt other nearby signals (known as *crosstalk*). This schematic erases all these various time- and distance-based effects by rendering space blank, empty, and banal. Thus hardware schematics, rather than escaping from the logic of sourcery, are also embedded within this structure. Indeed, as chapter 4 elaborates, John von Neumann, the generally acknowledged architect of the stored-memory digital computer, drew from Warren McCulloch and Walter Pitts's conflation of neuronal activity with its inscription in order to conceptualize modern computers. It is perhaps appropriate then that von Neumann, who died from a cancer stemming

from his work at Los Alamos, spent the last days of his life reciting from memory *Faust Part 1*.[32] At the source of stored program computing lies the Faustian erasure of word for action.

The notion of source code as source coincides with the introduction of alphanumeric languages. With them, human-written, nonexecutable code becomes source code and the compiled code, the object code. Source code thus is arguably symptomatic of human language's tendency to attribute a sovereign source to an action, a subject to a verb.[33] By converting action into language, source code emerges. Thus, Galloway's statement, "To see code as subjectively performative or enunciative is to anthropomorphize it, to project it onto the rubric of psychology, rather than to understand it through its own logic of 'calculation' or 'command,'" overlooks the fact that to use higher-level alphanumeric languages is already to anthropomorphize the machine. It is to embed computers in "logic" and to reduce all machinic actions to the commands that supposedly drive them. In other words, the fact that "code is law"—something legal scholar Lawrence Lessig emphasizes—is hardly profound.[34] After all, code is, according to the OED, "a systematic collection or digest of the laws of a country, or of those relating to a particular subject." What is surprising is the fact that software is code; that code is—has been made to be—executable, and this executability makes code not law, but rather every lawyer's dream of what law should be: automatically enabling and disabling certain actions, functioning at the level of everyday practice.[35]

Code is executable because it embodies the power of the executive, the power of enforcement that has traditionally—even within classic neoliberal logic—been the provenance of government.[36] Whereas neoliberal economist and theorist Milton Friedman must concede the necessity of government because of the difference between "the day-to-day activities of people [and] the general customary and legal framework within which these take place," code as self-enforcing law "privatizes" this function, further reducing the need for government to enforce the rules by which we play.[37] In other words, if as Foucault argues neoliberalism expands judicial interventions by reducing laws to "the rules for a game in which each remains master regarding himself and his part," then "code is law" reins in this expansion by moving enforcement from police and judicial functions to software functions.[38] "Code is law," in other words, automatically brings together disciplinary and sovereign power through the production of self-enforcing rules that, as von Neumann argues, "govern" a situation.

"Code is law" makes clear the desire for sovereign power driving both source code and performative utterances more generally. David Golumbia—looking more generally at widespread beliefs about computers—has insightfully claimed: "The computer encourages a Hobbesian conception of this political relation: one is either the person who makes and gives orders (the sovereign), or one follows orders."[39]

This conception, which crucially is also constantly undone by modern computation's twinning of empowerment with ignorance, depends, I argue, on this conflation of code with the performative. As Judith Butler has argued in *Excitable Speech*, Austinian understandings of performative utterances as simply doing what they say posit the speaker as "the judge or some other representative of the law."[40] It resuscitates fantasies of sovereign—that is *executive* (hence executable)—structures of power: it is "a wish to return to a simpler and more reassuring map of power, one in which the assumption of sovereignty remains secure."[41] This wish for a simpler map of power—indeed power as mappable—drives not only code as automatically executable, but also, as the next chapter contends, interfaces more generally. This wish is central to computers as machines that enable users/programmers to navigate neoliberal complexity.

Against this nostalgia, Butler, following Jacques Derrida, argues that iterability lies behind the effectiveness of performative utterances. For Butler, iterability is the process by which "*the subject who 'cites' the performative is temporarily produced as the belated and fictive origin of the performative itself*."[42] The programmer/user, in other words, is produced through the act of programming. Moreover, the effectiveness of performative utterances, Butler also emphasizes, is intimately tied to the community one joins and to the rituals involved—to the history of that utterance. Code as law—as a judicial process—is, in other words, far more complex than code as logos. Similarly, as Weizenbaum has argued, code understood as a judicial process undermines the control of the programmer:

> A large program is, to use an analogy of which Minsky is also fond, an intricately connected network of courts of law, that is, of subroutines, to which evidence is transmitted by other subroutines. These courts weigh (evaluate) the data given to them and then transmit their judgments to still other courts. The verdicts rendered by these courts may, indeed, often do, involve decisions about what court has "jurisdiction" over the intermediate results then being manipulated. The programmer thus cannot even know the path of decision-making within his own program, let alone what intermediate or final results it will produce. Program formulation is thus rather more like the creation of a bureaucracy than like the construction of a machine of the kind Lord Kelvin may have understood.[43]

Code as a judicial process is code as *thing*: the Latin term for thing, *res*, survives in legal discourse (and, as I explain later, literary theory). The term *res*, as Heidegger notes, designates a "gathering," any thing or relation that concerns man.[44] The relations that Weizenbaum discusses, these bureaucracies within the machine, as the rest of this chapter argues, mirror the bureaucracies and hierarchies that historically made computing possible. Importantly, this description of computers as following a set of rules that programmers must follow—Weizenbaum's insistence on the programmer's ignorance—does not undermine the resonances between neoliberalism and computation; if anything, it makes these resonances more clear. It also clarifies the desire

driving code as logos as a solution to neoliberal chaos. Foucault, emphasizing the rhetoric of the economy as a "game" in neoliberal writings, has argued, "both for the state and for individuals, the economy must be a game: a set of regulated activities . . . in which the rules are not decisions which someone takes for others. It is a set of rules which determine the way in which each must play a game whose outcome is not known by anyone."[45] Although small-s sovereigns proliferate through neoliberalism's empowered yet endangered subjects, it still fundamentally denies the position of the Sovereign who knows—a position that we nonetheless nostalgically desire . . . for ourselves.

Yes, Sir!

This conflation of instruction with result stems in part from software's and computing's gendered, military history: in the military there is supposed to be no difference between a command given and a command completed—especially to a computer that is a "girl." For computers, during World War II, were in fact young women with some background in mathematics. Not only were women available for work during that era, they also were considered to be better, more conscientious computers, presumably because they were better at repetitious, clerical tasks. They were also undifferentiated: they were all unnamed "computers," regardless of their mathematical training.[46] These computers produced ballistics tables for new weapons, tables designed to control servicemen's battlefield actions. Rather than aiming and shooting, servicemen were to set their guns to the proper values (not surprisingly, these tables and gun governors were often ignored or ditched by servicemen).[47]

The women who became the "ENIAC girls" (later the more politically correct "women of the ENIAC")—Kathleen/Kay McNulty (Mauchly Antonelli), Jean Jennings (Bartik), Frances Snyder (Holberton), Marlyn Wescoff (Meltzer), Frances Bilas (Spence), and Ruth Lichterman (Teitelbaum) (married names in parentheses)—were computers who volunteered to work on a secret project (when they learned they would be operating a machine, they had to be reassured that they had not been demoted). Programmers were former computers because they were best suited to prepare their successors: they thought and acted like computers. One could say that programming became programming and software became software when the command structure shifted from commanding a "girl" to commanding a machine. Kay Mauchly Antonelli described the "evolution" of computing as moving from female computers using Marchant machines to fill in fourteen-column sheets (which took forty hours to complete the job), to using differential analyzers (fifteen minutes to do the job), to using the ENIAC (seconds).[48]

Software languages draw from a series of imperatives that stem from World War II command and control structures. The automation of command and control, which

Paul Edwards has identified as a perversion of military traditions of "personal leadership, decentralized battlefield command, and experience-based authority,"[49] arguably started with World War II mechanical computation. Consider, for instance, the relationship between the volunteer members of the Women's Royal Naval Service (called Wrens), and their commanding officers at Bletchley Park. The Wrens also (perhaps ironically) called *slaves* by the mathematician and "founding" computer scientist Alan Turing (a term now embedded within computer systems), were clerks responsible for the mechanical operation of the cryptanalysis machines (the Bombe and then the Colossus), although at least one of the clerks, Joan Clarke (Turing's former fiancé), became an analyst. Revealingly, I. J. Good, a male analyst, describes the Colossus as enabling a man–machine synergy duplicated by modern machines only in the late 1970s: "the analyst would sit at the typewriter output and call out instructions to a Wren to make changes in the programs. Some of the other uses were eventually reduced to decision trees and were handed over to the machine operators (Wrens)."[50] This man–machine synergy, or interactive real-time (rather than batch) processing, treated Wrens and machines indistinguishably, while simultaneously relying on the Wrens' ability to respond to the mathematician's orders. This "interactive" system also seems evident in the ENIAC's operation: in figure 1.2, a male analyst issues commands to a female operator.

The story of the initial meeting between Grace Murray Hopper (one of the first and most important programmer-mathematicians) and Howard Aiken would also seem to buttress this narrative. Hopper, with a PhD in mathematics from Yale, and a former mathematics professor at Vassar, was assigned by the U.S. Navy to program the Mark 1, an electromechanical digital computer that made a sound like a roomful of knitting needles. According to Hopper, Aiken showed her "a large object with three stripes . . . waved his hand and said: 'That's a computing machine.' I said, 'Yes, Sir.' What else could I say? He said he would like to have me compute the coefficients of the arc tangent series, for Thursday. Again, what could I say? 'Yes, Sir.' I didn't know what on earth was happening, but that was my meeting with Howard Hathaway Aiken."[51] Computation depends on "Yes, Sir" in response to short declarative sentences and imperatives that are in essence commands. Contrary to Neal Stephenson, in the beginning—marking the possibility of a beginning—was the command rather than the command line.[52] The command line is a mere operating system (OS) simulation. Commands have enabled the slippage between programming and action that makes software such a compelling yet logically "trivial" communications system.[53] Commands lie at the core of the cybernetic conflation of human with machine.[54] I. J. Good's and Hopper's recollections also reveal the routinization at the core of programming: the analyst's position at Bletchley Park was soon replaced by decision trees acted on by the Wrens. Hopper, self-identified as a mathematician (not programmer), became an advocate of automatic programming. Thus routinization or automation lies at the

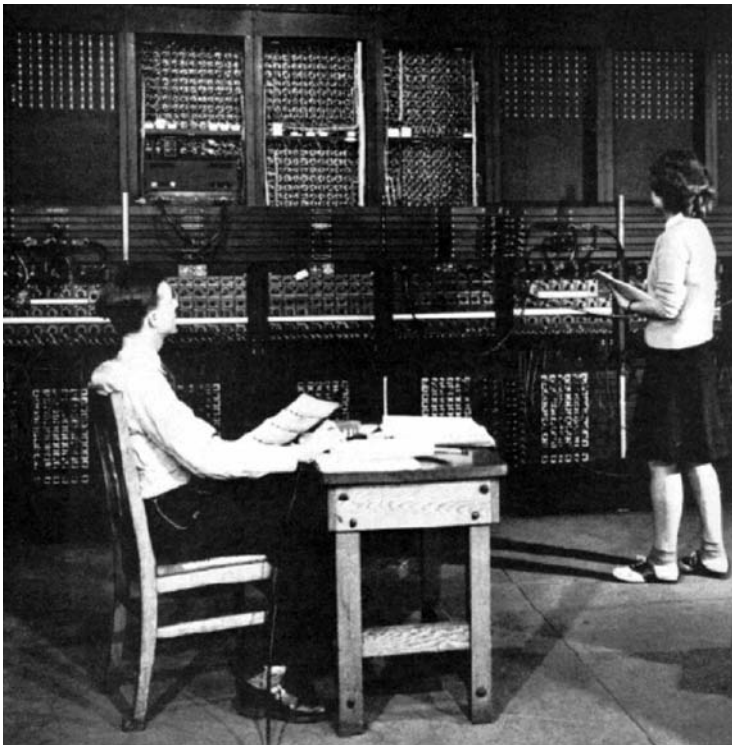

Figure 1.2
ENIAC programmers, late 1940s. U.S. military photo, Redstone Arsenal Archives, Huntsville, Alabama.

core of a profession that likes to believe it has successfully automated every profession but its own.[55]

This narrative of the interchangeability of women and software, however, is not entirely true: the perspective of the master, as Hegel famously noted, is skewed. (Tellingly, Mephistopheles offers to be Faust's servant.)[56] The master depends on the slave entirely, and it is the slave's actions that make possible another existence. Execution is never simple. Hopper's "Yes, Sir" actually did follow in the military command tradition. It was an acceptance of responsibility; she was not told how to calculate the trajectory. Also, the "women of the ENIAC," although an afterthought, played an important role in converting the ENIAC into a stored-program computer and in determining the trade-off between storing values and instructions: they did not simply operate the machine, they helped shape it and make it functional.[57] Users of the ENIAC usually were divided into pairs: one who knew the problem and one who knew the

machine "so the limitations of the machine could be fitted to the problem and the problem could be changed to fit the limitations."[58] Programming the ENIAC—that is, wiring the components together in order to solve a problem—was difficult, especially since there were no manuals or exact precedents.[59] To solve a problem, such as how to determine ballistics trajectories for new weapons, ENIAC "programmers" had first to break down the problem logically into a series of small yes/no decisions; "the amount of work that had to be done before you could ever get to a machine that was really doing any thinking," Bartik relates, was staggering and annoying.[60] The unreliability of the hardware and the fact that engineers and custodians would unexpectedly change the switches and program cables compounded the difficulty.[61]

These women, Holberton in particular, developed an intimate relation with the "master programmer," the ENIAC's control device. Although Antonelli first figured out how to repeat sections of the program, using the master programmer, Holberton, who described herself as a logician, specialized in controlling its operation.[62] As Bartik explains:

> We found it very easy to learn that you do this step, step one, then you do step two, step three, but I think the thing that was the hardest for us to learn was transfer of control which the ENIAC did have through the master programmer, so that you would be able to repeat pieces of program. So, the techniques for dividing your program into subroutines that could be repeated and things of this kind was the hardest for us to understand. I certainly know it was for me.[63]

Because logic diagrams did not then exist, Holberton developed a four-color pencil system to visualize the workings of the master programmer.[64] This drive to visualize also extended to the machine as a whole. To track the calculation, holes were drilled in the panels over the accumulators so that "when you were doing calculations these lights were flashing as the numbers built up and as you transferred numbers and things of this kind. So you had the feeling of excitement."[65] These lights not only were useful in tracking the machine, they also were invaluable for the demonstration. Even though the calculation for the demonstration was itself buggy, the flashing lights, the cards being read and written, gave the press a (to them) incomprehensible visual display of the enormity and speed of the calculation being done. In what would become a classic programming scenario, the problem was "debugged" the day after the demonstration. According to Holberton:

> I think the next morning, I woke up and in the middle of the night thinking what that error was. I came in, made a special trip on the early train that morning to look at a certain wire, and you know, it's the same kind of programming error that people make today. It's the, the decision on the terminal end of a do loop, speaking Fortran language, had the wrong value. Forgetting that zero was also one setting and the setting of the switch was one off. And I'll never forget that because there it was my first do loop error. But it went on that way and I remember telling Marlyn, I said, "If anybody asks why it's printing out that way, say it's supposed to be that way." [Laughter][66]

Programming enables a certain duplicity, as well as the possibility of endless actions that animate the machine. Holberton, described by Hopper as the best programmer she had known, would also go on to develop an influential SORT algorithm for the UNIVAC 1 (the Universal Automatic Computer 1, a commercial offshoot of the ENIAC).[67] Indeed, many of these women were hired by the Eckert–Mauchly company to become the first programmers of the UNIVAC, and were transferred to Aberdeen to train more ENIAC programmers.

Drawing from the historical importance of women and the theoretical resonances between the feminine and computing (parallels between programming and what Freud called the quintessentially feminine invention of weaving, between female sexuality as mimicry and Turing's vision of computers as universal machines/mimics) Sadie Plant has argued that computing is essentially feminine. Both software and feminine sexuality reveal the power that something that cannot be seen can have.[68] Women, Plant argues, "have not merely had a minor part to play in the emergence of digital machines. . . . Theirs is not a subsidiary role which needs to be rescued for posterity, a small supplement whose inclusion would set the existing records straight. . . . Hardware, software, wetware—before their beginnings and beyond their ends, women have been the simulators, assemblers, and programmers of the digital machines."[69] Because of this and women's early (forced) adaptation to "flexible" work conditions, Plant argues, women are best prepared to face our digital, networked future: "sperm count," she writes, "falls as the replicants stir and the meat learns how to learn for itself. Cybernetics is feminisation."[70] Responding to Plant's statement, Alexander Galloway has argued, "the universality of [computer] protocol can give feminism something that it never had at its disposal, the obliteration of the masculine from beginning to end."[71] Protocol, Galloway asserts, is inherently antipatriarchy. What, however, is the relationship between feminization and feminism, between so-called feminine modes of control and feminism? What happens if you take seriously Grace Murray Hopper's claims that the term *software* stemmed from her description of compilers as "layettes" for computers and the claim of J. Chuan Chu, one of the hardware engineers for the ENIAC, that software is the "daughter" of Frankenstein (hardware being the son)?[72]

To address these questions, we need to move beyond recognizing these women as programmers and the resonances between computers and the feminine. Such recognition alone establishes a powerful sourcery, in which programming is celebrated at the exact moment that programmers become incapable of "understanding"—of seeing through—the machine. The move to reclaim the ENIAC women as the first programmers in the mid- to late-1990s occurred when their work as operators—and the visual, intimate knowledge of machine operations this entailed—had become entirely incorporated into the machine and when women "coders" were almost definitively pushed out of the workplace. It is love at last (and first) sight, not just for these women but also for these interfaces, which really were transparent holes, in which inside and

outside coincided. Also, reclaiming these women as the first programmers and as feminist figures glosses over the hierarchies within programming—among operators, coders, and analysts—that defined the emergence of programming as a profession and as an academic discipline.[73] To put Hopper and the "ENIAC girls" together is to erase the difference between Hopper, a singular hero who always defined herself as a mathematician, and nameless disappearing computer operators. It is also to deny personal history: Hopper, a social conservative from a privileged background, stated many times that she was not a feminist, and Hopper's stances could be perceived as antifeminist (while the highest-ranking female officer in the Navy, she argued that women were incapable of serving in combat duty).[74] Not accidentally, Hopper's dream, her drive for automatic computing, was to put the programmer inside the computer and thus to rehumanize the mathematician: pseudocode was to free the mathematician and her brain from the shackles of programming.[75]

Bureaucracies within the Machine

TROPP: We talked about Von Neumann and I would like to talk about how you saw people like John Mauchly and the role that they played, and Goldstine and Burks and others that you came in contact with [including] Clippinger, and Frankel, and how, how they looked from your vantage point?
HOLBERTON: Well, we were lowly programmers, so I looked up to all these gentlemen.
TROPP: [Laughter][76]

The conflation of instruction with action, which makes computers understood as software and hardware machines such a compelling model of neoliberal governmentality and which resuscitates dreams of sovereign power, depends on incorporating historical programming hierarchies within the machine.

Programming, even at what has belatedly been recognized as its origin, was a hierarchical affair. Herman H. Goldstine and John von Neumann, in "Planning and Coding of Problems for an Electronic Computing Instrument," separated the task of planning (dealing with the dynamic nature of code through extensive flow charting) from that of coding (the microproduction of the actual instructions). Regarding dynamic or macroscopic aspects, they argued, "every mathematician, or every moderately mathematically trained person should be able to do this in a routine manner, if he has familiarized himself with the main examples that follow in this report, or if he has had some equivalent training in this method." Regarding the static or microscopic work, they asserted, "we feel certain that a moderate amount of experience with this stage of coding suffices to remove from it all difficulties, and to make it a perfectly routine operation."[77] The dropping of the pronoun *he* was not accidental: as Nathan Ensmenger and William Aspray note, the dynamic analysis was to be performed by "the 'planner,' who was typically the scientific user and overwhelmingly often was

male; the sixth task was to be carried out by 'coders'—almost always female."[78] Although this separation between operators, coders, and planners was not immediately accepted everywhere—the small Whirlwind group viewed itself more as a "model shop" in which coding, programming, and operations were mixed together—this hierarchical separation between what Philip Kraft calls the "head and the hand" became dominant as programming became a mass, commercial enterprise.[79]

SAGE (the Semi-Automatic Ground Environment) air defense system, widely considered the first large software project, was programmed by the Systems Development Corporation (SDC), an offshoot of the RAND Corporation. SDC had expanded from a few programmers to more than eight hundred by the late 1950s, making it by far the largest employer of programmers. Because its programmers went on to form the industry (it was dubbed the "university of programmers"), SAGE had a wide impact on the field's development. SAGE, however, not only taught people how to code but also inculcated a strict division of programming in which senior programmers (later systems analysts), who developed program specifications, were separated from programmers, who worked on coding specifications; they in turn were separated from the coders who turned coding specifications into documented machine code.[80] This separation, as Kraft has recorded, was still thriving in the 1970s.[81] This separation was also gendered. As Herbert D. Benington, one of the managers of SAGE, later narrated, "women turned out to be very good for the administrative programs. One reason is that these people tend to be fastidious—they worry how all the details fit together while still keeping the big picture in mind. I don't want to sound sexist, but one of our strongest groups had 80 percent women in it; they were doing the right kind of thing. The mathematicians were needed for some of the more complex applications."[82] Not accidentally, the SDC was spun off from the System Training Program, a group comprised of RAND psychologists focused on producing more effective groups.[83]

Buttressing this hierarchy was a strict system of control, "tools of a very complex nature" that did not survive SAGE. As Benington explains, these tools enabled managers to track and punish coders: "You could assign an individual a job, you could control the data that that individual had access to, you could control when that individual's program operated, and you could find out if that individual was playing the game wrong and punish that person. So we had a whole set of tools for design, for controlling of the team, for controlling of the data, and for testing the programs that were really quite advanced."[84] Because of this system of control, Benington viewed symbolic addressing and other moves to automate programming as "dangerous because they couldn't be well-disciplined." However, although automatic programming has been linked to empowerment, it has also led to the more thorough (because subtle and internalized) disciplining of programmers, which simultaneously empowers and disempowers programmers.

Indeed, this overt system of control and punishment was replaced by a "softer" system of structured programming that makes source code source. As Mahoney has argued, structured programming emerged as a "means both of quality control and of disciplining programmers, methods of cost accounting and estimation, methods of verification and validation, techniques of quality assurance."[85] Kraft targets structured programming as de-skilling: through it, programming was turned from a craft to an industrialized practice in which workers were reduced to interchangeable detail workers.[86] Structured programming limits the logical procedures coders can use and insists that the program consist of small modular units, which can be called from the main program. Structured programming (also generally known as "good programming" when I was growing up) hides, and thus secures, the machine. It focuses on and enables abstraction—and abstraction from the specific uses of and for the machine—thereby turning programming from a numerical- to a problem-based task.

Not surprisingly, having little to no contact with the actual machine enhances one's ability to think abstractly rather than numerically. Edsger Dijkstra, whose famous condemnation of "goto" statements has encapsulated to many the fundamental tenets of structured programming, believes that he was able to "pioneer" structured programming precisely because he began his programming career by coding for ghosts: for machines that did not yet exist.[87] In "Go To Statement Considered Harmful," Dijkstra argues, "the quality of programmers is a decreasing function of the density of go to statements in the programs they produce" because goto statements work against the fundamental tenet of what Dijkstra considered to be good programming, namely, the necessity to "shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible."[88] This is important because, if a program suddenly halts because of a bug, gotos (statements that tell a program to go to a specific line if a condition is met) make it difficult to find the place in the program that corresponds to the buggy code. Gotos make difficult the conflation of instruction with its product—the reduction of process to command—that grounds the emergence of software as a concrete entity and commodity. That is, gotos make it difficult for the source program to act as a legible source.[89] As this example makes clear, structured programming moves away from issues of program efficiency—the time it takes to run a program—and more toward the problem of minimizing all the costs involved in producing and maintaining large programs. This move also makes programming an "art." As Dijkstra argues in his letter justifying structured programming, "it is becoming most urgent to stop to consider programming primarily as the minimization of cost/performance ratio. We should recognize that already now programming is much more an intellectual challenge: the art of programming is the art of organizing complexity, of mastering multitude and avoiding its bastard chaos as effectively as possible."[90] Again, this depends on making "the

structure of the program text [reflect] the structure of the computation."[91] It means moving away from assembly and other languages that routinely offer bizarre exits and self-modifying code to languages that feature clear and well-documented repetitions (while . . . do . . .) that end in one clear place, that return control to the main program.

Structured programming languages "save" programmers from themselves by providing good security, where security means secure from the programmer (increasingly, "securing" the machine means making sure programmers cannot access or write over key systems).[92] Indeed, structured programming, which emphasizes programming as a problem of flow, is giving way to data abstraction, which views programming as a problem of interrelated objects, and hides far more than the machine. Data abstraction depends on information hiding, on the nonreflection of changeable facts in software. As John V. Guttag, a "pioneer" in data abstraction explains, data abstraction is all about forgetting, about hiding information about how a type is implemented behind an interface.[93] Rather than "polluting" a program by enabling invisible lines of contact between supposedly independent modules, data abstraction presents a clean or "beautiful" interface by confining specificities, and by reducing the knowledge and power of the programmer. Knowledge, Guttag insists, is dangerous: "'Drink deep, or taste not the Pierian Spring,' is not necessarily good advice. Knowing too much is no better, and often worse, than knowing too little. People cannot assimilate very much information. Any programming method or approach that assumes that people will understand a lot is highly risky."[94] Abstraction—the "erasure of difference in the service of likeness or equality"—also erases, or "forgets," knowledge, rendering it, like the machine, ghostly.[95]

Thus abstraction both empowers the programmer and insists on his/her ignorance—the dream of a sovereign subject who knows and commands is constantly undone. Because abstraction exists "in the mind of the programmer," abstraction gives programmers new creative abilities. Computer scientist David Eck argues, "every programming language defines a virtual machine, for which it is the machine language. Designers of programming languages are creating computing machines as surely as the engineer who works in silicon and copper, but without the limitations imposed by materials and manufacturing technology."[96] However, this abstraction—this move away from the machine specificities—hands over, in its virtual separation of machine into software and hardware, the act of programming to the machine itself. Mildred Koss scoffed at the early notion of computers as brains because "they couldn't think in the way a human thinks, but had to be given a set of step-by-step machine instructions to be executed before they could provide answers to a specific problem"—at that time software was not considered to be an independent object.[97] The current status of software as a commodity, despite the nonrivalrous nature of "instructions," indicates the triumph of the software industry, an industry that first struggled not only financially but also conceptually to define its product. The rise of software

depends both on historical events, such as IBM's unbundling of its services from its products, and on abstractions enabled by higher-level languages. Guttag's insistence on the unreliability and incapability of human beings to understand underscores the cost of such an abstraction. Abstraction is the computer's game, as is programming in the strictest and newest sense of the word: with "data-driven" programming, for instance, machine learning/artificial intelligence (computers as source of source code) has become mainstream.

Importantly, this stratification and disciplining of labor has a much longer history: human computing itself, as David Grier has documented, moved from an art to a routinized procedure through a separation of planners from calculators.[98] Whereas the mathematician Alexis-Claude Clairaut called on two of his colleagues/friends, Joseph Lalande, Nicole-Reine Lapaute, in 1757 to calculate the date of Halley's comet's 1758 return, Gaspard Clair François Marie Riche de Prony, director of the Bureau du Cadastre, devised a system of intellectual labor to calculate metric tables in 1791. Not accidentally, the tables were part of a revolutionary governmental project: the move to the metric system by the National Assembly in order to gain control of the French economy.[99] De Prony, inspired by Adam Smith, divided the group into manual workers (unemployed pre-Revolutionary wig makers or servants who had basic arithmetic skills) and planners (experienced computers who planned the calculation). This system in turn inspired Charles Babbage's difference and analytic engines, in which the engines would replace the manual workers: according to Grier, de Prony's system showed Babbage that "the division of labor was not restricted to physical work but could be applied to 'some of the sublimest investigations of the human mind,' including the work of calculation."[100] This routinized calculation was not smoothly adopted; for a long time within the United States, such a model was resisted and, even during World War I, computers were graduate students and young assistant professors. In order to produce calculations necessary for governmental projects (such as eugenics, census, navigation, weapons, etc.) in the twentieth century, however, mass computation became the norm.

The U.S. wholesale embrace of mass calculation also coincides with a governmental project. Begun during the Great Depression as a way to put unemployed high school graduates to work, the Work Progress Administration's (WPA) Math Tables Project (MTP) produced some of the finest error-free tables in the world.[101] Indeed, it was not until the Roosevelt administration and the New Deal that the United States became seriously involved in producing mathematical tables. Since it was a WPA project, many established academics refused to be involved with it. To gain credibility, those in charge (themselves "less desirable" or unconventional PhDs) were determined to produce the most accurate tables possible. Gertrude Blanch, who ran the program with Milton Abramowitz, insists that most of the people they hired were qualified.[102] In contrast, Ida Rhodes, another PhD hired by the MTP, claims: "[Most] of the people

[who] came to us really knew nothing at all about mathematics or [even] arithmetic. Gertrude Blanch says that they were all High School graduates, and they may have been. I never checked on that. But if they were, very few of them had remembered anything about the arithmetic or the algebra or whatever mathematics they had [studied]."[103] By the end, however, they were transformed. According to Rhodes, Blanch performed miracles, "welding a malnourished, dispirited crew of people, coming from [the] Welfare Rolls, [into] a group that Leslie J. Comrie said was the 'mightiest computing team the world had ever seen.'"[104] To Rhodes, the social work involved in this project—"[salutary benefit conferred on] the spirit of those people [by] raising them from abject and self-despising people into a team that [acquired] a magnificent esprit de corps"—has been overlooked.[105] As Rhodes's rhetoric indicates, this was a patronizing if admirable project, run by "saints." Rhodes, herself partially deaf, would become an advocate for including physically challenged people in programming work. (Blanch interestingly had a more edgy view of sainthood. Describing Rhodes, she remarked, "if there are saints on earth, she's one of them. Saints may be difficult to live with but . . . it's nice to have a few around").[106]

This saintly salutary work comprised dividing the group into four categories, listed in ascending ability—the adders, the multipliers, the dividers, and the checkers—and creating worksheets so that "people who knew nothing about mathematics could [do advanced functions] by just following one step at a time."[107] The flawlessness of these tables stemmed both from these worksheets, created by Blanch, and from the degree to which these tables were checked (the Bessel function, for instance, was checked more than twenty-two times). Since the goal of the project was to keep these people busy, as well as to produce tables, accuracy was stressed over expediency and over sophistication of numerical techniques. Accuracy, according to Rhodes, became an obsession."[108]

Not surprisingly, though, the MTP computers were sometimes suspicious of their oversight. Rhodes relates, "we had impressed upon our workers over and over and over again that we were not watching them. We were not counting their output." Rather, "the only thing we asked of them is complete accuracy." This accuracy was also inscribed in the worksheets themselves in a nontransparent, repetitive manner. Rhodes and Blanch created worksheets, "in which every operation had to be done at least twice" and in which this duplicity was hidden. Rhodes explains, "for example, if we added a and b we wouldn't immediately say: add b and a. But some time later we saw to it that b got added to a, and we had arrows connecting the answers saying that these two answers should agree to, say one or two [units in] the last place. If they did not get such an agreement, then they were to [erase the pertinent portion] and [re-compute it]."[109] Again, the fact that these tables were largely unnecessary—and hence not time-sensitive—made this emphasis on accuracy over timeliness possible.

According to Rhodes, only two girls did not internalize the accuracy-ethic and cheated.[110] Rhodes revealingly narrates the dishonesty of the "colored" girl who joined the group after claiming that she was being discriminated against in another project:

> [Being] a softy, [I swallowed her story.] I should have checked with [her] boss and found out why she was not liked. But I didn't. And so I asked Gertrude's permission and she said, "All right, let's give her a chance." [And] she started working for us.
>
> Well, she hadn't been with us long enough apparently to absorb that feeling of accuracy, although, of course, we also gave her the [same] lecture that we gave everybody else. She must have thought that the more she produces, the more we will think of her and the more anxious we will be to keep her. [Her checker] reported to us that the girl was a whiz, she handed in many more sheets than anyone else; and I began to feel very proud of myself, thinking, oh, I got [me a] good girl, working so hard.
>
> You see, all that the [checker did was to examine the values, connected by the] arrows and if they agreed within one or two [units,] he was satisfied. In her case he once mentioned, "It's remarkable, they agree to the very last place." That should have given me an idea, but I was too busy with other things. Well, one evening Gertrude and I sat down to do our regular job of checking the sheets, and [when] we got [to] hers, [no values] differenced, absolutely nothing differenced. That was something we couldn't believe. How could [they] not difference? The arrows showed perfect agreement—too perfect, as a matter of fact.
>
> Well, lots of things can happen. First of all, the formula can be wrong. [Or we] could have made a mistake [in breaking down] the formula [while preparing] the worksheet. [Or] we could have made a mistake in [a sign.] We could have made a mistake in a constant. It happened to be my worksheet, so I checked [it] over: no mistake there. [She had to] copy certain information from other Tables. Maybe [I] gave her the wrong tables. [An examination showed] that she copied the correct Tables. What else could have happened? The point [is] that we were so innocent and so trusting, it never occurred to us that what really happened [could have occurred.] What had happened was that she would get the first answer, and then when she got to [it] the second time — where the arrows showed that they had to agree — and [they] didn't agree, she merely erased the [second] answer and copied down the first [one.] We found that out [when] Gertrude and I recomputed all her sheets.[111]

This remarkable story reveals the contradictions in this disciplinary system: although Rhodes denies that they judged performance by speed, she thinks she got herself a "good girl" when the "colored girl" performs quickly. Also, although math presumably requires some intellectual labor, intelligence is condemned. The "colored girl"'s ability to figure out the system, the algorithm, is denounced as cheating, and the managers' faith in their own nontransparent plans described as "trusting." These worksheets were an early form of programming: a breakdown of a complex operation into sequence of simple operations that depends on accurate and single-minded calculation. As this example makes clear, such programming depended on mind-numbingly repetitive operations by the "dumb" and the downtrodden, whose inept or deceitful actions could disrupt the task at hand. Modern computing replaces these with vacuum tubes and transistors.

As Alan Turing contended, "the class of problems capable of solution by the machine can be defined fairly specifically . . . [namely] those problems which can be solved by human clerical labour, working to fixed rules, and without understanding."[112]

Source code become "thing"—the erasure of execution—follows from the mechanization of these power relations, the reworking of subject-object relations through automation as both empowerment and enslavement and through repetition as both mastery and hell. Embedded within the notion of instruction as source and the drive to automate computing—relentlessly haunting them—is a constantly repeated narrative of liberation and empowerment, wizards and (ex-)slaves.

Automation as Sourcery

Automatic programming, what we could call programming today, reveals the extent to which automation and the history of programming cannot be considered a simple deskilling (Kraft's argument) or a march toward greater human power. Rather, through automation, expertise is both created and called into question: it is something that coders did not simply fear, but also appreciated and drove.

Automatic programming arose from a desire to reuse code and to recruit the computer into its own operation—essentially, to transform singular instructions into a language a computer could write. As Koss, an early UNIVAC programmer, explains:

> Writing machine code involved several tedious steps—breaking down a process into discrete instructions, assigning specific memory locations to all the commands, and managing the I/O buffers. After following these steps to implement mathematical routines, a sub-routine library, and sorting programs, our task was to look at the larger programming process. We needed to understand how we might reuse tested code and have the machine help in programming. As we programmed, we examined the process and tried to think of ways to abstract these steps to incorporate them into higher-level language. This led to the development of interpreters, assemblers, compilers, and generators—programs designed to operate on or produce other programs, that is, automatic programming.[113]

Automatic programming is an abstraction that allows the production of computer-enabled human-readable code—key to the commodification and materialization of software and to the emergence of higher-level programming languages.

Higher-level programming languages, unlike assembly language, explode one's instructions and enable one to forget the machine. In them, simple operations often call a function, making it a metonymic language par excellence. These languages also place everyone in the position of the planner, without the knowledge of the coder. They enable one to run a program on more than one machine—a property now assumed to be a "natural" property of software ("direct programming" led to a unique configuration of cables; early machine language could be iterable but only on the same machine—assuming, of course, no engineering faults or failures). In order to emerge

as a language or a source, software and the "languages" on which it relies had to become iterable. With programming languages, the product of programming would no longer be a running machine but rather this thing called software—something theoretically (if not practically) iterable, repeatable, reusable, no matter who wrote it or what machine it was destined for; something that inscribes the absence of both the programmer and the machine in its so-called writing.[114] Programming languages enabled the separation of instruction from machine, of imperative from action, a move that fostered the change in the name of source code itself, from "pseudo" to "source." Pseudocode intriguingly stood both for the code as language and for the code as program (i.e., source code). The manual for UNIVAC's A-2 compiler, for instance, defines pseudocode as "computer words other than the machine (C-10) code, design [sic] with regard to facilitating communications between programmer and computer. Since a pseudo-code cannot be directly executed by the computer, there must be programmed a modification, interpretation or translation routine which converts the pseudo-codes to machine instruction and routines."[115] Pseudocode, which enables one to move away from machine specificity, is called "information"—what later would become a ghostly immaterial substance—rather than code.

According to received wisdom, these first attempts to automate programming—the "pseudo"—were resisted by "real" programmers.[116] John Backus, developer of FORTRAN, claims that early machine language programmers were engaged in a "black art"; they had a "chauvinistic pride in their frontiersmanship and a corresponding conservatism, so many programmers of the freewheeling 1950s began to regard themselves as members of a priesthood guarding skills and mysteries far too complex for ordinary mortals."[117] Koss similarly argues, "without these higher-level languages and processes . . . , which democratized problem solving with the computer, I believe programming would have remained in the hands of a relatively small number of technically oriented software writers using machine code, who would have been essentially the high priests of computing."[118]

This story of a "manly" struggle against automatic programming resonates with narratives of mechanical computing itself as "feminizing" numerical analysis. Whirlwind team member Bob Everett offers the following summary of a tale describing two different ways of approaching automatic computing, which was told at Aiken's mid-1940s meeting: "One was the woman who gets married, and that's fine, and she looks ahead to a life-time of three meals a day, 365 days a year, and dishes to wash after each one of them. Her husband brings her home from the honeymoon, and she discovers he's bought her an automatic dishwasher. That's one way. The other way is the guy who decides to climb a mountain, and he buys all the rope, pitons, and one thing and another, and he goes to the mountain and finds that somebody has built a funicular railway."[119] According to this description, automatic computing is feminine or emasculating: an escape from domestic drudgery or the automation of a properly

masculine enterprise. Thus, it is not just the introduction of automatic programming that inspired narratives of masculine expertise under siege, but also the introduction of—or, more properly, the appreciation of—the (automatic) computer.

In a related manner, Hopper (and perhaps only Hopper) experienced the U.S. Navy, in particular her initial training as a thirty-seven-year-old woman, as "the most complete freedom I'd ever had." Whereas her younger counterparts rebelled "against the uniforms and the regulations," she embraced the Navy's strict structure as a release from domestic duties. As she relates, "All of a sudden I didn't have to decide anything, it was all settled. I didn't even have to bother to decide what I was going to wear in the morning, it was there. I just picked it up and put it on. So for me all of a sudden I was relieved of all minor decisions. . . . I didn't even have to figure out what I was going to cook for dinner." The difficulties of domestic life and sacrifice during World War II colored Hopper's enthusiasm, since "housekeeping had gotten to be quite a chore by then to figure out how much meat you could have and could you give dad some sugar 'cause he loved it and you might have some extra points. That's when I learned to drink most of my drinks without any sugar in them so that dad could have it. And we had very little gasoline and we had to have a car and you had to plan every trip very carefully. Well, all of a sudden I'm in midshipmen's school and all of a sudden you don't have to do any of it.[120] Importantly, though, this release was also an insertion into a well-defined system, in which one both gave and received commands. When a *Voice of America* interviewer asked, "You are supposed to command, but also to conform and obey. How do you come to terms with those two extremes?" Hopper replied, "The essential basic principle of the Navy is leadership. And leadership is a two-way street. It is loyalty up and loyalty down. Respect your superior, keep him informed of what you are doing, and take care of your crew. That is everyone's responsibility."[121]

Automatic programming, seen as freeing oneself from both drudgery and knowledge, thus calls into question the simple narrative of it as dispersing a reluctant "priesthood" of machine programmers. This narrative of resistance assumes that programmers naturally enjoyed tedious and repetitive numerical tasks and as well as developing singular solutions for their clients. The "mastery" of computing can easily be understood as "suffering." Indeed, Hopper called her early days with the Harvard Mark 1 her "sufferings" and argued, "experienced programmers are always anxious to make the computer carry out as much routine work as they can."[122] Harry Reed, an early ENIAC programmer, relays, "the whole idea of computing with the ENIAC was a sort of *hair-shirt* kind of thing. Programming for the computer, whatever it was supposed to be, was a redemptive experience—one was supposed to *suffer* to do it." According to Reed, programmers were actively trying to convince people to write small programs for themselves. In the 1970s, he "actually had to take my Division and sit everybody down who hadn't taken a course in FORTRAN, because, by God, they were going to write their own programs now. We weren't going to get computer

specialists to write simple little programs that they should have been writing.[123] Also, the first programmers were the first writers of reusable subroutines. Holberton, for instance, developed the first SORT generator to save her colleagues' time, "I felt for all the work that Betty Jean and I had done on sorting methods, it was a shame for people to have to sit down and re-do and re-code that same thing even though they could use the books to do it, if it could be done by a machine. And that's the reason, and it only took six months to program the thing. That's six more months."[124] Thus, rather than programmers circling the wagons to protect their positions, it would seem that many programmers themselves welcomed and contributed to the success of automatic programming.

As well, since programmers were in incredible demand in the 1950s through the 1960s, the need to create boundaries to protect jobs seems odd. Although compilers and interpreters may not have been accepted immediately, especially by those already trained in machine programming, the resistance may have stemmed more from the work environment than from personal arrogance. Coders were under great pressure to be as efficient as possible. As Holberton and Bartik relay in a 1973 interview, early coders often developed a persecution complex, because machine time was the most important and expensive thing:

BARTIK: The worst sin that you could commit was to waste that machine time. So that we really became paranoid.
HOLBERTON: Mhm. Efficiency.
BARTIK: We thought everybody was after us.
TROPP: [Laughter]
BARTIK: For our inefficiency.
HOLBERTON: You wasted one add time, you were being inefficient.
BARTIK: So it was fine for us to struggle for two days to cut off the slightest amount on that machine.[125]

Compilers were arguably accepted because the demand for programmers meant a loss in quality (an ever widening recruitment)—programming efficiently in machine language therefore became a mark of expertise. In this sense, the introduction of automatic programming, which set a certain standard of machine efficiency, helped to produce the priesthood it was supposedly displacing.

Corporate and academic customers, for whom programmers were orders of magnitude cheaper per hour than computers, do seem to have resisted automatic programming. Jean Sammet, an early female programmer, relates, in her influential *Programming Languages: History and Fundamentals*, that customers objected to compilers on the ground that they "could not turn out object code as good as their best programmers. A significant selling campaign to push the advantages of such systems was underway at that time, with the spearhead being carried for the numerical scientific languages (i.e., FORTRAN) and for 'English-language-like' business data-processing languages by

Remington Rand (and Dr. Grace Hopper in particular)."[126] This selling campaign not only pushed higher-level languages (by devaluing humanly produced programs), it also pushed new hardware: to run these programs, one needed more powerful machines. The government's insistence on standardization, most evident in the development and widespread use of COBOL, itself a language designed to open up programming to a wider range of people, fostered the general acceptance of higher-level languages, which again were theoretically, if not always practically, machine independent or iterable. The hardware-upgrade cycle was normalized in the name of saving programming time.

This "selling campaign" led to what many have heralded as the democratization of programming, the opening of the so-called priesthood of programmers. In Sammet's view, this was a partial revolution

> in the way in which computer installations were run because it became not only possible, but quite practical to have engineers, scientists, and other people actually programming their own problems without the intermediary of a professional programmer. Thus the conflict of the open versus closed shop became a very heated one, often centering [on] the use of FORTRAN as the key illustration for both sides. This should not be interpreted as saying that all people with scientific numerical problems to solve immediately sat down to learn FORTRAN; this is clearly not true but such a significant number of them did that it has had a major impact on the entire computer industry. One of the subsidiary side effects of FORTRAN was the introduction of FORTRAN Monitor System [IB60]. This made the computer installation much more efficient by requiring less operator intervention for the running of the vast number of FORTRAN (as well as machine language) programs.[127]

The democratization or "opening" of computing, which gives the term *open* in *open source* a different resonance, would mean the potential spread of computing to those with scientific numerical problems to solve and the displacement of human operators by operating systems. But the language of priests and wizards has hardly faded and scientists have always been involved with computing, even though computing has not always been considered to be a worthy scientific pursuit. The history of computing is littered with moments of "computer liberation" that are also moments of greater obfuscation.[128]

Higher level programming languages—automatic programming—may have been sold as offering the programmer more and easier control, but they also necessitated blackboxing even more the operations of the machine they supposedly instructed. Democratization did not displace professional programmers but rather buttressed their position as professionals by paradoxically decreasing their real power over their machines, by generalizing the engineering concept of information.

So what are we to do with these contradictions and ambiguities? As should be clear by now, these many contradictions riddling the development of automatic programming were key to its development, for the automation of computing is both an

acquisition of greater control and freedom, and a fundamental loss of them. The narrative of the "opening" of programming reveals the tension at the heart of programming and control systems: are they control systems or servomechanisms (Norbert Wiener's initial name for them)? Is programming a clerical activity or an act of Hobbesian mastery? Given that the machine takes care of "programming proper"—the sequence of events during execution—is programming programming at all? What is after all compacted in the coinciding changes in the titles of "operators" to "programmers" and of "mathematicians" to "programmers"? The notion of the priesthood of programming erases this tension, making programming always already the object of jealous guardianship, and erasing programming's clerical underpinnings.[129]

Programming in the 1950s does seem to have been fun and fairly gender balanced, in part because it was so new and in part because it was not as lucrative as hardware design or even sales: the profession was gender neutral in hiring if not pay because it was not yet a profession.[130] The "ENIAC girls" were first hired as subprofessionals, and some had to acquire more qualifications in order to retain their positions. As many female programmers quit to have children or get married, men (and compilers) took their increasingly lucrative positions. Programming's clerical and arguably feminine underpinnings—both in terms of personnel and of command structure—became buried as programming sought to become an engineering and academic field in its own right.[131] Democratization did not displace professional programmers but rather buttressed their position as professionals by paradoxically decreasing their real power over their machines. It also, however, made programming more pleasurable.

Causal Pleasure

The distinction between programmers and users is gradually eroding. With higher-level languages, programmers are becoming more like simple users. Crucially, though, the gradual demotion of programmers has been offset by the power and pleasure of programming. To program in a higher-level language is to enter a magical world—it is to enter a world of logos, in which one's code faithfully represents one's intentions, albeit through its blind repetition rather than its "living" status.[132] Edwards argues, "programming can produce strong sensations of power and control" because the computer produces an internally consistent if externally incomplete microworld, a "simulated world, entirely within the machine itself, that does not depend on instrumental effectiveness. That is, where most tools produce effects on a wider world of which they are only a part, the computer contains its own worlds in miniature. . . . In the microworld, as in children's make-believe, the power of the programmer is absolute."[133] Joseph Weizenbaum, MIT professor, creator of ELIZA (an early program that imitated a Rogerian therapist) and member of the famed MIT AI (Artifical Intelligence) lab, similarly contends:

> The computer programmer . . . is a creator of universes for which he alone is the lawgiver. So, of course, is the designer of any game. But universes of virtually unlimited complexity can be created in the form of computer programs. Moreover, and this is a crucial point, systems so formulated and elaborated *act out* their programmed scripts. They compliantly obey their laws and vividly exhibit their obedient behavior. No playwright, no stage director, no emperor, however powerful, has ever exercised such absolute authority to arrange a stage or a field of battle and to command such unswervingly dutiful actors or troops.[134]

The progression from playwright to stage director to emperor is telling: programming languages, like neoliberal economics, model the world as a "game."[135] To return to the notion of "code is law," programming languages establish the programmer as a sovereign subject, for whom there is no difference between command given and command completed. As a lawgiver more powerful than a playwright or emperor, the programmer can "say" "let there be light" and there is light. Iterability produces both language and subject. Importantly, Weizenbaum views the making performative or automatically executable of words as the imposition of instrumental reason, inseparable from the process of "enlightenment" critiqued by the Frankfurt school.[136] Instrumental reason, he argues, "has made out of words a fetish surrounded by black magic. And only the magicians have the rights of the initiated. Only they can say what words mean. And they play with words and they deceive us."[137]

Programming languages offer the lure of visibility, readability, logical if magical cause and effect. As Brooks argues, "one types the correct incantation on the keyboard, and a display screen comes to life, showing things that never were nor could be."[138] One's word creates something living. Consider this ubiquitous "hello world" program written in C++ ("hello world" is usually the first program a person will write):

```
// this program spits out "hello world"
#include <iostream.h>
int main ()
{
  cout << "Hello World!";
  return 0;
}
```

The first line is a comment line, explaining to the human reader that this program spits out "Hello World!." The next line directs the compiler's preprocessor to include iostream.h, a standard file to deal with input and output to be used later. The third line, "int main (),"begins the main function of the program; "cout << 'Hello World!';" prints "Hello World!" to the screen ("cout" is defined in iostream.h); "return 0" terminates the main function and causes the program to return a 0 if it has run correctly.

Although not immediately comprehensible to someone not versed in C++, this program nonetheless seems to make some sense, and seems to be readable. It comprises a series of imperatives and declaratives that the computer presumably understands and obeys. When it runs, it follows one's commands and displays "Hello World!."

It is no accident that "hello world" is the first program one learns because it is easy, demonstrating that we can produce results immediately. This ease, according to Weizenbaum, is what makes programming so seductive and dangerous:

> It happens that programming is a relatively easy craft to learn. . . . And because programming is almost immediately rewarding, that is, because a computer very quickly begins to behave somewhat in the way the programmer intends it to, programming is very seductive, especially for beginners. Moreover, it appeals most to precisely those who do not yet have sufficient maturity to tolerate long delays between an effort to achieve something and the appearance of concrete evidence of success. Immature students are therefore easily misled into believing that they have truly mastered a craft of immense power and of great importance when, in fact, they have learned only its rudiments and nothing substantive at all.[139]

The seeming ease of programming hides a greater difficulty—executability leads to unforeseen circumstances, unforeseen or buggy repetitions. Programming offers a power that, Weizenbaum argues, corrupts as any power does.[140] *What corrupts, Weizenbaum goes on to explain, however, is not simply ease, but also this combination of ease and difficulty*. Weizenbaum argues that programming creates a new mental disorder: the compulsion to program, which he argues hackers, who "hack code" rather than "work," suffer from (although he does note that not all hackers are compulsive programmers).[141]

To explain this addiction, Weizenbaum explains the parallels between "the magical world of the gambler" and the magical world of the hacker—both entail megalomania and fantasies of omnipotence, as well as a "pleasureless drive for reassurance."[142] Like gambling, programming can be compulsive because it both rewards and challenges the programmer. It is driven by "two apparently opposing facts: first, he knows that he can make the computer do anything he wants it to do; and second, the computer constantly displays undeniable evidence of his failures to him. It reproaches him. There is no escaping this bind. The engineer can resign himself to the truth that there are some things he doesn't know. But the programmer moves in a world entirely of his own making. The computer challenges his power, not his knowledge."[143] According to Weizenbaum, because programming engages power rather than truth, it can induce a paranoid megalomania in the programmer.[144] Because this knowledge is never enough, because a new bug always emerges, because an unforeseen wrinkle causes divergent unexpected behavior, the hacker can never stop. Every error seems correctable; every error points to the hacker's lack of foresight; every error leads to another. Thus, unlike the "useful programmer," who "works" by solving the problem at hand and carefully documents his code, the hacker aimlessly hacks code: programming

becomes a technique, a game without a goal and thus without an end. Hackers' skills are thus "disembodied" and this disembodiment transforms their physical appearance: Weizenbaum describes them as "bright young men of disheveled appearance, often with sunken glowing eyes . . . sitting at computer consoles, their arms tensed and waiting to fire their fingers, already poised to strike, at the buttons and keys on which their attention seems to be as riveted as a gamer's on the rolling dice.[145]

Although Weizenbaum is quick to pathologize hackers as pleasureless pitiful creatures, hackers themselves emphasize programming as pleasurable—and their lack of "usefulness" can actually be what is most productive and promising about programming. Linus Torvalds, for instance, argues that he, as an eternal grad student, decided to build the Linux operating system core just "for fun." Torvalds further views the decisions programming demands as rescuing programming from becoming tedious. "Blind obedience on its own, while initially fascinating," he writes, "obviously does not make for a very likable companion. In fact, that part gets boring fairly quickly. What makes programming so engaging is that, while you can make the computer do what you want, you have to figure out *how*."[146] Richard Stallman, who fits Weizenbaum's description of a hacker (and who was in the AI lab, probably building those indispensable functions) likewise emphasizes the pleasure, but more important the "freedom" and "freeness" associated with programming—something that stems from programming as not simply the production of a commercial (or contained) product. Hacking reveals the extent to which source code can become a fetish: something endless that always leads us pleasurably, as well as anxiously, astray.

Source Code as Fetish

Source code as source means that software functions as an axiom, as "a self-evident proposition requiring no formal demonstration to prove its truth, but received and assented to as soon as it is mentioned."[147] In other words, whether or not source code is only a source after the fact or whether or not software can be physically separated from hardware,[148] software is always posited as already existing, as the self-evident ground or source of our interfaces. Software is axiomatic. As a first principle, it fastens in place a certain neoliberal logic of cause and effect, based on the erasure of execution and the privileging of programming that bleeds elsewhere and stems from elsewhere as well.[149] As an axiomatic, it, as Gilles Deleuze and Félix Guattari argue, artificially limits decodings.[150] It temporarily limits what can be decoded, put into motion, by setting up an artificial limit—the artificial limit of programmability—that seeks to separate information from entropy, by designating some entropy information and other "non-intentional" entropy noise. Programmability, discrete computation, depends on the disciplining of hardware and programmers, and the desire for a programmable axiomatic code. Code, however, is a medium in the full sense of the word. As a

medium, it channels the ghost that we imagine runs the machine—that we see as we don't see—when we gaze at our screen's ghostly images.

Understood this way, source code is a fetish. According to the OED, a fetish was originally an ornament or charm worshipped by "primitive peoples . . . on account of its supposed inherent magical powers."[151] The term *fetisso* stemmed from the trade of small wares and magic charms between the Portuguese merchants and West Africans; Charles de Brosses coined the term *fetishism* to describe "primitive religions" in 1757. According to William Pietz, Enlightenment thinkers viewed fetishism as a "false causal reasoning about physical nature" that became "the definitive mistake of the pre-enlightened mind: it superstitiously attributed intentional purpose and desire to material entities of the natural world, while allowing social action to be determined by the . . . wills of contingently personified things, which were, in truth, merely the externalized material sites fixing people's own capricious libidinal imaginings."[152] That is, fetishism, as "primitive causal thinking," derived causality from "things"—in all the richness of this concept—rather than from reason:

> Failing to distinguish the intentionless natural world known to scientific reason and motivated by practical material concerns, the savage (so it was argued) superstitiously assumed the existence of a unified causal field for personal actions and physical events, thereby positing reality as subject to animate powers whose purposes could be divined and influenced. Specifically, humanity's belief in gods and supernatural powers (that is, humanity's unenlightenment) was theorized in terms of prescientific peoples' substitution of imaginary personifications for the unknown physical causes of future events over which people had no control and which they regarded with fear and anxiety.[153]

A fetish allows one to visualize what is unknown—to substitute images for causes. Fetishes allow the human mind both too much and not enough control by establishing a "unified causal field" that encompasses both personal actions and physical events. Fetishes enable a semblance of control over future events—a possibility of influence, if not an airtight programmability—that itself relies on distorting real social relations into material givens.

This notion of fetish as false causality has been most important to Karl Marx's diagnosis of capital as fetish. Marx famously argued:

> the commodity-form . . . is nothing but the determined social relation between men themselves which assumes here, for them, the phantasmagoric form of a relation between things. In order, therefore, to find an analogy we must take a flight into the misty realm of religion. There the products of the human head appear as autonomous figures endowed with a life of their own, which enter into relations both with each other and with the human race. So it is in the world of commodities with the products of men's hands. I call this the . . . fetishism.[154]

The capitalist thus confuses social relations and the labor activities of real individuals with capital and its seemingly magical ability to reproduce. For, "it is in interest-

bearing capital . . . that capital finds its most objectified form, its pure fetish form. . . . Capital—as an entity—appears here as an independent source of value; a something that creates value in the same way as land [produces] rent, and labor wages."[155] Both these definitions of fetish also highlight the relation between things and men: men and things are not separate, but rather speak with and to one another. That is, things are not simply objects that exist outside the human mind, but are rather tied to events, to the timing of events.

The parallel to source code seems obvious: we "primitive folk" worship source code as a magical entity—as a source of causality—when in truth the power lies elsewhere, most importantly, in social and machinic relations. If code is performative, its effectiveness relies on human and machinic rituals. Intriguingly though, in this parallel, Enlightenment thinking—a belief that knowing leads to control, to a release from tutelage—is not the "solution" to the fetish, but, rather, what grounds it, for source code historically has been portrayed as the solution to wizards and other myths of programming: machine code provokes mystery and submission; source code enables understanding and thus institutes rational thought and freedom. Knowledge, according to Weizenbaum, sustains the hacker's aimless actions. To offer a more current example of this logic than the FORTRAN one cited earlier, Richard Stallman, in his critique of nonfree software, has argued that an executable program "is a mysterious bunch of numbers. What it does is secret."[156] Against this magical execution, source code supposedly enables an understanding and a freedom—the ability to map and know the workings of the machine, but, again, only through a magical erasure of the gap between source and execution, an erasure of execution itself. If we consider source code as fetish, the fact that source code has hardly deprived programmers of their priestlike/wizard status makes complete sense. If anything, such a notion of programmers as superhuman has been disseminated ever more and the history of computing—from direct manipulation to hypertext—has been littered by various "liberations."

But clearly, source code can do and be things: it can be interpreted or compiled; it can be rendered into machine-readable commands that are then executed. Source code is also read by humans and is written by humans for humans and is thus the source of some understanding. Although Ellen Ullman and many others have argued, "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," source code must be able to function, even if it does not function—that is, even if it is never executed.[157] Source code's readability is not simply due to comments that are embedded in the source code, but also due to English-based commands and programming styles designed for comprehensibility. This readability is not just for "other programmers." When programming, one must be able to read one's own program—to follow its logic and to predict its outcome, whether or not this outcome coincides with one's prediction.

This notion of source code as readable—as creating some outcome regardless of its machinic execution—underlies "codework" and other creative projects. The Internet artist Mez, for instance, has created a language called mezangelle that incorporates formal code and informal speech. Mez's poetry deliberately plays with programming syntax, producing language that cannot be executed, but nonetheless draws on the conventions of programming language to signify.[158] Codework, however, can also work entirely within an existing programming language. Graham Harwood's perl poem, for example, translates William Blake's nineteenth-century poem "London" into London.pl, a script that contains within it an algorithm to "find and calculate the gross lung-capacity of the children screaming from 1792 to the present."[159] Regardless of whether or not it can execute, code can be—must be—worked into something meaningful. Source code, in other words, may be the source of things other than the machine execution it is "supposed" to engender.

Source code as fetish, understood psychoanalytically, embraces this nonteleological potential of source code, for the fetish is a deviation that does not "end" where it should. It is a genital substitute that gives the fetishist nonreproductive pleasure. It allows the child to combat castration—his inscription within the world of paternal law and order—for both himself and his mother, while at the same time accommodating to his world's larger oedipal structure. It both represses and acknowledges paternal symbolic authority. According to Freud, the fetish, formed the moment the little boy discovers his mother's "lack," is "a substitute for the woman's (mother's) phallus which the little boy once believed in and does not wish to forego."[160] As such, it both fixes a singular event—turning time into space—and enables a logic of repetition that constantly enables this safeguarding. As Pietz argues, "the fetish is always a meaningful fixation of a singular event; it is above all a 'historical' object, the enduring material form and force of an unrepeatable event. This object is 'territorialized' in material space (an earthly matrix), whether in the form of a geographical locality, a marked site on the surface of the human body, or a medium of inscription or configuration defined by some portable or wearable thing."[161] Even though it fixes a singular event, the fetish works only because it can be repeated, but again, what is repeated is both denial and acknowledgment, since the fetish can be "the vehicle both of denying and asseverating the fact of castration."[162] Slavoj Žižek draws on this insight to explain the persistence of the Marxist fetish:

> When individuals use money, they know very well that there is nothing magical about it—that money, in its materiality, is simply an expression of social relations . . . on an everyday level, the individuals know very well that there are relations between people behind the relations between things. The problem is that in their social activity itself, in what they are *doing*, they are *acting* as if money, in its material reality is the immediate embodiment of wealth as such. They are fetishists in practice, not in theory. What they "do not know,"

> what they misrecognize, is the fact that in their social reality itself—in the act of commodity exchange—they are guided by the fetishistic illusion.[163]

Fetishists, importantly, know what they are doing—knowledge, again, is not an answer to fetishism, but rather what sustains it. The knowledge that source code offers is no cure for source code fetishism: if anything, this knowledge sustains it. As the next chapter elaborates, the key question thus is not "what do we know?" but rather "what do we do?"

To make explicit the parallels, source code, like the fetish, is a conversion of event into location—time into space—that does affect things, although not necessarily in the manner prescribed. Its effects can be both productive and nonexecutable. Also, in terms of denial and acknowledgment, we know very well that source code in that state and without the intercession of other "layers" is not executable, yet we persist in treating it as so. And it is this glossing over that makes possible the ideological belief in programmability.

Code as fetish means that computer execution deviates from the so-called source, as source program does from programmer. Turing, in response to the objection that computers cannot think because they merely follow human instructions, contends:

> Machines take me by surprise with great frequency. . . . The view that machines cannot give rise to surprises is due, I believe, to a fallacy to which philosophers and mathematicians are particularly subject. This is the assumption that as soon as a fact is presented to a mind all consequences of that fact spring into the mind simultaneously with it. It is a very useful assumption under many circumstances, but one too easily forgets that it is false. A natural consequence of doing so is that one then assumes that there is no virtue in the mere working out of consequences from data and general principles.[164]

This erasure of the vicissitudes of execution coincides with the conflation of data with information, of information with knowledge—the assumption that what is most difficult is the capture, rather than the analysis, of data. This erasure of execution through source code as source creates an intentional authorial subject: the computer, the program, or the user, and this source is treated as the source of meaning. The fact that there is an algorithm, a meaning intended by code (and thus in some way knowable), sometimes structures our experience with programs. When we play a game, we arguably try to reverse engineer its algorithm or at the very least link its actions to its programming, which is why all design books warn against coincidence or random mapping, since it can induce paranoia in its users. That is, because an interface is programmed, most users treat coincidence as meaningful. To the user, as with the paranoid schizophrenic, there is always meaning: whether or not the user knows the meaning, s/he knows that it regards him or her. To know the code is to have a form of "X-ray vision" that makes the inside and outside coincide, and the act of revealing sources or connections becomes a critical act in and of itself.[165] Code

as source leads to that bizarre linking of computers to visual culture, to transparency, which constitutes the subject of chapter 2.

Code as fetish thus underscores code as thing: code as a "dirty window pane," rather than as a window that leads us to the "source." Code as fetish emphasizes code as a set of relations, rather than as an enclosed object, and it highlights both the ambiguity and the specificity of code. Code points to, it indicates, something both specific and nebulous, both defined and undefinable. Code, again, is an abstraction that is haunted, a source that is a re-source, a source that renders the machinic—with its annoying specificities or "bugs"—ghostly. As Thomas Keenan argues, "haunting can only be thought as the difficult (simultaneous and impossible) movement of remembering and forgetting, inscribing and erasing, the singular and the different."[166] Embracing software as thing, in theory and in practice, opens us to the ways in which the fact that we cannot know software can be an enabling condition: a way for us to engage the surprises generated by a programmability that, try as it might, cannot entirely prepare us for the future.