

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI[®]

THE ART OF CODE

Maurice J. Black

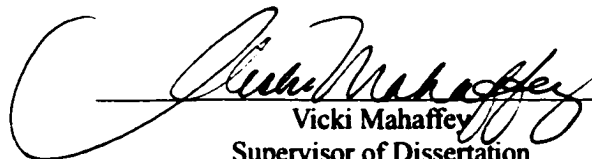
A DISSERTATION

in

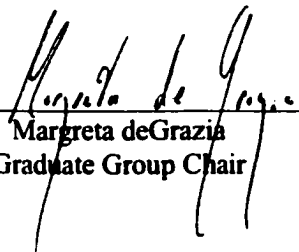
English

**Presented to the Faculties of the University of Pennsylvania
in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy**

2002



Vicki Mahaffey
Supervisor of Dissertation



Margreta deGrazia
Graduate Group Chair

UMI Number: 3072974

Copyright 2002 by
Black, Maurice Joseph

All rights reserved.

UMI[®]

UMI Microform 3072974

Copyright 2003 by ProQuest Information and Learning Company.
All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

COPYRIGHT

Maurice J. Black

2002

To my parents, with love and gratitude.

Acknowledgements

I am indebted to my dissertation adviser, Vicki Mahaffey, for her inspiring intellectual mentorship during my graduate school career and for her consistently perceptive and challenging responses to my writing. I am also grateful to my dissertation readers, James English and Jean-Michel Rabaté, for their thoughtful comments on this project. In addition, I want to acknowledge the financial support provided by the University of Pennsylvania and by the Mellon Foundation.

I want to thank my parents, Thomas and Wilhelmina Black, my brother Derek, and my sister Sandra; their unflagging support and good humor helped keep me sane and happy during my graduate studies. My extended family in the United States have always welcomed me with open arms and good cheer; I am especially grateful to John McDermott and Carole Fleahman for giving me a home when I needed one. Katherine and Brian O'Connor have been an invaluable source of help and encouragement while I worked on this dissertation; indeed, many of my most productive writing sessions took place at their beautiful Oregon home. I also owe a huge debt of gratitude to Paul Ferry, Jennifer Chachkes, and Alexandra Krijgsman for their enduring loyalty, friendship, and understanding.

My deepest thanks go to Erin O'Connor. Her unerring critical eye and editorial acumen played a huge role in bringing this dissertation to fruition; more importantly, her boundless kindness, generosity, and patience sustained me as I researched and wrote it. I cannot thank her enough for all that she has done.

Abstract

THE ART OF CODE

Maurice J. Black

Vicki Mahaffey

The Art of Code originates at the nexus of literature's and computing culture's related but distinct aesthetic systems. Arguing that software's increasing abstraction from hardware has defined computer programming practices for the last half-century, this dissertation shows how that abstraction has shaped the aesthetics, politics, and professional culture of programming. Specifically, the dissertation examines how some programmers have adopted a literary approach to coding, describing carefully crafted code as "beautiful," "elegant," "expressive," and "poetic"; writing and reading programs as literary texts; and even producing hybrid artifacts that are at once poems and programs. The project has two central goals: first, to show how identifiably linguistic sensibilities have influenced programming theory and culture; second, to show how programming theory, as a body of knowledge that thinks deeply about the semantics and organization of textual structures, can contribute to the project of literary study. As such, the dissertation's three chapters work together to provide both an aesthetic history of computing culture and a related analysis of how programming aesthetics can inform modern criticism. Chapter One outlines a range of historical, technological, philosophical, political, and legal conceptions of what software is, focusing on how those conceptions have shaped our ideas about how software should be written, distributed, and protected. Chapter Two discusses the aesthetic history of code, examining the importance of the literary ideal to programming culture. Chapter Three examines

the intersections between modern programming theory and the authorial practices employed by James Joyce, arguing that understanding computer programming as a literary technique, a mode of writing with inherent artistic capabilities, enables a powerful re-imagining of the complex linguistic and structural experiments Joyce conducts in *Finnegans Wake*. Concluding with a reconsideration of Martin Heidegger's conceptions of *technē* and *poiēsis*, *The Art of Code* aims to initiate philosophical inquiry into the complex, dynamic interrelationship between the semantics of computer programming and of literature.

Contents

Acknowledgements	iv
Abstract	v
Introduction	1
1 Crossed Wires: Historiographies of Code	21
1.1 From Code to Language	30
1.2 Inventing and Regulating Software	50
1.3 Rebel Code: Free Software and Open Source	58
1.4 Consensual Hallucinations: Producing a Metaphysics of Code	66
1.5 A Manifesto for Informed Theory	79
2 “At the Edge of Language”: The Art of Code	89
2.1 Programming Aesthetics	96
2.2 Intellectual Property Law and the Ontology of Code	107
2.3 Long Close Readings: Literary Criticism and Code	119
2.4 Literate Programming	131
2.5 Perl Poetry	137
3 “Harmonic Condenser Enginium”: Object-Oriented Joyce	149
3.1 Engineering Joyce	154
3.2 Object-Oriented Programming	165

3.3	“Joyceware”: The Object-Oriented <i>Wake</i>	174
3.4	A Modernist Genealogy of Object-Oriented Design	188
	Conclusion: The Poetics of <i>Technē</i>	202
	Works Cited	212

Introduction

I

Larry and Andy Wachowski's 1999 film *The Matrix* presents a world, some two hundred years from the present, that has been reduced by its own technological self-indulgence to a shattered, blackened shell of itself, a stark, ruined landscape filled with the burnt stumps of skyscrapers and overhung by roiling, scorched skies. In its vision of an earth destroyed by humanity's war against its own too-human machines, *The Matrix* belongs to a post-apocalyptic tradition that includes H. G. Wells's *Time Machine*, Franklin Schaffner's *Planet of the Apes*, and Ridley Scott's *Blade Runner*—but with one important difference. In Wells's fiction and in recent film, the fantasy of a devastated future is rendered as an unrecognizable alienation from our own comfortable present. In *Planet of the Apes*, for example, the main character assumes he has crash-landed on a primitive planet, and the full horror of his situation is revealed only at the end: it is not until he stumbles across a half-buried, defaced Statue of Liberty that he realizes the desolate place he has discovered is the Earth of the future. In *The Matrix*, by contrast, post-apocalyptic ruination is largely hidden from sight by an eponymous software application so powerful and complex that it is capable of (re)generating reality itself.

People living in the Matrix do not realize that their lives are programmed to seem to be real: that is the nature of the Matrix, to simulate "reality" so thoroughly that the fact of that simulation ceases to be recognizable as such. Nor do people living in the Matrix realize that their bodies have been turned into the batteries that power the Matrix along: that is the purpose of the Matrix, to keep itself up and running by harnessing the whole energy of humankind. In its fantasy of a post-apocalyptic world programmed to look and feel exactly like 1999, *The Matrix* brings the tradition inaugurated by Wells full circle. Whereas Wells dreamt of a machine that would allow people to travel back and forth through time, *The Matrix* envisions a program that prevents people from seeing that time itself has run out. Whereas Wells depicted a weirdly regressive future populated by creepy mutants, *The Matrix* depicts a weirdly familiar future populated by ourselves. In the one, crucial differences between us and them, now and then, what is and what might be are maintained in the moment of flirting with their disappearance. In the other, there are no

differences: *The Matrix* does not ask us to consider what might happen someday; it asks us to consider whether we are already in the Matrix.

Through the trope of the non-existent future, then, *The Matrix* is really fantasizing about the present. On one level, this is so obvious it hardly needs saying: it's hard not to notice that a story about a world decimated by computers that have become too smart for our own good is a story that is running on the worried excitement animating our confused, impassioned debates about cyberspace and artificial intelligence. On another level, though, the manner of the film's engagement with the present moment is far from obvious: if you're unfamiliar with the history of computing, it's hard to notice that *The Matrix* is as much in love with digital technology as it is anxious about it. Indeed, its seemingly technophobic story about a world decimated by computers is also a story about how computer programmers can save the world.

In *The Matrix*, it isn't *whether* you compute that matters—the film knows better than to propose an unworkable distinction between technophiles who plug in and technophobes who don't—but *how* you compute. In a world where electronic neuronal implants collapse meaningful distinctions between experiential and virtual reality, no life is untouched by computers; instead, people are differentiated by how well they understand the computing code that quite literally shapes their lives. Indeed, the film's "good guys," the small band of rebel programmers who hope to free the human race from enslavement, are the people who have made themselves responsible for understanding how the Matrix works. Their programming skill is such that they can not only conceptualize how an enormous virtual reality generator such as the Matrix could be created, but they can also see through the aesthetic illusions the Matrix generates and into the computing code that underpins it. The rebel hackers keep an eye on the Matrix by sitting in front of screens that scroll endless columns of green, luminous code; in turn, their eyes automatically translate that code into the virtual reality it projects: buildings, streets, people, and so on. As the rebels become more adept at reading, understanding, and translating the Matrix's code, they learn to move, work, and fight more effectively within the hallucinatory structure the program generates. The film's action thus moves fluidly between two interconnected and competing realities: the illusory facade of the virtual Matrix and the barren, desolate world underpinning it. In the former, invisible computing

code is the agent of coercive domination; in the latter, reading and “hacking” visible code becomes a means of resistance and liberation.

I begin this study with *The Matrix* because the film’s handling of computing code and computer programming, although steeped in science fiction convention, is not all futuristic fantasy. In fact, by making the visibility of computer code the site of a political struggle for human freedom, *The Matrix* imports as a central structural element a charged computing debate that is by now more than a quarter-century old: corporate interests in keeping software source code secret versus a grassroots libertarian programming philosophy that historians and anthropologists of computing culture have called the “hacker ethic.” As explained by writers such as Steven Levy, Eric Raymond, and Pekka Himanen, the hacker ethic evolved as the discipline of programming itself evolved, effectively becoming an underground constitution governing the writing and distribution of computer programs. The hacker ethic argues that program source code should circulate freely and openly among programmers and computer users, and that corporations and courts should not attempt to restrict that free circulation by means of patents, copyrights, trade secrets, or other technical or legal mechanisms. Indeed, the alliance *The Matrix* draws between visible, open code and the future of human liberty strongly resonates with arguments about computing code advanced by programming activists such as Richard Stallman, founder of the Free Software Foundation, and legal theorists such as Lawrence Lessig, author of books outlining the intersection of constitutional law and computing architecture. Given the ubiquity of computing technology in the twenty-first century Western world, Stallman, Lessig, and others argue that corporate or governmental restrictions on code—particularly software companies’ proprietary “blackboxing” practices and governments’ willingness to legislate and enforce restrictive intellectual property laws—are so far-reaching that they have the potential to erode not only the freedom of programmers but the very liberal tradition itself. Specifically, they contend that to withhold code from users infringes on freedoms that are essential to truly liberal societies: Lessig speaks of how closed computing architectures are simultaneously architectures of control that ignore the democratic ideal; Stallman speaks more locally, focusing on how blackboxing infringes on the user’s freedom of expression: blackboxed code is code that the reader may neither read nor adapt to his needs. So unrelenting

are Stallman and Lessig on this point that their efforts to defend liberal ideals for the digital age resonate strongly with Friedrich Hayek's rationale for writing his *Constitution of Liberty*: "If old truths are to retain their hold on men's minds, they must be restated in the language and concepts of successive generations" (1). Defenders of free software have in this regard modernized the classic defense of liberty, upgrading its rhetoric and reach for our digital age. Although the chapters that follow focus on programming literary aesthetics, they must by necessity take into account these intimately related political and ethical struggles for programming freedom.¹

The Matrix dramatizes these debates about programming freedom when it centers on a hacker named Neo, a sort of digitized Christ figure who is given the unenviable job of redeeming humankind from the evil posed by unacknowledged, unseen code. When a cellular phone call from an underground resistance movement informs him that he is "The One" who can save the world, Neo responds with a self-doubt and an epistemological incredulity that prevail through most of the film. His insecurity and skepticism only lift when Neo, during a climactic battle with the Agents who program, patrol, and protect the Matrix, finally begins to comprehend the nature of the electronic simulation that surrounds him, an epiphany the film records through Neo's eyes as the moment he learns to see past the program's projected reality and into its underlying source code. This understanding, and the confidence that comes with it, translates immediately into absolute mastery of the dominating system. One moment, an unenlightened Neo's life is in danger of being snuffed out by his own residual belief in Matrix-generated reality (though his body is not

¹Concerning itself specifically with corporate and legal restrictions on computing source code, this study foregoes extensive discussion of electronic civil liberties issues. It should be noted, however, that legislation governing matters such as digital copyright or strong encryption technology can seriously inhibit programmers' attempts to write, publish, and circulate code. Forbidding any attempt to circumvent electronic copyright protection on digital media, the Digital Millennium Copyright Act explicitly prevents certain kinds of program from being written. In 2001, the FBI jailed Russian programmer Dimitri Sklyarov, who had written a program for circumventing copy-protection on electronic books, and held him in United States custody for five months—even though Sklyarov had written his program in Russia and had not broken any United States laws. For a detailed discussion of recent digital copyright legislation, see Marcia Wilbur's book *The Digital Millennium Copyright Act*. For an excellent history of the struggles between computer hackers and the U.S. Federal government over the regulation of strong encryption technology, see Steven Levy's *Crypto: How the Code Rebels Beat the Government—Saving Privacy in the Digital Age*.

really “there,” given that the rebels project themselves into the Matrix by plugging their minds temporarily into the program’s virtual reality, he will die if his mind *believes* he has been killed); the next moment, a newly empowered Neo defeats the forces of evil with one hand while his eyes process the bright green code surging all around him. In *The Matrix*, domination originates in inaccessible source code: the Matrix owns those who can’t, or won’t, familiarize themselves with it. Likewise, the only way to defeat the Matrix is to access and reprogram its source code. The film’s message is clear: the victim of the future is the complacent luddite who doesn’t think twice about depending on machines whose inner workings are a mystery; the savior is the hacker, the individual whose complex passion for programming forms the basis of an entire ethical system.²

²The convoluted history of the word “hacker” deserves some clarification at this point. Until the mid-1980s, “hacker” indicated a skillful, talented, and dedicated computer programmer; this is how Steven Levy uses the word in his laudatory 1984 book *Hackers: Heroes of the Computer Revolution*. The word acquired negative connotations when mid-1980s journalists adopted it to describe teenage pranksters and rogue programmers who put their computer skills to malicious or criminal use. Real hackers, trying to preserve the purity of their terminology, coined the term “cracker” to describe these computer pranksters and criminals; the neologism hints most obviously at safe-cracking, but, true to the habitual wordplay of hacker culture, also conjures the image of a boastful, lying braggart (a Renaissance usage). The *Jargon File*, a glossary of computing terminology developed collaboratively across the ARPANET in the early 1970s and still maintained today, quotes Shakespeare’s *King John* to make this point clear: “What cracker is this same that deafs our ears / With this abundance of superfluous breath?” (Act I, Scene ii). Despite these efforts, the neologism largely failed to catch on, and the negative connotation of “hacker” remains the dominant one in popular use today. Attitudes toward the term vary, even within computing culture. Finnish programmer Linus Torvalds, creator of the Linux operating system kernel, notes that “I usually try to avoid the term ‘hacker.’ In personal conversations with technical people, I would probably call myself a hacker. But lately the term has come to mean something else: underage kids who have nothing better to do than sit around electronically breaking into corporate data centers” (Torvalds 122). However, in exploring programming history, culture, and ethics, writers such as Eric Raymond and Pekka Himanen have recently tried to restore the term to its original dignified implications: for them, it signifies programming skill, technical mastery, and adherence to the programmers’ code of honor known as the “hacker ethic.” I also use the term “hacker” in this dissertation in this way, to skilled programmers for whom the act, and art, of computer programming is imbued with a self-consciously ethical and aesthetic purpose.

II

Although *The Matrix* provides a useful analogical framework for beginning to think through issues in computing culture, any appeal to the film for the truth of programming politics must end with an admission of the glossy, aestheticized vision the film puts forth. The way *The Matrix* reformulates the history of hackers' struggle against proprietary code—turning it into a series of carefully choreographed martial arts battles between a stylish and ethnically diverse band of rebel men and women and a homogeneous group of white male Agents uniformly outfitted in reflective shades and dark business suits—is expressly and literally fantastic. The very act of reading, understanding, and reprogramming code is similarly imbued with science-fiction convention, so much so that the film depicts mere attentive human eyes performing the translation between executing application and underlying source code. If you spend enough time watching glowing hexadecimal characters scrolling on a screen, *The Matrix* suggests, you will naturally acquire, just as a child acquires language, the ability to make sense of what looks at first to be a mass of jumbled signs. In the real world, as subsequent chapters will make clear, acquiring access to the source code that underpins computing applications is a vastly more difficult, more complex, and more politically and economically fraught process. That said, however, *The Matrix's* sleek and stylized portrayal of computing programmers' desperate bid for human freedom makes the film an ideal entry point to a study of computing code's aesthetics. Positioning computer programming as a means of stylized resistance to crude, ugly domination, the film ultimately values a particularly artistic attitude to the work of coding: this attitude is revealed in the film when the rebel hackers deride the Matrix for its ugly glitches or admire their own programming handiwork in the illegitimate and subterfugal additions they make to its structure. Highlighting the “applied aesthetic” that has historically been a foundational element of programming practice, the film ultimately draws parallels between the aesthetic and political trends in computing programming history that will be central to this study. More specifically, *The Matrix* focuses its computer programming aesthetic by picking up on another practice common in computing culture—the tendency to understand writing code as a process akin to writing literature.

Relying heavily on literary and theoretical allusions to interpret for the lay viewer the technological and epistemological struggles it depicts, *The Matrix* figures two books into its structure: Jean Baudrillard's *Simulations* and Lewis Carroll's *Alice's Adventures in Wonderland*. The first appears as a hollow book: when not grinding away in a cubicle at a giant software corporation, Neo runs a black market sideline in illicit software, copies of which he stores inside a box that looks like Baudrillard's text. The second appears as a series of guiding metaphors. The resistance movement first gets Neo's attention by breaking into his home computer and advising him to "follow the white rabbit"; later, when Neo has to decide between joining the resistance or just going home, Morpheus, the movement's leader, explains that living within the Matrix's virtual reality entails self-consciously accepting a view of life as literature: "You take the blue pill and the story ends," he tells Neo; "You wake in your bed and you believe whatever you want to believe . . . You take the red pill and you stay in Wonderland and I show you how deep the rabbit-hole goes." In *The Matrix*, Baudrillard's *Simulations* is itself a simulation, a fake book about fakery in a world so perfectly fabricated that it can readily absorb the very theories that ought to debunk it. Conversely, *The Matrix* sees *Alice's Adventures in Wonderland* not as a story about a dreamed reality, but as a frame of reference for reality itself: when Neo goes down the rabbit-hole, he falls into the truth about the world. That truth, however, is the truth of simulation. One of the first things Morpheus says to Neo when he introduces the young hacker to the simulated nature of Matrix-generated reality is "Welcome to the desert of the real," a quotation from Baudrillard's *Simulations*. The film thus unfurls by way of a double indebtedness to its oddly matched pair of muses, a stammering Victorian mathematician with a penchant for building logical problems into children's dreamscapes, and a postmodernist, postcapitalist French theorist best known for his provocative theses about how Disneyworld, television, and other virtual realities have come to supplant objective reality. The reason for *The Matrix's* dependence on literary and literary critical models quickly becomes clear: at its most basic level, Matrix-generated reality is itself text, a vast web of code that, in its two-dimensional incarnation, can be viewed, read, interpreted, and analyzed. Only a literary sensibility, Morpheus implies, will allow Neo to inhabit, and finally master, that incarcerating textual world; only a literary sensibility will allow him to understand

computing code as a flexible language that can be reshaped and rewritten to the ends of freedom rather than domination.

The Matrix's subtle dependence on literary metaphor draws upon the embedded relationship between computing code and literary aesthetics that will be the focus of this dissertation. *The Art of Code* will argue that aesthetics, specifically *textual* aesthetics, has constituted a foundational element of programming practice and culture ever since the mid-1950s, when compiler technology began to enable translations between higher-level programming languages and machine code. These developments freed computer operators from the laborious, mathematically intensive work of writing computer programs directly in binary or hexadecimal machine code; they also coincided with a shift toward electronic business data processing that created a new class of computing professional: the programmer. No longer reliant upon the expertise of trained mathematicians and engineers, corporations hired workers who demonstrated an aptitude for working with new high-level languages such as FORTRAN or COBOL. Crafting their professional identity around computers, yet lacking the formal mathematical or engineering training considered *de rigeur* in computing's earliest years, these new programmers came to shape their professional identities around their work with high-level languages. From the computing industry's rapid expansion, then, came a burgeoning computing culture with its own traditions, its own politics, and its own finely honed sense of aesthetics. As writers such as Steven Levy, Eric Raymond, Glyn Moody, and Pekka Himanen have repeatedly stressed, computer programmers from the 1950s onward began to regard themselves embodying a radically divided identity: they were artists as much as engineers, bohemians as much as corporate workers. Understanding their programs as carefully crafted texts that are as much for reading as for running, programmers have historically relied heavily on traditional aesthetic ideals of formal elegance, crisp and creative expression, striking originality, local precision, and gorgeous overall effect to guide their work.

This understanding of code as an aesthetic artifact partly explains programmers' longstanding impulse to share their code openly rather than hiding it in proprietary, inaccessible binary formats.³

³Binary code will work on computers but is indecipherable to the human reader. I will explore the technical and political ramifications of binary-only code distribution in Chapter One.

In the foreword to Donald Rosenberg's *Open Source: The Unauthorized White Papers*, Linux programmer Jon Hall writes that he once lacked a concrete understanding of why programmers willingly gave away their software source code; then it struck him that the aesthetics of code itself constituted a powerful motivation—or, as he puts it, “Very few amateur painters put their paintings in a dark closet” (xi). Just as painters exhibit their work to others and benefit from critical commentary on it, programmers improve their skill by offering their code for critique and by studying the programs written by their more accomplished and talented peers. Although Hall draws his analogy from the visual arts, programmers more commonly seek a formal mastery of *language*; in fact, so firm are the associations between programming skill and linguistic elegance that Eric Raymond advises aspiring hackers to work on their language skills as well as their programming skills, encouraging them to develop their facility with puns and wordplay alongside their understanding of programming theory and languages (246). Among masterful programmers, one set of compositional skills begets the other; writing code is to them as syntactically demanding and as potentially inspiring as writing poetry is to a poet. In fact, for many expert programmers, producing line after line of elegant, expressive code is not only *analogous* to writing a perfectly structured sonnet or a flawlessly balanced iambic couplet; programming is itself a poetic undertaking. Throughout this study I will argue that programmers' approach to their work represents a merger of aesthetic sensibilities that challenges conventional distinctions between the work of the programmer and that of the poet.

The Art of Code finds its inspiration at the nexus of literature's and computing culture's related but distinct aesthetic systems. In the chapters that follow, I seek not only to untangle the complicated knot that they form, but also to show how important it is to both cultures to identify the strands within the knot. It will thus be my goal in this dissertation to aestheticize the work of computer programmers in a very different way than does *The Matrix*. I strive here not to impose a trendy postmodern paradigm onto computer programming practice, nor to define a new “cyberaesthetic” for the digital era, but rather to examine from a literary perspective an aesthetic that has always been deeply embedded in computer programming culture and that can be regarded as foundational to it. In exploring this merger of aesthetic sensibilities, this dissertation will study

the emergence of programming aesthetics during the 1950s, will discuss how those aesthetics took on specifically literary connotations in ensuing decades, and will analyze the implications of programming aesthetics for critical and theoretical debates within literary studies. The project has five central goals: first, to show how revolutions in computing architecture and programming methodology have shaped and refined the aesthetics of code; second, to outline a responsible historiography of code (an undertaking that involves challenging how cultural and literary theorists have appropriated code and used it metaphorically in ways that systematically ignore the history, culture, and aesthetics of programming culture); third, to show how an identifiably literary aesthetic emerged as part of programming practice, and to show how that aesthetic came to depend on theories of authorship and models of close reading that, at least until recently, were regarded as essential to the literary critical enterprise; fourth, to show how a literary appreciation of programming aesthetics not only allows for a sophisticated literary theoretical understanding of programming history but advances our understanding of literary form itself (I focus here on the late experimental writings of James Joyce); and fifth, to show how, in developing alternative concepts of authorship, and in making the “beauty” of code into a powerfully motivating political force, programming aesthetics challenge the current tendency among professional literary scholars to regard “the literary” and “the aesthetic” as mere agents of false consciousness, as naive mystifications that shelter dominant ideologies from literary theory’s radical critique.

This study thus aims to supplement, to historicize, and to extend debates on computing technology as they are currently conducted within the humanities, particularly within departments of language and literature. In doing so, it takes a field—computer programming—that has recently fallen under the purview of cultural theory, and resituates it within the realm of traditional literary study. There are compelling methodological reasons for such an undertaking. First, the past decade has seen a series of impassioned and extraordinarily polarized debates about the impact of electronic textualities, digital archives, the Internet, and other computer-mediated phenomena on everything from the future of scholarly work to the very fabric of human identity. I argue that these debates, refracted as they have been through the polarized and hyperbolic rhetoric of the “culture wars,” have failed to import into literary studies a solid understanding of computing or

programming history; as such, this study will attempt to counteract and correct some prevailing distortions that continue to shape the theoretical reception of computers within the humanities. Specifically, I wish to argue that literary scholars have overlooked the deep literary affiliations embedded in the history and art of computer programming; when they do invoke computing code, it serves merely as a convenient synecdoche for all manner of radical and transgressive political and aesthetic agendas. Acting as a counterweight to these approaches, this study insists that theoretical generalizations about code, about the political effects of “digital culture,” and about the ontological nature of “cyberspace” must ultimately be rooted in an understanding of how computers, computer programs, and computing networks actually function. In other words, to understand the progressive technical and cultural abstractions that have been built atop computing hardware and software, one must also understand the underlying mechanisms of those electronic and coded systems and appreciate the technical and cultural histories underpinning their construction. Second, I contend in this study that traditional textual hermeneutics is simply an indispensable tool for understanding programmers’ aesthetic conception of what code is and about what writing code means. As such, a founding conviction of this project is that a nuanced, historically informed, and genuinely interdisciplinary understanding of our evolving technological landscape has been further impoverished by cultural critics’ recent virulent reaction against traditional text-based hermeneutic approaches. However, because programmers often understood their code as an art form, and, more often than not, as a *literary* art form, it is a strange irony of the history that I trace that the text-based hermeneutic strategies so frequently derided by cultural theorists as politically retrograde and theoretically naive simply provide the most progressive and illuminating tools available for articulating the special aesthetic valences of programming practice. Rather than aligning the computer with the emergence of the postmodern condition, an assumption that animates many literary theorists’ interests in computing, I suggest that a responsible theoretical investigation of code’s history and aesthetics brings the computer closer to a *modernist* sensibility than a postmodern one. The trajectory of the dissertation, which begins with the earliest attempts to program digital computers in the 1940s and ends with a comparative study of programming theory and Joyce’s authorial evolution, is designed to reflect that theoretical realignment.

III

The dissertation is divided into three chapters, which work together to provide an aesthetic history of computing culture, an analysis of how the literary idea has structured the history of programming practice, and a related analysis of how an understanding of the computer programmer's special relationship to literary aesthetics can serve to inform the project of modern literary critical analysis. To ground this project theoretically and historically, the dissertation begins by tracing competing, and often conflicting, ways of situating software within a wider cultural framework. This work is necessary because software's cultural history has been almost entirely neglected by academic computing historians, who have focused the bulk of their efforts on computing's mathematical and engineering prehistory and on the early construction of computers before and during World War II; and by computer scientists, whose historical efforts have been devoted primarily to documenting exhaustively the technical intricacies of the many different programming languages that emerged onto the computing scene from the 1940s onward. What we lack at present are comprehensive and synthetic *cultural* histories of code, histories that could provide a sound historiographical basis for the study of code's literary aesthetics. In outlining existing approaches to computing's history, my goal in this chapter is not necessarily to privilege one mode of investigation over another, but to highlight the inadequacies of various approaches and to argue for a more informed, more genuinely interdisciplinary approach to code's convoluted and complex history and ontology. As such, I concentrate on tightly intertwined and constantly shifting historical, technological, philosophical, political, and legal conceptions of what software is and has been, focusing particularly on how those conceptions have shaped our ideas about how software should be written, distributed, and protected. While contributing to the self-contained argument of Chapter One, these sections also provide essential background material for the theses about programming practice that I will explore elsewhere in this study.

My goal in the first sections of Chapter One is to concentrate on the histories of computing architecture, programming methodology, and software economics to outline the complex interconnections among technical innovation, evolving corporate structures, and the cultures that

have grown up around computer programming over the last half century. Beginning with the intense digital computing research conducted during World War II, I focus mostly on architectural and methodological developments in the United States. At the University of Pennsylvania's Moore School of Engineering in the mid-1940s, a team of engineers and mathematicians collaborated to build one of the first large-scale programmable digital computers, a machine known as ENIAC. My focus here is less on the physical construction of early computers (this history has been documented extensively by computing historians) than on the conceptual architecture developed in parallel with ENIAC that has come to define modern computing, a theoretical model commonly known today as the "von Neumann" or "stored program" architecture. Showing how computer programming history can be understood as a history of progressive abstraction from both computing hardware and the foundational binary code that computers natively "understand," the first section of the chapter explores both how this stored program architecture enabled computing hardware and software to become distinctly separate entities and how it allowed the creation of the high-level programming languages that divided computer programs according to their closeness to their human author (high-level source code) or the computers that run them (low-level binary code). In short, I argue that the architectural and methodological revolutions of the 1940s and 1950s ultimately abstracted *code* into *language*, and that that crucial abstraction both shaped the identity of programmers (as writers) and set the stage for future political battles over defining code and defining the nature of software authorship.

The layered relationships among high-level source code, low-level binary code, and computing hardware, coupled with the software sharing traditions that arose among programmers and corporations during the 1950s, enabled rapid early advances in computer programming theory and practice. In turn, these early decades saw computer programmers, members of a new professional discipline nestled uncomfortably between the authoritative, established fields of mathematics and electrical engineering, establish strong internal cultural traditions that came to define the practice and craft of writing software. As programmers differentiated themselves from mathematicians and engineers and came to understand themselves as writers, as skilled manipulators of language and syntax, the uninhibited, unregulated exchange of code became enshrined as a core tenet of

their grassroots ethic. This ethic ensured a high degree of openness between programmers and corporations; in turn, this spirit of co-operative code sharing helped the nascent programming profession to grow and flourish at an extraordinarily rapid rate. This was one way in which the architecture of computing hardware and software helped shape the operative aesthetic models and political conditions under which early programmers worked. However, alternative economic and political ramifications of the von Neumann architecture emerged in the 1970s to threaten the libertarian traditions developed within early programming culture. I discuss the corporate and economic background to that shift, describing how computer programs, understood until the mid-1960s as “value-added” enhancements to expensive and fragile hardware installations, became commercial *products* in their own right as hardware dropped radically in cost during the late 1960s and 1970s. As the economic status of software changed, it became clear that stored program hardware architectures could enforce secrecy and closure just as easily as they had previously enabled openness and cooperation. The chapter analyzes how the establishment of a defined software industry in the 1970s led corporations to exploit the economic potential of the stored-program architecture, using that architecture’s enabling internal division of programs into source and object code to protect their products from unauthorized duplication, appropriation, and alteration. This process, called “blackboxing,” was an affront to the traditions of free information exchange established over the first three decades of programming culture, and some programmers have struggled ever since to keep code free and open, even as corporations and courts have invoked technological methods and legal mechanisms designed to protect source code from disclosure and free exchange. I discuss some programming movements that arose in response to this corporate and legal clampdown (and to the profit motives driving that clampdown), and outline the various pragmatic, ethical, and aesthetic rationales that underpin their actions.

The chapter then considers an alternative approach to computing code’s cultural history, discussing the theoretical, ideological work that the computer performs in contemporary literary and cultural studies. In this section I argue that while programmers and computing scientists have imposed layers of *theoretical* abstraction upon computing hardware to aid their work in programming and controlling computers, cultural theorists have imposed an additional layer

of *metaphysical* abstraction upon the computer, one that works to advance explicit ideological agendas. “Cyberspace,” as this abstraction has become known, serves as a consensual utopian geography where cultural studies’ enduring political preoccupations (including, but not limited to, race, class, gender, power, and embodiment) can find a more or less ideal resolution. Reading a series of choice cybertheoretical texts, I discuss how code circulates in the “integrated circuit” (the phrase is Donna Haraway’s) of cultural theory as a metaphoric means of theorizing the future of subjectivity, race, gender, society, and even reality itself. I argue that using the computer as a platform for this metaphysical and political utopia not only displaces engagement with the real conditions surrounding the writing, refining, circulating, and even marketing of software at the turn of the century, but works against a responsible scholarly analysis of code’s literary aesthetics. My aim will be to preserve the ideal of more closely integrating computing and literary histories while differentiating the fantasies propounded by cybertheory from the realities cybertheory ignores. Turning to the writings of legal theorist Lawrence Lessig, I argue for models of computing abstraction that are more historically, ideologically, and aesthetically responsible. Lessig’s investigations into code’s political and technological architectures will allow me to argue for a “layered” model of cyberspace that respects the history outlined in the chapter’s earlier sections while opening the possibility for a developed literary aesthetics of code.

Drawing upon the historiographical and methodological framework outlined in Chapter One, my second chapter develops the aesthetic history of code further, examining the importance of the literary ideal to programmers and to programming culture. The chapter first discusses the specifically literary turn within programming history and culture, showing how literary philosophy became an influential element of programming theory (for instance, the preeminent computer scientist Donald Knuth, whose magisterial *Art of Computer Programming* is a definitive reference for programmers, has frequently stated his belief that programming is as much a literary art as it is a mathematical or engineering process). The chapter’s second section studies an area where the literary status of code is hotly contested: the ongoing legal debates about the applicability of patent and copyright law to computing code. At the core of this debate, one finds a contentious ongoing argument about what code *is*, about how it functions, and about whether writing code

constitutes an act of engineering or a mode of creative expression. In one sense, code is a utilitarian component in a machine; code exists so that computers can perform work. But code, because of the complex abstractions enabled by the von Neumann architecture, is also a *textual* entity, written by programmers in defined high-level languages. Confounding traditional distinctions between the expressive and the utilitarian, the creative and the functional, code ultimately challenges the very metaphysical basis of intellectual property law.

The chapter's first two sections propose that the complex ontological blurring that emerges from these legal debates signals the need for literary critics to recognize and understand the literary sensibility that has helped shape programming history. The chapter's remaining sections explore that literary sensibility by analyzing three central episodes in the history of code as a literary, artistic artifact. First, I will demonstrate how a "literary" practice of close reading code has been central to shaping programming culture and practice. From Ken Thompson, co-creator of UNIX, who formed a UNIX "reading group" at the University of California, Berkeley, in 1975, to John Lions, an Australian computer science professor whose 1977 annotated critical edition of Unix Version 6 became a textbook for his programming course, to Daniel Bovet and Marco Cesati, who recently published a book encouraging people to close read the Linux kernel's source code, programmers have built strong literary and literary critical values into their notions of teaching, learning, and sharing the art and science of computer code. I further this discussion by examining a second central episode in code's literary history: Donald Knuth's "literate programming" system of computer programming. Explicitly exhorting programmers to understand writing programs as a new way of writing literature, Knuth designed his literate programming method so that programmers could closely integrate the *compilation* of source code with its *publication*. In a literate programming system, a programmer/author's completed source code becomes both an executable binary intended for use in a computer system and an annotated critical edition intended for traditional publication on paper. To this end, Knuth included in the literate programming system his T_EX digital typography program, a piece of software that incorporates not only Knuth's ideas about the aesthetics and philosophy of programming but his exhaustive, meticulous research into the history of print culture. The orientation of literate programming, then, is to encourage

programmers to become authors, to see as the final product of their effort not only a working program but a beautiful published edition of their writing. Moving on from literate programming, the chapter's final section focuses on "Perl poetry," a poetic genre created by Perl programmers in the early 1990s. Combining close attention to poetic form with equally close attention to the parameters of the Perl programming language, Perl poets such as Sharon Hopkins strive to produce texts that are at one and the same time literary artifacts and working programs. In their close integration of coding practice and poetics, literate programming and Perl poetry exemplify and celebrate the "art of code."

Building on these discussions of code's literary dimensions, Chapter Three examines a series of telling intersections between modern programming theory and the models of literary authorship employed by James Joyce. The chapter begins by outlining the engineering metaphors that Joyce frequently used to describe his late authorial practices and by showing how critics' manipulation and extension of those metaphors has culminated in a critical vision of Joyce as a "software engineer." In an attempt to concretize those engineering and computing metaphors and show how they can work as more than vague analogies, the chapter suggests that deep structural affinities exist between Joyce's late authorial processes and a sophisticated, complex paradigm known as object-oriented programming that has dominated programming theory for the past two decades. My argument in this chapter is that an understanding of computer programming as a literary technique, a mode of writing with inherent aesthetic capabilities, enables a powerful re-imagining of the complex linguistic and structural experiments Joyce conducts in *Finnegans Wake*. Specifically, I want to suggest that contemporary programming theory can enable us to build upon and revise the structural models of *Finnegans Wake* developed by Joyceans such as Clive Hart, A. Walton Litz, and David Hayman. Although the chapter initially proceeds in an analogical, formalist manner, it ends by historicizing the connections between Joyce and object-oriented programming. Finally, tracing the roots of the object-oriented paradigm to modernist theories of child psychology, the chapter concludes by studying how Joyce's conceptualization of the *Wake* intersects with his lifelong deliberations about how children think and learn. The purpose of historicizing these theoretical claims about Joyce is to argue for a conception of computer

programming that is deeply indebted to modernist ideals of authorship and, in turn, to propose the need for further historical and theoretical work situating modern computer programming in relation to the modernist tradition.

The conclusion to *The Art of Code* draws together many of the dissertation's concerns in a reconsideration of Martin Heidegger's technological aesthetics. This reconsideration responds to post-Heideggerian theorists of technology such as Hubert Dreyfus and Albert Borgmann, for whom Heidegger's technological speculations (in "The Question Concerning Technology" and other texts) have licensed a profoundly reactionary aesthetic stance toward modern computing systems. Although Dreyfus, Borgmann, and others envision technology's detrimental impact on core societal values (Borgmann's recent *Holding on to Reality* reinforces the idea that computing technology has set us on a Matrix-like course toward ontological oblivion), I argue in the conclusion that Heidegger is concerned less with technology *per se* than with the impact of technology upon traditional aesthetic values. What the rational, mechanistic twentieth century lacks, per Heidegger's philosophical diagnosis, is a sense of historical and ontological groundedness—hence Heidegger both romanticizes a Rhine valley peasant lifestyle that he sees as continuous with Ancient Greek ontology and valorizes aesthetic celebrations that understand language itself as part of a larger philosophical return to origins. Extending Heidegger's aesthetic analysis to the models of computer programming practice explored within the body of the dissertation, the conclusion argues that computer programming practice lends itself to Heideggerian analysis as an area where *poiēsis* self-consciously "presences forth" from within technology; programmers' approaches to their work thus question the readings of Heidegger that underlie pessimistic speculations such as Borgmann's. While the conclusion cannot do justice to the vast body of philosophical work on the nature of technology, it does suggest the need to situate Heidegger much more carefully in relation to technological aesthetics, particularly in relation to the literary poetics of programming with which this dissertation is concerned.

In the last analysis, *The Art of Code* offers a comparative history of recent developments in literary and computing cultures. Recent decades have seen the explicit politicization of both literary critical and computing cultures, although their respective politicizations have taken effect

in very different ways and for very different reasons. The first has demonstrated its political commitments by all but abandoning literature and aesthetics as its subject matter (witness the rise of new historicism, cultural studies, cultural materialism, body criticism, and a related assortment of expressly political methodologies that increasingly disregard literary subject matter) and even by demonstrating its loss of faith in aesthetics through its reification of ugliness at the level of critical style and choice of subject matter—with the goal of transgressing and destabilizing the literary canon for political effect, cultural theorists now turn with ready alacrity to subjects such as madness, torture, pain, monstrosity, pornography, and disease. Computing culture, on the other hand, has *adopted* a traditional model of literary aesthetics as a means of effecting change, finding political utility and social value in the well-crafted product that is at once entirely usable and wholly beautiful to contemplate. The distinctions are clearly evident in the respective disciplines' discourses: whereas terms such as “elegant” and “beautiful” circulate freely in computer culture to describe well-crafted code, elegance, beauty and all their synonyms have been effectively exiled from the vocabulary of literary and cultural theory, where their use now signifies only the presence of theoretical naivete and ideological mystification. The point of the comparison is threefold: first, to question literary criticism's deliberate equation of aesthetics and of “traditional” literary value with irresponsibly apolitical, even reactionary, theoretical stances; second, to reclaim the traditional close-reading methodologies of literary scholarship as powerful tools for technological analysis and political critique; and third, to show how necessary computing knowledge is to a responsible literary and cultural studies—both as part of textual transmission and cultural formation and as an increasingly important part of literary history. *The Art of Code* is both frankly polemical and openly hopeful: the hope of the polemic is to forge productive, informed bridges between the semantics of computer programming and of literature, and to open new ground for debate on the complex, dynamic interrelationship between politics and aesthetics. Because the aim of the dissertation is finally to open new fields of inquiry, it is often more suggestive than definitive, covering more territory rather than less so that I can ask broader and ultimately more pressing questions about the history and future of literary and technological aesthetics.

Chapter 1

Crossed Wires: Historiographies of Code

For all its enigmatically opaque symbolism, Stanley Kubrick's *2001: A Space Odyssey* (1968) reflects an ideal of clarity that was dear to the heart of 1960s programming philosophy: that humans and digital computers could one day interact through the transparent medium of ordinary conversational English. Instead of laboriously entering arcane commands via a teletype terminal, the standard human-computer interface at the time the film was made, Kubrick's astronauts communicate with their mainframe computer HAL as if HAL were another human being on board: speaking to the computer, they conduct maintenance checks with it, even play chess against it. Moreover, Kubrick extends HAL's perceptive powers beyond mere speech comprehension: when the astronauts attempt to talk privately in a space pod about HAL's dangerously literal interpretation of its programmed instructions, the computer "overhears" their conversation by reading their lips. A film in which computers are capable of comprehending speech—and even of lipreading—looks to be a fantastic, if potentially Orwellian, vision for a far-distant future; nonetheless, dismissing Kubrick as a technophobic reactionary would ignore the complexity and sophistication of his response to computing technology. While Kubrick shares with contemporary figures such as Hubert Dreyfus a profound distrust of artificially generated intelligences, *2001: A Space Odyssey* also reveals his deep interest in the debates, pursuits, and ambitions of 1950s and 1960s computing culture and theory.

On the surface, Kubrick's film feeds a paranoid technophobic vision of intelligent machinery run amok: the film derives much of its suspense and terror from the combination of HAL's ability to understand spoken instructions and its programmed inclination to interpret those instructions absolutely literally. Emphasizing the destructive capacity of a pure, logical intelligence untempered by emotional or ethical understanding, Kubrick's *2001* was received within technological circles and the mass media alike as a searing critique of early Artificial Intelligence research. Yet the film also appealed to computer programmers and theorists by exemplifying a vision that by 1968 had long animated their work: to bring human-computer interaction into the realm of language, and ultimately to develop a means of communicating with computers that would be as linguistically transparent as human speech. That vision's animating ideal was, and remains, that of progressive abstraction from lower-level machinic and computational processes to higher-level, more human

faculties. Just as humans can conduct conversations without necessarily understanding the complex lower-level neurobiological and auditory mechanisms that make speech possible, early computing researchers sought to make the work of telling computers what to do accessible and easy by distancing the method of programming from the internal workings of the computing hardware. Kubrick explicitly acknowledges the mechanisms of such abstraction in the film's famous climactic scene: in order to disarm the megalomaniacal HAL, astronaut Dave Bowman tears out the memory modules that contain the computer's programming instructions. As the astronaut removes more and more of HAL's code, the computer "regresses" through ever more infantile stages, singing "Daisy, daisy . . ." and gurgling like a baby just before it finally shuts down. Kubrick's representation is striking: as the astronaut strips away the higher faculties conferred through HAL's software, the computer undergoes a developmental and linguistic devolution that reflects its reversion to pure mechanism. Without its programmed capacity for the mechanical equivalent of abstract reasoning, HAL can exist only as a harmless, "lifeless" chunk of hardware, just like any standard IBM mainframe of the era.¹ In dramatizing a pseudo-psychological reversal of software's developmental history, Kubrick's film captures the essence of that history: over time, the process of programming digital computers has grown ever more abstract (in its relation to the machine) while at the same time growing ever more transparent (in its relation to the human beings who write and read programs for those machines). More to the point, the history of software's gradual abstraction from hardware is also the history of programming's language acquisition: as HAL understood English, so computers since the 1950s have "understood" a variety of programming languages, including FORTRAN, COBOL, and C, each of which allows a programmer to give instructions to a computer without resorting to the difficult, complex machine code that computers can directly interpret.

I will be concerned throughout this chapter to articulate some of the major cultural, political, and aesthetic effects of machine code's abstraction into ever more complex and capable programming languages. First, I examine how abstracting code from hardware made it possible for programmers to conceptualize their work as a new kind of *writing* rather than as an esoteric

¹Indeed, as many commentators have noted, the name *HAL* makes a thinly veiled allusion to IBM, the letters *H*, *A*, and *L* being those that respectively precede *I*, *B*, and *M* in the alphabet.

form of mathematics; as I will show in this chapter and in the next, this emphasis on writing has been crucial both in shaping programming sensibilities and in imbuing the act of computer programming with an identifiably literary aesthetic. Second, I show how the abstractions afforded by high-level computing languages enabled programmers to write portable programs that could be shared across computers and among computing architectures; this “share-ability” of software has had important cultural and political effects, the many valences of which I will trace throughout this chapter. Third, I show how the advent of high-level programming languages paradoxically created the potential for the very obverse of code sharing: they allowed programmers and corporations to “blackbox” their code, to prevent users from seeing programs in their high-level (i.e., most legible) form by distributing them only in their simplest, most inaccessible (i.e., least abstract) binary formats. Over the past sixty years, these three phenomena have interacted in different, but always decisive, ways to shape the field of software development, the cultures that have grown up around it, and the ideals of authorship that drive it. Indeed, it is hardly an exaggeration to suggest that software’s entire cultural, corporate, and legal history may be understood in terms of the interrelated effects of code’s abstraction into language.

This chapter traces the principal trajectory of those effects, locating its roots in the earliest days of digital computing history, following it through the dark days of corporate and legal restrictions on software distribution, then watching it find its current incarnation in the conflict between blackboxing and software sharing that characterizes our present computing era—an era in which powerful corporate and legal enforcement confronts the vibrant and colorful, if comparatively tiny, “open source” and “free software” movements. As such, this chapter reflects one of the driving methodological convictions of this dissertation: that any responsible analysis of computing culture’s politics and social impact must be anchored both in a material understanding of how computers actually work and in an historical understanding of where computing code came from and how it has evolved over time.

Trying to arrive at such an understanding is itself a radically interdisciplinary undertaking: one can find the cultural history of code articulated in a variety of academic contexts—including computing history, the history and sociology of science, cultural studies (especially in the area

of cultural studies known as cybertheory), legal studies, and computing science—as well as in non-academic contexts such as the growing published literature on open source and free software, and in computer programmers’ efforts to document their own history and culture. As will become clear in this chapter and in the next, the history and even the ontology of computing code is a hotly contested and intensely disputed field of inquiry. Furthermore, vehement disputes over the definition and history of computing code take place against a backdrop of academic neglect: computing historians have shied almost entirely away from software as an object of study, focusing their efforts instead on computing’s mathematical and engineering prehistory and on the early construction of computers before and during World War II. Work on the history of the computer after 1970 is rare; the cultural history of programming, with which I will be primarily concerned in this study, has been almost entirely neglected.² In a 2002 article, computing historian Michael Mahoney bemoans the present sad state of software history:

We have practically no historical accounts of how, starting in the early 1950s, government, business, and industry put their operations on the computer. Aside from a few studies with a primarily sociological focus in the 1970s, programming as a new technical activity and programmers as a new labor force have received no historical attention. Except for very recent studies of the origins and development of the Internet, we have no substantial histories of the word processor, the spreadsheet, communications, or the other software on which the personal computer industry and some of the nation’s largest personal fortunes rest. (“Software: The Self-Programming Machine” 92)

In his 1998 *History of Modern Computing*, Paul Ceruzzi observes how existing software histories have tended to assume one of two distinct shapes (9). The first type of history charts the rise of individual software companies—such as IBM, Microsoft, Oracle, or Apple—and situates those companies’ software products within a corporate and marketing framework.³ Written largely for a

²An important and timely study of programming’s emergence as a professional discipline is Nathan L. Ensmenger’s 2001 dissertation *From “Black Art” to Industrial Discipline: The Software Crisis and the Management of Programmers*. Martin Campbell-Kelly and William Aspray’s *Computer: A History of the Information Machine* is another important study that carefully situates programming methodology within the context of an emerging software industry—see especially chapters 8 and 11.

³See, for example, Daniel Ichbiah and Susan Knepper’s *The Making of Microsoft*, James Wallace and Jim Erickson’s *Hard Drive: Bill Gates and the Making of the Microsoft Empire*, Stephen Manes’s and Paul Andrews’s *Gates: How Microsoft’s Mogul Reinvented an Industry—and*

general audience, these histories tend to ignore the technical aspects of software development, focusing instead on how particular software products have contributed to the reputations of particular companies, or on how they enable computer users to think and work in different ways. The second type of software history noted by Ceruzzi charts the rise of individual programming languages such as FORTRAN, COBOL, and Simula, and is primarily concerned with how each language worked at a particular moment to enable programmers to extract different kinds of functionality from the computer's hardware. Computer scientists and historians from the 1960s onward have expended a great deal of effort documenting the technical history of computer programming languages; we now have articles and conference proceedings that enumerate exhaustively the formal intricacies of the many different languages constructed from the 1940s onward.⁴ Studies written in this mode are often highly formalist, focusing more on the technical

Made Himself the Richest Man in America, John Scully's *Odyssey: Pepsi to Apple ... A Journey of Adventures, Ideas, and the Future*, and Steven Levy's *Insanely Great: The Life and Times of Macintosh, the Computer that Changed Everything*.

⁴Donald Knuth and Luis Trabb Pardo give a comprehensive survey of the earliest languages in their paper "The Early Development of Programming Languages." Tracing developments in programming methodology during the first decade of digital computing's history, Knuth and Pardo offer a useful prehistory to the study of high-level programming languages. Methods studied include Zuse's Plankalkül (1945), Goldstine and von Neumann's Flow Diagrams (1946), Curry's Composition (1948), Mauchly *et. al.*'s Short Code (1949), Burks's Intermediate PL (1950), Rutishauser's Klammerausdrücke (1951), Böhm's Formules (1951), Glennie's AUTOCODE (1952), Hopper *et. al.*'s A-2 (1953), Laning and Zierler's Algebraic Interpreter (1953), Backus *et. al.*'s FORTRAN (1954–57), Brooker's Mark I AUTOCODE (1954), Ershov's III (1955), Kamynin and Liubimskii's III-2 (1955), Germs and Porter's BACAIC, Elsworth *et. al.*'s Kompiler 2 (1955), Blum's ADES (1956), Perlis *et. al.*'s IT (1956), and Katz *et. al.*'s MATH-MATIC (1956–58). Usefully, Knuth and Pardo give a practical elucidation of these early programming systems by coding a sample algorithm in each language they study. Saul Rosen's 1967 collection *Programming Systems and Languages* attempts to trace a history of programming languages up to that point, while Jean Sammet's detailed 1969 study *Programming Languages: History and Fundamentals* gives technical descriptions of the many major languages then in use. Wenger's 1975 article "Programming Languages—The First 25 Years" offers a useful survey of programming developments until the mid-1970s, as does Allen Tucker's 1977 book *Programming Languages*. Richard Wexelblat's monumental 1981 collection *History of Programming Languages* collects the proceedings of the 1978 ACM SIGPLAN History of Programming Languages Conference. Reprinting detailed contemporary discussion sessions on the languages FORTRAN, ALGOL, LISP, COBOL, APT, JOVIAL, GPSS, SIMULA, JOSS, BASIC, PL/I, SNOBOL, and APL, the volume gives a comprehensive overview of the methodological schools of thought that governed language development through programming's first three decades. A more recent volume in the

intricacies of language development than on the economic and cultural contexts where those languages were used. As such, neither type of history draws substantive links among economic contexts, corporate histories, programming methodologies, and the evolving technology of the extraordinarily dynamic computing field. Nor do they address how the vital, intensely creative culture of programmers was essential both to the evolution of complex programming languages and to the growing success of the software industry.

In the years since Ceruzzi's book was published, a new kind of software history has started to emerge, one that begins to address at least some of these omissions. This new type of history charts the rise of the free software and open source software movements that arose during the 1980s and 1990s, relating these movements to the history and traditions of hacker programming culture that originated mostly in academic computing centers during the late 1950s and 1960s. As such, this strand of history begins to address how economic and ideological factors have shaped the cultural and authorial traditions of programming and how those traditions in turn have affected and even determined what kinds of software get written. Even so, this new brand of software history often largely fails to place itself firmly within the longer evolutionary history of computing technology and software development. Although several books stand out as noteworthy in their attempts to collate the dispersed history and lore of hacker programming culture,⁵ many have a strictly circumscribed focus on 1990s open source projects such as the Linux operating system kernel, the

same vein is Thomas Bergin and Richard Gibon's 1996 edited collection *History of Programming Languages—II*.

⁵An admirable early attempt is Steven Levy's 1984 study *Hackers: Heroes of the Computer Revolution*. A more recent example focused around the rise of Linux culture is Glyn Moody's 2001 book *Rebel Code: The Inside Story of Linux and the Open Source Revolution*. Pekka Himanen's 2001 study *The Hacker Ethic and the Spirit of the Information Age* takes a wider historical and philosophical view, showing how the creative playfulness cultivated by computer programmers challenges rigid and punitive concepts of work developed under the Protestant ethic. The essays collected in Di Bona *et. al.*'s *Open Sources: Voices from the Open Source Revolution* provide a variety of perspectives on the 1990s resurgence of hacker culture, by contributors who include Larry Wall, Bruce Perens, Tim O'Reilly, Michael Tiemann, and Richard Stallman. Eric Raymond, the self-proclaimed "anthropologist" of hacker programming culture, has collected many computer-related writings in his 1999 book *The Cathedral & the Bazaar*. More of Raymond's work, including a work-in-progress on Unix hacker culture, may be found on his comprehensive Web site: see <http://www.tuxedo.org/~esr/>.

Apache web server, or the Perl programming language. As a rule, histories of free software and the open source movement also have a deceptively circumscribed political vision, overlooking the temporal complexity of computing history in order to cast the recent iconoclastic programming movements as straightforward protests against the powerful software empires that arose during the 1980s and 1990s.

While existing software histories all cover important territory, then, they also collectively fail to integrate the different stories they tell. Indeed, as Mahoney indicates, a truly synthetic history of digital computing—one that links economic, political, cultural, corporate, and military histories to the development of computing architectures, software engineering methodologies, and programming cultures—has yet to be written. My aim in this chapter is not to attempt such a comprehensive synthesis, but rather to prepare the way for the particular undertaking of this dissertation, firstly by gesturing toward the kinds of insights that can arise from a more textured and nuanced approach to programming history than has yet been the norm, and secondly by challenging distorted and ahistorical appropriations of that history in the cultural studies subgenre of cybertheory. This chapter thus traces a strategically partial history of software from the 1940s to the present, paying particular attention to how changes in hardware architecture and programming methodology altered what it meant to write and share software; to the cultures that grew up around computers as those changes took effect; to how government and corporate interests responded differently at different times to the values of those cultures, particularly to their adoption of bohemian literary sensibilities and to their libertarian tendency to oppose restrictions upon the free circulation of code; to how software in the late 1960s became a product traded on an open, competitive market; to how some programmers have managed, in the face of great corporate and legal pressure, to maintain the traditions of software sharing established in computing's early days; and to how programmers, corporations, and lawyers have been divided ever since about the legal status of code, the ethics of "blackboxing" code, and about what it means for software to be "free." In tracing the contours of this complex and multi-layered history, these sections lay the groundwork for the main project of this dissertation: to identify, explicate, and analyze how certain movements within programming culture have framed their relationship to code as a

relationship with a shared corpus of elegantly written, freely shared, historically resonant literary texts. In particular, this chapter's analysis of software's abstraction into language prepares the way for Chapter Two, which traces in detail how programmers came to understand their work less as an exercise in clever manipulation of machine code than as a writerly, linguistic endeavor; and for Chapter Three, which argues for a reciprocal relationship between programming and literary theory by showing how programming theory can constitute a useful analytical tool for interpreting late modernist textual experimentation.

Lastly, in their emphasis on grounded, technologically informed analysis, these sections serve as a point of contrast to "cybertheoretical" or "technocultural" discourse, which uses the computer to anchor an entirely different, largely unearned set of theoretical claims. No effort to articulate a historically and materially responsible literary aesthetics of code would be complete without a commentary on literary and cultural theory's attempts to theorize the computer. As such, the chapter ends by analyzing cultural studies' theories of the computer age, theories that (as commentators such as Robert Markley and Richard Coyne have noted) rely on a metaphysics that is nominally extrapolated from, yet remains almost entirely ignorant of, computing history and culture. Showing how cultural studies' theories about the computer—which tend to center on virtual realities, cyborgian (dis)embodiment, "posthuman" subjectivity, and transgressively nonlinear textualities—are neither embedded in nor derived from an awareness or understanding of the technology about which they so lavishly fantasize, I will argue that these theories' literally fantastic nature, what Richard Coyne calls their "technoromanticism," has at once materially shaped humanist scholars' perception of computing and drastically limited the scope and utility of that perception. Because these fantasies are at once so dominant and so uninformed, they largely blind literary scholars both to the complex historiographical and political issues raised by and within computing history and to the aesthetic importance computer programming theory and practice has for our understanding of literary history. Because my larger goal in this study is to outline a historically and philosophically responsible analysis of the conjunction between computer programming and literary aesthetics, the aim of this critique is not merely to expose the sloppy philosophical and methodological models that underpin the vast majority of cybertheoretical

analyses. In exposing cybertheory's inadequacies, I argue that we need a new way to think computing history within literary studies, one that refuses to treat the computer or computing code as a blank slate on which to build utopian fantasies about identity, embodiment, and textuality, one that instead situates itself in a truly interdisciplinary and informed way among the various fields and philosophies that computing integrates. It is partly because computing history has been alternately ignored and mishandled by cultural theorists that I include so much corrective detail in this study about the complex cultural and aesthetic history that has accumulated around the computer in the past half-century.

1.1 From Code to Language

I begin this section by looking at a theoretical eclipse, a small but telling moment of conceptual maneuvering that marks one cultural critic's move to historicize and aestheticize the computer. In "A Tale of Two Aesthetics," the opening chapter of her 1995 book *Life on the Screen*, Sherry Turkle maps an aesthetic schema onto computing history, dividing computing systems into two distinct types, modernist and postmodernist. In Turkle's labels and in her eagerness to apply them to different kinds of computing systems, one discerns the kind of territorial mapping endemic to 1990s cultural criticism, whereby cultural theorists routinely classified texts and authors along theoretical and ideological lines, often dividing them into "traditional" and "radical" camps. In a polarized schema where "traditional" serves as a code word for unenlightened conservatism and where "radical" indicates putatively progressive dissections of Western aesthetic, philosophical, cultural, and economic values, modernism itself, despite its own historical dependence on restless and inventive formal experimentation, became a suspicious category, aligned at best with a politically suspect nostalgia for literary tradition, aligned at worst with the fascist politics of Nazi Germany. Postmodernism, as the term suggests, became modernism's successor, promising to resolve modernism's political complications by replacing the earlier movement's emphases on tradition and aesthetic value with its own particular brand of subversive, ahistorical surface playfulness. Turkle's aesthetic schema, then, is far from politically neutral: importing the

political associations of modernism and postmodernism into a categorical schema that purports to historicize mid-1990s attitudes toward the personal computer, Turkle ultimately gives us a categorization that reveals as much about the ideological stances of cultural theory as it elides about computing history.

According to Turkle's schema, a computing system organized around a "modernist" aesthetic is one that requires its user to "figur[e] out the hierarchy of underlying structure and rules" (35); by contrast, a "postmodernist" interface design "encourag[es] users to stay at a surface level of visual representation" (34). Turkle's "tale of two aesthetics" takes shape around two complementary axes: the contrast between command-line interfaces and graphical user interfaces, as represented respectively by Microsoft's MS-DOS and Apple's Macintosh OS; and the interlocking contrast between users of these systems, who, she alleges, adhere respectively to modernist and postmodernist models of subjectivity. Expanding on the surface/depth model that has long dominated discussions of the postmodern condition, Turkle deploys spatial metaphors to describe how these modern and postmodern users interact with their machines. Arguing that MS-DOS gives its users "access to the guts of the operating system," Turkle quotes an accountant who cites discomfort with the play of surfaces as his reason for scorning the graphical, icon-based Macintosh interface: "[On the Macintosh] you are just dealing with representations," he claims; "I like the thing itself. I want to get my hands dirty" (39); she also cites a physicist who "enjoys the feeling of virtual dirt on his hands and feels threatened by opaque objects that are not of his own devising" (39). Modernist computing subjectivities, then, are characterized by a distrust of seemingly transparent but actually opaque representations and a quest for meaning and clarity beneath their deceptive surfaces. In the postmodern Macintosh world, by contrast, "the user [is] presented with a scintillating surface on which to float, skim, and play. There [is] nowhere visible to dive" (34). Turkle's luxuriant language extends these contrasting subjectivities into contrasting models of erotic pleasure: the physical enjoyment the Macintosh lover takes in sensualized surfaces, in floating, skimming, playing without a need to "dive" into the computer contrasts with the MS-DOS user's compulsion to "get inside the machine," his penetrative need to access

the command line.⁶ For Turkle, the competing pulls of modernist and postmodernist aesthetics and erotics form the basis of a historical split within personal computing culture: “[B]y the late 1980s, the culture of personal computing found itself becoming practically two cultures, divided by allegiance to computing systems. There was IBM reductionism vs. Macintosh simulation and surface: an icon of the modernist technological utopia vs. an icon of postmodern reverie” (36). In Turkle’s vision, computing history is almost perfectly analogous to the history of postmodernism itself: as we live today in a Baudrillardian world of simulated surfaces, so do we compute.

Such a formulation is not unique to Turkle: a significant precursor to *Life on the Screen’s* technological divisions is Robert Pirsig’s *Zen and the Art of Motorcycle Maintenance* (1976), whose protagonist anticipates Turkle’s spatial aesthetic categorizations when he differentiates between “classic” and “romantic” approaches to the technological world. Pirsig, too, divides adherents of these approaches according to elemental subjective distinctions: technological classicists are those who appreciate the “tremendous richness of underlying form”; technological romantics understand technology “primarily in terms of immediate experience” (76). As with Turkle’s modernist and postmodernist models of computing, Pirsig’s distinctions themselves form the basis for an entire history and philosophy: Pirsig’s narrator, himself a compulsive technological classicist, uses motorcycle maintenance to ground an elaborate philosophical system, tinkering obsessively with his motorcycle as a way to fine-tune his evolving ideas about Western metaphysics. In the process, he grows endlessly irritated with his friend John, whose trenchant technological romanticism makes him refuse even to glance under the hood of his bike, even makes him throw up his hands in despair whenever he encounters the slightest mechanical problem. For the narrator, John’s adamant unwillingness to learn how his motorcycle works signifies John’s lack of engagement with everything but appearances and surfaces.

⁶Turkle explains that her analysis holds true for Microsoft Windows, too, since “unlike the Macintosh operating system, [Windows] is only a program that runs on top of the MS-DOS operating system” (38). This qualification—while correct when Turkle’s book was published—does not hold true for versions of Windows built atop Microsoft’s “NT” core; this include Windows NT, Windows 2000, and Windows XP. In addition, Apple has since issued a revamped and modernized version of its Macintosh operating system built upon a BSD Unix core. Released in April 2001, “Mac OS X” features a Unix command-line interface that now allows Macintosh users access to the “depth” Turkle associates with MS-DOS.

Turkle's surface/depth distinctions work in similarly divisive ways: while MS-DOS users spend their computing time "inside" the machine, getting their hands dirty amid its "guts," Macintosh users float on the computer's "surface," avoiding contamination by the virtual "dirt" that lies beneath their interface. These metaphors both introduce and ground *Life on the Screen*, a title whose very preposition gives an intriguing insight into the book's focus: its basic premise is that in a matter of a very few years, everything will be mediated by the computer, even human emotion, sensation, sex, love, and desire—in other words, even those things that we understand as most human, most separate from machinery, most intrinsically what and who we are. Turkle thus sets out to analyze subjectivities that indulge in an endless play of surfaces, that use the "scintillating surface" of the computer screen to mediate their entire lives. Unlike Pirsig, however, Turkle does not interrogate the founding presuppositions of her aesthetic models. Faced with a complex and deceptively singular machine, the computer, Turkle simplifies her object of analysis into stock categories. Faced with a relatively simple machine, the motorcycle, Pirsig problematizes it, subjecting to scrupulous philosophical scrutiny the classic/romantic duality upon which the technological world ostensibly relies, taking his motorcycle to pieces as he dismantles the philosophical foundation upon which his perception of it rests. Turkle, by contrast, neglects to historicize the divide she posits, failing, for example, to recognize that the MS-DOS command-line interface—which arrived on the computing scene in 1981—is itself a "scintillating surface," the result of three decades' worth of accumulated abstraction from the real "guts" of computing hardware. For all its seeming mechanical roughness, for all its apparent proximity to the guts of the machine, the MS-DOS interface nonetheless constitutes a sophisticated "representation" of lower-level computational processes, one that positions the user as one who can command the machine's hardware by typing in selected predefined commands. It is here that Turkle's spatial metaphors reveal their historiographical inadequacy: a simple categorical model imported from the lexicon of cultural and critical theory cannot begin to account for the complex, multivalent historical evolution of the computer.⁷ Taking a wider historical view, the move away from the

⁷For a historical survey of computer interface evolution, see Susan B. Barnes's "Computer Interfaces" and her 1995 doctoral dissertation *The Development of Graphical User Interfaces from 1970 to 1993*.

“guts” of computing hardware began not in 1984 with Apple Computer’s launch of its GUI-driven Macintosh computer, as Turkle’s analysis implies, but in the mid-1940s with the invention of the “stored program” computer architecture, a development I will discuss in detail below. While hardly as resolutely “superficial” as the Mac OS, MS-DOS must be understood as a complex representational system in its own right, one whose creation itself represents the culmination of thirty-five years’ worth of research into increasingly abstract techniques for programming and controlling computer systems. The remainder of this dissertation will insist that computing history cannot adequately be framed by metaphors and analogies derived from literary and cultural theory; instead I argue that computing historiography must be understood as a model of *layered abstraction* that finds its historical origins in the technological revolutions of World War II and that continues to unfold today. The remainder of this chapter is devoted to explaining what that layered model is and how it came into being. Chapter Two will explore how that layered abstraction led to the development of programming languages and a distinct programming aesthetics; the final chapter will discuss the consequences of these developments for literary history and theory.

Both the mathematical ideas that underpin computing and the first attempts to construct mechanical calculating devices long predate the developments of the mid-twentieth century.⁸ However, it is important to understand that these theoretical and practical explorations largely constituted distinct and separate traditions until fused together by the computing revolutions of the 1940s. Building on a mathematical tradition that comprises the work of Leibniz, Boole, and others, Alan Turing laid the theoretical groundwork for general-purpose computing in the 1930s with his “universal Turing machine” concept. Consisting of a read/write head and an infinitely long paper tape divided into squares, Turing’s machine is an abstract theoretical representation of a general purpose computing device. Capable of reading, printing, and deleting symbols on the paper tape, it

⁸In *The Universal Computer: The Road from Leibniz to Turing*, Martin Davis outlines the mathematical discoveries that laid the groundwork for digital computing. George Chase’s “History of Mechanical Computing Machinery” surveys early attempts to build mechanical computing devices. Martin Campbell-Kelly and William Aspray’s *Computer: A History of the Information Machine* emphasizes historical continuities between early attempts to mechanize the field of information processing and the evolution of digital computing.

can exist in a number of different “functional states”; its future actions, in turn, are determined by its present state, by the data it reads, and by a predefined instruction set. This deceptively simple device has enormous flexibility and power: in his famous 1936 paper “On Computable Numbers with an Application to the Entscheidungsproblem,” Turing used it to argue that “[i]t is possible to invent a single machine which can be used to compute any computable sequence” (241). However, the hardware that turned Turing’s abstract insights into concrete realities emerged from an entirely different tradition—the centuries-long effort to engineer sophisticated calculating devices.

This engineering tradition, which stretches through John Napier (1550–1617), Blaise Pascal (1623–1666), Charles Babbage (1791–1871) and others culminated in the 1940s with the creation of electronic calculating machines that were also program-controlled digital computers. Fusing mathematical and engineering insights, these computers were fully-fledged embodiments of the general purpose Turing machine. In the 1950s, we see the emergence of a third distinct computing tradition. After mathematical discoveries and syntheses had made computing theoretically possible and engineering innovations had embodied those theoretical possibilities in working machines, a need arose to control those machines, to produce the instruction sets that Turing had identified as an essential element of any general-purpose computational device. This need gave birth to the discipline of computer programming, whose distinct culture and aesthetics will concern me in this study. To fully comprehend that culture and aesthetics, however, it is first necessary to understand the historical origins of computer programming and to study the evolution of its various technological and social practices. As such, I turn now to the moment when the need for programming first arose: the early digital computers that emerged from wartime scientific research in the 1940s.

The theoretical and practical strands of computing research identified above were forced together by the exigencies of World War II, when the need for better wartime technologies shaped the first major advances in digital computing. By 1942, the Allies’ ground-based anti-aircraft weapons had already proved woefully inadequate at combating the Nazis’ fast, highly maneuverable airplanes; likewise, the Allies desperately needed better weaponry to improve the accuracy of their offensive shelling. The Allies rallied to build better, more sophisticated guns,

but then hit a mathematical stumbling block: they could not deploy new guns in the field without first calculating the tables that told gunners where to shoot. In order to hit a fast-moving airplane, ground-based gunners had to shoot not *at* the plane but in front of it; successful strikes depended on knowing exactly how far in front of the flying plane to aim. Shooting at ground-based targets required similar calculations: gunners had to aim *above* the target to compensate for the parabolic curve of the shell's trajectory. Because each gun had to be calibrated and aimed differently, even the most highly-skilled and practiced gunner could not shoot accurately without consulting sets of precalculated firing tables. In turn, each new type of gun required a new set of tables. It was this need to calculate ever more ballistics tables for ever more new guns that spurred the development of digital computing in the United States. In Britain, by contrast, it was cryptanalysis, particularly efforts to break the German ENIGMA cipher, that fueled the most important computing developments. At Bletchley Park in England, a team of technicians led by Alan Turing developed special-purpose codebreaking computers—such as Colossus—that had many structural similarities to those being designed in the United States. As important as these British computing developments were to the Allied war effort, the historical impact of the machines themselves has been minimal: the British government classified the Bletchley Park codebreaking work top secret and went so far as to destroy Colossus and other computers used to do it.⁹ It was not until the 1970s that the British government began to release World War II cryptanalytical documents and photographs to the Public Record Office and Science Museum. American computing projects, by contrast, were relatively open to interested parties and early British computing engineers and mathematicians traveled to the United States in the post-war years to further their interest in the field. The contrast between the closed, classified computing projects of Bletchley Park and the relatively open, public developments in the United States explains why post-war British computers

⁹Konrad Zuse's Z3, the world's first working programmable electromechanical digital computer, was completed in 1941 but was also destroyed during the war. A replica of the Z3, built in 1960, is now on display in the Deutsches Museum in Munich. Completion of a subsequent Zuse machine, the Z4, was disrupted by air raids in Berlin; the machine was taken to several locations before arriving at the Eidgenössisch Technische Hochschule in Zurich, where it remained in use until 1955.

were based largely on American models.¹⁰

A major U.S. site for developing and testing new artillery and munitions was the Aberdeen Proving Ground in Maryland, which outsourced much of the detailed, time-consuming work of drawing up firing tables to teams of young women volunteers at the University of Pennsylvania's Moore School of Engineering.¹¹ Known informally as the "Moore School girls," these women worked full time calculating ballistics tables with the aid of simple desk calculating machines.¹² This process was absurdly arduous. Because a typical ballistics table included entries for around three thousand different trajectories, and because each trajectory took between one and two days to calculate by hand, preparing a single table cost a team of one hundred women approximately a month's worth of time and labor. Even the Moore School's state of the art mechanical differential analyzer, which then-undergraduate Herman Lukoff describes as a "tremendous maze of shafts, gears, motors, servos, etc." (18), took a month to calculate a similar table (Campbell-Kelly and Aspray 83). In the hope of devising a more efficient way to automate these mass calculations, John Mauchly, a Moore School instructor whose wife had the job of supervising the young women "computers," began to explore the possibility of building an alternative computing machine, one that would vastly outstrip both human effort and that of the mechanical differential analyzer in its ability to perform large numbers of calculations quickly and accurately. Mauchly joined

¹⁰The first working stored-program computers were built in the United Kingdom: the Manchester Mark I was completed in 1948 at the University of Manchester, and Maurice Wilkes revealed a more powerful stored-program design at Cambridge University the following year. Atsushi Akera explains this apparent historical anomaly by pointing out that "there was a straightforward integration between the mathematical and engineering aims of the British projects that pushed their work forward more rapidly" while "U.S. projects were plagued by overambitious goals, diffuse research interests, and financial difficulties" (69).

¹¹Atsushi Akera notes that ballistics calculations are based on a branch of mathematics known as *analysis*, which was a standard part of the University of Pennsylvania mathematics curriculum at the time (65). Moore School women would thus have been familiar with the mathematical theories underpinning the problem of ballistics calculation.

¹²Quoting the *Oxford English Dictionary* definition of *computer* as "one who computes; a calculator, reckoner; *specifically* a person employed to make calculations in an observatory, in surveying, etc.," Martin Campbell-Kelly and William Aspray note that the word *computer* originally indicated a human occupation. This use of the term was dominant in the nineteenth century, and was still in common use prior to World War II (9). Only after World War II did the word *computer* come primarily to signify electronic calculating devices.

forces with a talented young University of Pennsylvania engineer named John Presper Eckert and together they designed the computer that would become known as ENIAC. An electronic system of unparalleled size and complexity, ENIAC's design consisted of 18,000 vacuum tubes, 70,000 resistors, 10,000 capacitors, 6,000 switches, and 1,500 relays. It cost almost half a million dollars to build; when finally constructed it weighed thirty tons, and filled a large room.¹³

Although ENIAC never actually contributed to the war effort—it was not ready for production work until November 1945—its construction marked a quantum leap over previous attempts to automate calculating technology. In particular, ENIAC offered a raw processing speed that far exceeded the capabilities of existing mechanical and electromechanical devices. Consider that in 1937, Howard Aiken, then a graduate student in theoretical physics at Harvard, drew up a proposal for an electromechanical calculating machine and presented it to IBM, which provided more than \$100,000 in funding and a great deal of talented engineering support to develop, build, and test the “IBM Automatic Sequence Controlled Calculator”—otherwise known as the Harvard Mark I. When Aiken's machine¹⁴ ran its first test program in January 1943, it could process two hundred instructions per minute, a computational feat that absolutely captured the public imagination—*American Weekly* called the Mark I “Harvard's robot superbrain,” and *Popular Science Monthly's* headline was “Robot Mathematician Knows All the Answers” (Campbell-Kelly and Aspray 74). But when Mauchly and Eckert's digital ENIAC became operational less than three years later, it could manage *five thousand operations per second*, making it a thousand times faster than Aiken's computer. The ENIAC's processing speed, phenomenal for the time, spoke to the key

¹³For a history of the ENIAC computer, see Scott McCartney's *ENIAC: The Triumphs and Tragedies of the World's First Computer*. Arthur and Alice Burks's 1981 article “The ENIAC: First General-Purpose Electronic Computer” offers a highly-detailed technical description of the ENIAC's architecture, as does Jan Van der Spiegel, *et. al.*'s “The ENIAC: History, Operation and Reconstruction in VLSI.” The latter article recounts how a team of University of Pennsylvania faculty and students celebrated the ENIAC's 50th anniversary in 1996 by reconstructing the thirty-ton machine on a 7.4 x 5.3 mm square silicon chip.

¹⁴Aiken, then a Navy lieutenant, took charge of the Mark I during World War II, but to call him its “inventor” is to fall victim to Aiken's own distorted version of history. At the Mark I's official dedication on August 7, 1944, Aiken took full credit as the machine's sole inventor; his arrogant, self-interested appropriation of IBM's engineering efforts and research dollars infuriated the company's founder, Thomas J. Watson. See Campbell-Kelly and Aspray 69–76.

role computers played in a war where technological research and development were crucial to military strategy and home defense.¹⁵ It also spoke to the key role the war played in creating the conditions for modern computing.

The invention of large scale, general purpose, electronic digital computers during the 1940s marks the beginning of the history of software abstraction that concerns me here. While we take categorical distinctions between “hardware” and “software” for granted today, it is important to note that this seemingly self-evident distinction simply did not exist in the nascent days of digital computing; indeed, the very idea of a computer as a general-purpose programmable device (the concept that underpins our modern use of computers to send e-mail, compose word-processed documents, calculate spreadsheets, and perform countless other tasks) was largely absent from the earliest computing projects. Although Turing had laid the theoretical groundwork for general purpose computing a decade earlier, the urgency of the war effort led early architects and coders to focus almost exclusively on the computer’s potential for performing high-speed calculations. For instance, Howard Aiken, skeptical about the computer’s potential as a general-purpose information processing device, once wrote that “if it should turn out that the basic logics of a machine designed for the numerical solution of differential equations coincide with the logics of a machine intended to make bills for a department store, I would regard this as the most amazing coincidence that I have ever encountered” (qtd. in Davis 140). So certain was Aiken that computers should be designed to perform specialized functions that he completely failed to recognize the computer’s commercial potential as an information processing device. Seeing no future whatsoever in business computing, he forecast that the American market for computers would never grow beyond five or six commercial machines (Ceruzzi 13).

Even though we can now appreciate the absurdity of Aiken’s prediction—by the early twenty-first century it is now almost impossible to imagine a business that *doesn’t* utilize digital computing in some manner—his beliefs were commonly held in digital computing’s earliest years: R. W. Hamming remarks that programmers did not fully realize until the early 1950s that the

¹⁵On the role of the military in computing research from the early 1940s until the early 1960s, see Paul Edwards, *The Closed World: Computers and the Politics of Discourse in Cold War America* 43–73.

computer could do more than just crunch numbers (Metropolis *et. al.* 8). By the time computers started to move into the corporate world in the mid-1950s, though, it was clear that those who saw only a limited role for computers in business had grossly underestimated both the wide appeal and the broad applicability of high-speed data processing. The truth of Alan Turing's 1936 demonstration was by then apparent to all: general-purpose computers could apply the same basic logical processes to *any computable problem*. It was thus far from an "amazing coincidence" that computers could calculate department store bills just as easily as they could solve differential equations; such versatility was, indeed, the very *point* of computers. The key to making computers usable—and therefore profitable—lay not in designing them to perform specific functions or calculations, but in building general-purpose hardware that could easily be programmed to perform any task. This was easier said than done: important conceptual and architectural changes had to occur before Turing's theoretical concepts could be translated into general-purpose computers that were fast, flexible, and easy to program. For our purposes, the most significant of these changes involved refining and simplifying the means by which computers could be told what to do. Thus I focus here on two crucial components of the modern, multi-purpose computer: the creation of software (as an abstraction from hardware) and the related development of programming languages.

Few technological inventions bear less similarity to their earliest ancestors than the programmable digital computer. One can easily trace the genealogy of the automobile, train, and airplane from their earliest incarnations to their modern counterparts; however, the sleek laptops and handheld PDAs on which we compute today bear almost no resemblance to the hulking thirty-ton ENIAC, even though they are unimaginably more powerful. What holds true for the computer's external architecture also holds true for its programming architecture, so much so that programming techniques of the 1940s would be virtually unrecognizable to most programmers today. The earliest programmers had no notion of the "integrated development environments," "virtual machines," "programming languages," or even "operating systems" that help automate the work of giving instructions to a modern computer: in the absence of such programming aids, their work mostly entailed inputting the exact machine instructions (in binary or octal code)

necessary for the computer to carry out a particular task and specifying to the computer exactly where the instructions and data would be stored in memory. Such direct interaction with the computer's low-level processes presumed a familiarity with the mathematics of computation itself: specifying even basic operations required the programmer to produce complex sequences of machine instructions. After writing these instructions in the numerical, extraordinarily abstract notation of machine code, the coder would punch them onto paper tape, and feed the code into the machine via a mechanical reading device. Although this heavily mechanized, manual system of early coding had numerous methodological limitations, it was this reliance upon tape or card readers that posed the most immediate problem for wartime computing architects. Given that the main rationale for powerful computers was high-speed numbercrunching, these crude input devices seriously limited the processing speed of the computer itself; a computer could function only as fast as the tape or cards could be fed into it. As we will see, architectural improvements designed to overcome these speed limitations also had a profound impact on programming methodology.¹⁶

Electromechanical computers invented by the early 1940s (such as the Harvard Mark I) were already pushing punched card equipment to their limits. When Mauchly and Eckert built the much faster ENIAC in 1944, they faced a now-classic engineering conundrum: a computing system is only as fast as its slowest or most limited component. The problem was elementary. For the ENIAC to run at its full five-thousand-instructions-per-second capacity, input speed had to keep pace with processing speed—as Mauchly himself put it, “calculations can be performed at high speed only if instructions are supplied at high speed” (qtd. in Ceruzzi 22). Adding tape readers or punched card readers as input devices was an untenable solution, yet no technology yet existed to deliver instructions at the rapid rate of five thousand per second. In desperation, Eckert and Mauchly dispensed with input devices altogether and set up a mechanism for

¹⁶A useful overview of computer architecture's evolution is Richard E. Smith's "Historical Overview of Computer Architecture." Smith notes how many of the basic concepts that define computing were already in place by 1950: "There are many computers in use today that rely on few architectural concepts that appeared later than 1950; most computers rely on none introduced after 1965. Even recent concepts such as reduced instruction set computing (*RISC*) and *parallel processing* simply reintroduce older architectural ideas, adapting them to work in new systems" (279).

hardwiring instructions into the computer by means of a plug board. The resulting “programs” were very far from what we have come to expect from software today: Martin Campbell-Kelly and William Aspray describe the appearance of the programmed ENIAC as “rather like a telephone exchange, with hundreds of patch cords connecting up the different units of the machine to route electrical signals from one point to another” (89). Mauchly and Eckert adopted this solution under the presumption that the repetitive nature of ballistics calculations, the ENIAC’s original intended purpose, would obviate the need for extensive rewiring. However, the original purpose of the ENIAC became redundant before the machine became operational; instead of calculating ballistics trajectories, ENIAC was deployed in the service of various other scientific and military projects. ENIAC thus turned out to be a more general-purpose machine than Mauchly and Eckert had anticipated; consequently, the need for extensive reprogramming far exceeded their initial expectations. ENIAC’s coders programmed the machine by laboriously rewiring its hardware, a process that could take hours, or even days, and that had to be repeated each time the engineers wanted the machine to do something new. Scott McCartney notes the degree of difficulty faced by ENIAC’s early programmers, a group of seven Moore School women, who “were given little instruction on how to make the thing work—not even an incomprehensible manual. All they had were block diagrams and wiring schematics, and the chance to quiz engineers” (96–97). Although one of these women, Jean Bartik, remembers programming the ENIAC as “the most exciting work I ever did” (97), others were less appreciative of Mauchly and Eckert’s inelegant input solution: Stan Augarten calls the task of building a functional program from the ENIAC’s endless muddle of cables, plugs, and switches “a one-way ticket to the mad house” (128).¹⁷

Despite its cumbersome architecture, the ENIAC sparked enormous interest among engineers and mathematicians, and the post-war Moore School became a focal point for modern computing innovation. Taking the potential of digital computing far beyond the original goal of calculating ballistics trajectories, talented mathematicians, logicians, and engineers worked there to design and build the machine that would become the ENIAC’s successor: the EDVAC. One of the

¹⁷For more on ENIAC’s early programmers, see W. Barkley Fritz’s 1996 article “The Women of ENIAC.”

EDVAC designers' main goal was to overcome the obvious structural shortcomings that had forced the ENIAC's inventors to improvise the makeshift and unwieldy programming solution embodied in Mauchly and Eckert's hotwired plug board. A principal figure in this collaboration was the legendary mathematician John von Neumann, who had left Hungary in 1930 to join the mathematics faculty at nearby Princeton University. An expert in numerous fields, including meteorology, hydrodynamics, stellar astronomy, game theory, ballistics, and statistics, von Neumann served on several national wartime committees and had joined the Aberdeen Proving Grounds in 1940 as a technical consultant. By 1943, von Neumann became aware that the mathematical complexity of wartime scientific projects—particularly the complex partial differential equations crucial to the Los Alamos atomic bomb effort—could benefit greatly from the power of digital computing. While he expressed interest in various nascent computing innovations of the era (among them Howard Aiken's Mark I, George Stibitz's electromechanical relay computers, and Jan Schilt's computing work at Columbia's Watson Scientific Computing Laboratory), he was fascinated by the ENIAC, which he first discovered in 1944 through an accidental train platform encounter with Moore School engineer Herbert Goldstine. Recognizing how the digital computers under development at the Moore School could facilitate unprecedented breakthroughs in mathematical and scientific research, von Neumann visited the University of Pennsylvania in 1944 and became a consultant member of its computing team shortly afterward.¹⁸

Von Neumann immediately recognized the architectural limitations that so hampered the ENIAC and set about collaborating with its engineers to design the ENIAC's successor, the EDVAC. In June 1945, he circulated the seminal *First Draft of a Report on the EDVAC*, a document that proposed theoretical solutions to many of the ENIAC's outstanding structural and conceptual constraints. Although von Neumann listed himself as the sole author of the *Report*, and has consequently often been given sole credit for its insights, the innovations contained in the document were actually developed collaboratively among von Neumann, Eckert, Mauchly,

¹⁸For a summary of John von Neumann's work in computing, see William Aspray's "John von Neumann's Contributions to Computing and Computer Science." Von Neumann's own computing papers are collected in William Aspray and Arthur Burks's edited collection *Papers of John von Neumann on Computing and Computer Theory*.

and other mathematicians and engineers.¹⁹ The *Report*'s most noteworthy breakthrough was its proposal for a series of new logical and structural principles for approaching computer design. Now known as the "von Neumann architecture," these principles radically and definitively altered the face of computing, so much so that they continue to define computing today. The von Neumann architecture's central innovation was the idea of the "stored program," which made it possible for computers to "read in" programs via a traditional card or tape reader, and store them in memory as digital code.²⁰ A computer designed according to the "stored program" model would execute programs *from memory*, thus circumventing the structural limitations that had so hampered the ENIAC. A stored program computer was not limited by the slow speed of manual input devices; a program could run as fast as electrical pulses could travel through its circuitry.

The stored program model did more than just solve the mechanical problems that plagued the ENIAC: it also effectively restructured the very idea of the computer itself. Able to store both instructions and data as binary code, the "stored program" computer was a profoundly different kind of machine than its predecessors. Where earlier computers were primarily mechanical, the von Neumann computer was fully *digital*; its instruction sets did not reside in paper, tape, or wiring, but were instead stored in the computer's memory as electronic data. The invention of the digital computer thus marked the separation of computer hardware from computer programs: it is on this fundamental separation of hardware and what would eventually become known as software that all subsequent debate about the nature of code rests.

As we have seen, early programming methodology was fraught with inefficiency. Whether inputting machine code on punched tapes or wiring cables into plugboards, programmers found their work to be a difficult, frustrating, and error-prone business. Howard Aiken based his pessimism about the future of digital computing in part on the painfully laborious process of

¹⁹For a discussion of the *Report*'s authorship and an ensuing split between those interested in the research and commercial potential of the stored-program computer, see Campbell-Kelly and Aspray 87–97.

²⁰The EDVAC's memory was itself greatly enhanced by John Presper Eckert's research into mercury delay lines, which could store far more information than the vacuum tubes previously used in the ENIAC. Aspray and Campbell-Kelly note that delay lines "produce[d] a 100-to-1 improvement in the amount of electronics used and [made] large amounts of storage feasible" (92).

creating programs: in the 1940s and 1950s, Aiken repeatedly stressed that computers could only be programmed by trained mathematicians; noting that such mathematicians were extremely scarce. he proclaimed that “[i]f all the machines now being built are completed, there would not be enough mathematicians to run them” (qtd. in Cohen 27). However, programmers struggled to prove the pessimistic Aiken wrong. Many of the earliest computing pioneers actively sought ways to make programs easier to write, concentrating their efforts on inventing ways to circumvent the necessity of writing programming routines from scratch every time they were needed. Efforts to automate the programming process produced two effects that would prove foundational for the development of future computing cultures and programming aesthetics: firstly, a culture of software sharing emerged to enable code to be recycled; secondly, machine code was abstracted into higher-level programming languages.

Programmers shared information so that they could avoid the huge waste of time and effort involved in rewriting the same basic tools and utilities each time they were needed. Grace Murray Hopper, one of the Harvard Mark I's first programmers, recalls the informal process for obtaining shared code: “If I needed a sine subroutine, I'd whistle at [a fellow programmer] and say ‘Can I have your sine subroutine?’,” she remembers, “and I'd copy it out of his notebook.” Looking back, she notes that “as early as 1944 we started putting together things which would make it easier to write more accurate programs and get them written faster” (Wexelblat 8). Understood as a kind of public cheat sheet, the earliest computing code existed primarily as written notation, often scribbled in notebooks or even on scraps of paper. Such informal information sharing was pivotal to early computing culture and flourished through the 1950s and 1960s as computer hardware manufacturers, corporate customers, and individual programmers cooperated to share programs. The main incentive for such cooperative code-sharing was a pragmatic one. As Paul Armer explained in 1956: “The amount of redundant effort was horrendous; the cost of developing a system for using the machine, and a set of routines to go with that system, was usually in excess of a year's rental for the equipment” (124). So eager were companies to reduce that redundant effort that they formed cooperative groups where they could swap and “recycle” code. SHARE, formed in 1955 to facilitate the sharing of code for the IBM 704, was the largest such cooperative,

bringing together programmers from such disparate companies as Bell Labs, Boeing, Dow Chemicals, Esso, General Motors, General Electric, Lockheed, Los Alamos, MIT, Rand, SAC, Standard Oil, and Union Carbide. As this list of unlikely alliances indicates, even companies that competed directly with one another for business—such as Esso and Standard Oil—were happy to collaborate in creating and sharing code.

The pragmatic practice of sharing subroutines had far-reaching—if unintended—effects on the shape of software itself. Computing historian I. Bernard Cohen notes that

The chief programmer of Mark I, Richard M. Bloch, kept a notebook in which he wrote out pieces of code that had been checked out and were known to be correct. . . . Both Bloch and Bob Campbell had notebooks full of such pieces of code. Years later, the programmers realized that they were pioneering the art of subroutines and actually developing the possibility of building compilers. (“Howard Aiken and the Dawn of the Computer Age” 117)

Subroutines, segments of standard code that could be stored in a library and “plugged in” to other programs as needed, were the first major innovation in making computer programs faster and easier to write. John von Neumann and Herbert Goldstine advocated using subroutines in their 1948 three-part report “Planning and Coding Problems for an Electronic Computing Instrument,” and a great boon came in 1951 when English engineers Maurice Wilkes, David Wheeler, and Stanley Gill published *The Preparation of Programs for an Electronic Digital Computer: With Special Reference to the EDSAC and the Use of a Library of Subroutines*. As the title indicates, the volume reprinted a library of subroutines for use with Cambridge University’s EDSAC, the first working stored-program computer.²¹ The book circulated so widely among programmers of the early 1950s that it effectively cemented a tradition of sharing code. At the same time, Grace Murray Hopper further encouraged the use of subroutines by creating a “compiler,” an automatic device that would copy subroutine code into programs and eliminate the tedium associated with hand-editing register addresses.²² First released in 1951, Hopper’s innovation dramatically reduced the time and effort required to assemble complex working programs.

²¹For a detailed account of early programming activity on Cambridge’s EDSAC, see Martin Campbell-Kelly’s “Programming the EDSAC: Early Programming Activity at the University of Cambridge.”

²²Although Hopper used the term *compiler*, her invention was not a compiler in the modern sense. Hopper’s program only automated subroutine handling; it did not translate high-level

During the 1950s, the stored-program architecture's potential to advance programming methodology began to become clear. Because the von Neumann architecture allowed computers to store instructions in memory as electronic data, it also enabled programmers to write programs that could themselves write the low-level binary code that formerly had to be written by hand. This new technology allowed software to become what Michael Mahoney calls a "self-programming machine" (91). The 1950s saw both the rise of "assemblers," which allowed programmers to replace machine code with a shorthand symbolic notation known as assembly language, and then the emergence of fully-fledged high-level programming languages and compilers.²³ The first such language was developed by IBM between 1954 and 1957. The brainchild of programmer and mathematician John Backus, it was known as FORTRAN (FORMula TRANslation).

FORTRAN's successes were twofold: first, it allowed programs to be written in concise, readily comprehensible mathematical notation; second, the FORTRAN compiler would then translate that notation into efficient machine code without the programmer needing any knowledge of low-level computational processes.²⁴ Jean Sammett gives a sample problem that FORTRAN could be used to solve:

Construct a subroutine with parameters A and B such that A and B are integers and $2 < A < B$. For every odd integer K with $A \leq K \leq B$, compute $f(K) = (3K + \sin(K))^{\frac{1}{2}}$ if K is a prime, and $f(K) = (4K + \cos(K))^{\frac{1}{2}}$ if K is not a prime. For each K , print K , the value of $f(K)$, and the word PRIME or NONPRIME as the case may be.

Assume there exists a subroutine or function PRIME(K) which determines whether or not K is a prime, and assume that library routines for square root, sine, and cosine are available. (151)

programming languages into low-level machine code. However, for a remarkably forward-looking paper that anticipates many modern programming techniques, see Grace Murray Hopper's "The Education of a Computer" (1952).

²³For the sake of brevity, I eclipse here the many important automatic coding systems developed prior to 1957 that helped pave the way for high-level programming languages. Chief among them are Short Code, A-2, and A-3 (developed for the UNIVAC), Speedcoding (for the IBM 701) and the Laning and Zierler system (for Whirlwind).

²⁴John Backus describes the evolution toward FORTRAN, as well as the development of FORTRAN I, II, and III, in "The History of FORTRAN I, II, and III." Nathan Ensmenger explores the implications of both FORTRAN and COBOL for 1950s programming culture in his doctoral dissertation; see *From "Black Art" to Industrial Discipline: The Software Crisis and the Management of Programmers* 58–79.

Here is the FORTRAN program that solves this problem:

```
      SUBROUTINE PROBLEM (A, B)
      INTEGER A, B
      J = 2*(A/2) + 1
      DO 10 K = J, B, 2
      T= K
      IF (PRIME(K) .EQ. 1) GO TO 2
      E = SQRT (4.*T + COS (T))
      WRITE (1, 5) K, E
      GO TO 10
2     E = SQRT (3.*T + SIN(T))
10    CONTINUE
5     FORMAT (16, F8.2, 4X, 8H NONPRIME)
6     FORMAT (16, F8.2, 4X, 5H PRIME)
      RETURN
      END
```

Once translated into machine code and executed on the computer, this FORTRAN program will calculate the required results. Because the program draws its square root, sine, and cosine subroutines from a separate code library, the programmer does not have to tell the computer how to calculate these functions. In fact, the programmer can rely on the compiler to include pre-written library functions automatically in the compiled code; there was no need for the programmer to know how these library functions worked or even how they were themselves programmed. Although FORTRAN handled large data structures poorly and was limited, at least in its early incarnations, to a specific manufacturer's architecture, its use of both subroutine libraries and high-level programming concepts constituted a major step toward the kinds of computer programming practice that we know today.

Beginning with FORTRAN, the 1950s saw a number of increasingly sophisticated mechanisms for enabling what we can now understand as programming's progressive abstraction from computing hardware. A wide variety of programming languages followed the first version of FORTRAN, each one aiming to make it easier for a programmer to accomplish particular tasks in more intuitive and/or discipline-specific ways.²⁵ Chief among these was COBOL (COmmon Business Oriented Language), which was developed by an independent committee of programmers

²⁵Major languages introduced during the period 1957–1970 include ALGOL 58, Flowmatic, IPL V, FORTRAN II, ALGOL 60, COBOL, LISP, PL/I, ALGOL 68, Pascal, and Simula.

and computer manufacturers working with the Department of Defense. Designed to meet the needs of business computing, COBOL allowed programmers to communicate instructions in an English-like syntax. Sammett offers the following example of a COBOL sort routine (337):

```
OPEN INPUT INPUT-FILE-1, OUTPUT FILE-2, FILE-3.  
SORT SORT-FILE-1 ASCENDING FIELD-AA DESCENDING  
FIELD-BB ASCENDING FIELD-CC INPUT PROCEDURE  
RECORD-SELECTION OUTPUT PROCEDURE  
PROCESS-SORTED-RECORDS. CLOSE INPUT-FILE-I,  
FILE-2, FILE-3. STOP RUN.
```

While many professional computer scientists expressed contempt for COBOL—Edsger Dijkstra said that the language “cripples the mind” (Shneiderman 350)—the language offered several advantages over FORTRAN. Chief among these was its support for machine-independence: COBOL provided a standard language that could transcend the architectural specifics of individual manufacturers’ computers. COBOL was also arguably less intimidating than FORTRAN, giving programmers at least the illusion that they were “speaking” to the computer in their own language. In COBOL, we can see the goal reflected in Kubrick’s *2001: A Space Odyssey*, that of the computer that understands instructions delivered in something resembling ordinary written (and possibly spoken) English, and of programmers who interact with their computer without needing to know the intricacies of its hardware architecture.

Software’s abstraction from hardware is, for the purposes of this dissertation, the single most important aspect of early programming history. It was the modernization of computing architecture that enabled code to become a form of writing; the aesthetic ramifications of that shift will be discussed in Chapter Two. This shift also had important cultural and political ramifications: the transition from numerical, machinic programming techniques to more abstract, linguistic, syntactical approaches enabled code to become both a highly contentious object of economic competition and a deeply rewarding scene of cultural formation. In the next two sections, I will develop these claims, charting some of the central social, economic, and political effects of code’s linguistic abstraction in order to show how the writtenness of code gave rise to both an extremely profitable software market and an extremely vital culture of resistance to that market.

An understanding of these political and cultural movements is essential to grasp the political ramifications of coding aesthetics.

1.2 Inventing and Regulating Software

In *Questioning Technology*, philosopher Andrew Feenberg draws on the work of Bruno Latour to discuss how technology designers can embed political choices within mechanical devices in such a way as to enable and naturalize particular techno-political orders. For example, Feenberg shows how nineteenth-century machinery embodied and reinforced certain assumptions about workforce demographics. By designing machines to match children's stature, Feenberg argues, industrialists "hard-wired" child labor into technical devices; only in retrospect do we find those encoded political choices unnatural, ideological, and disturbing (86–87). In general, Feenberg argues, such political and social agendas have historically been a determining factor in technological design: "disputes over the definition of technologies are settled by privileging one among many possible configurations," he writes. "This process, called closure, yields an 'exemplar' for further development in the field." The von Neumann architecture proved to be just such an exemplar: not only do the majority of today's computers still follow the conceptual model developed almost sixty years ago, but the computing industry has capitalized on the economic and political potential built into that model. This section elucidates the technopolitical choices hardwired into computing architecture during the 1940s and discusses the entwined technical and cultural ramifications of those choices over the following decades.

In designing the stored program computer, the Moore School engineers were also designing a series of political and economic potentials. In the era preceding the von Neumann architecture, programs had been "open," visible to the trained eye in the masses of cables and plugs that composed them; by contrast, the "stored program" concept made it possible to lock digital code inside machines in an invisible, virtually inaccessible form. As a technological guarantor of secrecy and intellectual property rights, the stored program architecture had enormous economic potential. As I noted above, higher-level programming languages enabled programmers to write

software that could be independent from both individual computers and specific computing architectures; the stored program model made it possible to protect the completed software. In effect, these languages created the potential for software to become a valuable new commodity; the result of the von Neumann architecture, they contained within them the possibility of a software industry. That economic potential in turn had enormous political ramifications: a software industry built on the computer's capacity to "hide" code would be an industry that threatened the collaborative culture of programming to its very core.

As we know all too well, corporations eventually capitalized on the economic potential of the von Neumann architecture. It took several decades for that to happen, however. In this section and the one that follows, I will address the technological and social developments of the years between code's abstraction and code's commodification, concentrating specifically on why that commodification was deferred, on the crystallization of a highly individualistic and artistic programming culture during those precious unregulated years, and on how it was that the term of code's corporate deferral finally came to a close.

The absence of a large-scale software industry during the 1950s and 1960s looks at first to be a peculiar lapse in an otherwise impeccably opportunistic economy; on second glance, it proves to have been an extremely powerful marketing strategy for the dominant hardware manufacturer of the era: IBM. By 1960, the majority of computers in use were IBM models; so unrivalled was Big Blue that its seven smaller mainframe competitors had become known as "the seven dwarves" (Campbell-Kelly and Aspray 135). IBM's market share increased steadily during the 1960s: by 1969, the company had captured fully three-quarters of the worldwide mainframe market (Campbell-Kelly and Aspray 147). A major producer of software as well as hardware, IBM supplied entire suites of programs to major client industries such as banking, retail, and manufacturing. IBM did not, however, sell this software separately from hardware as an independent, independently valuable entity; instead, IBM based its business model around leasing a complete package of hardware, software, and support services. Larger companies employed their own programmers to write customized software for individual business needs, and many of these programmers and clients in turn readily swapped and recycled their code through groups such as

SHARE. IBM was thus well positioned to profit from this software sharing culture: cross-company collaboration among programmers saved IBM much of the time and money it would otherwise have devoted to providing extensive and redundant support services (“programming support” at the time often involved providing customers with custom-coded solutions to their software needs). The more code in free circulation, the greater the variety of programs one could “borrow” from IBM’s growing software library or from one’s fellow programmers. The greater the variety of shareable programs, the more valuable IBM computers became to the various companies and government agencies that used them to do business. By not charging money for software, ironically, IBM cornered the computing market.

During the 1960s, IBM both established seemingly inassailable market dominance and began to make a serious bid for control over the nature and definition of programming itself. Thus it was not surprising that by the end of the decade the U.S. government had become concerned about IBM’s seeming monopoly. With an antitrust suit looming, IBM appointed a task force in December 1968 to review its software bundling strategy. However, this action by itself was not enough to appease the government: in January 1969, the Department of Justice initiated an antitrust investigation against IBM, an action that definitively altered both the culture of software sharing and the nature of software development. On June 30, 1969, IBM announced sweeping defensive changes in its business model: as of January 1, 1970 it would reduce the price of hardware leases by three percent, but would sell software and systems support services separately. By enabling independent programmers and other companies to write commercial software for IBM hardware, IBM hoped to downplay its monopolistic profile, and so deter litigation. This did not happen—in fact, the government’s investigation of IBM dragged on until 1982, when the Reagan Administration dismissed it as being “without merit” (Ceruzzi 171). IBM thus retained its dominant market share in hardware, which was after all the more “valuable” part of the computer, but its 1969 decision opened software to the creative opportunities and economic pressures of the free market.²⁶

²⁶In her article “A View from the Sixties: How the Software Industry Began,” Lucanne Johnson notes that some commercial products—such as Applied Data Research’s AUTOFLOW—were available as early as 1965. Johnson points to Larry Welke’s 1967 software product catalog for additional evidence that some commercial software development preceded IBM’s unbundling

IBM's defensive maneuvering had an enormous impact upon the programming industry: its 1969 decision essentially marks the moment when the established culture of software sharing came under pressure from a newly-created software industry. By unbundling software from hardware, IBM effectively created a software market in which any third-party vendor could freely write and sell software compatible with IBM computers. One effect of this free-market software economy was that competition came definitively to replace collaboration. As software, once an "enhancement" to computing hardware, increasingly became a commodity in its own right, the tradition of software sharing was jeopardized: where companies once had an incentive to share code, they now had a profit motive *not* to share. Thus do we begin to see how the stored program architecture contained within it a number of potential political effects. The first of these was the creation of the software market; the second of these effects is software blackboxing.

Where the creation of the software market grew out of government concerns that IBM was monopolizing the computing industry, blackboxing arose out of a corporate desire to monopolize code. The story of blackboxing solidified around another computing innovation of the 1970s: the personal computer. It's the story of the violent ideological clash between two emergent computing cultures: that of the profit-oriented software entrepreneur and that of the community-oriented hobbyist programmer whose social and technological fulfillment revolves around sharing code. As "hobbyist" computers suitable for private, individual use became available during the mid-1970s, clubs arose where home users could meet and share ideas. The Homebrew Computer Club, probably the most famous such amateur association, held its first meeting on March 5th, 1975. Homebrew, and the many other clubs like it that sprang up across the U.S. from the 1970s onward, continued the traditions of software sharing that had emerged in mainframe programming culture thirty years before, bringing them into the home, where they both flourished and met with a serious and ongoing challenge from the corporate interests they publicly and unrepentantly flouted. In short, personal computers put programmers on a collision course with the new software market.

announcement. She does accept, however, that IBM's decision "helped to legitimize the concept of paying for software and was a great boon to the growing software industry. It represented a major change in the environment that was created by IBM, which totally dominated the computer market ... throughout the 1960s" (102).

An irony of the new culture of hobbyist program sharing was that it quickly met with opposition from hobbyist programmers themselves. While most hobbyists saw software sharing as a form of sociability and wanted simply to share code freely and without constraint, some entrepreneurial programmers saw the chance to turn a profit writing software for the home computer. One of the first public expressions of the new commercial sentiment came in 1975 from none other than a nineteen-year-old Bill Gates, who, with his friend Paul Allen, had co-authored a version of BASIC for the early hobbyist Altair computer manufactured by Model Instrument Telemetry Systems (MITS). Rather than making Altair BASIC freely available to users, Gates and Allen offered it for sale through the MITS product catalog. Unlike other hobbyist software circulating at that time, BASIC was not meant to be freely shared; it was meant *only* to be sold for profit. When Altair owners shared copies of the software anyway, tensions between the new corporate desire for economic control and the hobbyist's traditional desire for free, unfettered redistribution of code reached fever pitch. Altair BASIC's angry young developer, furious at losing his royalties, published an outraged "Open Letter to Hobbyists" in computing newsletters across the country. Outlining an uncompromisingly profit-oriented theory of software development, Gates's letter proved to be a galvanizing wakeup call for hackers and hobbyists who wanted to keep software free. "To me, the most critical thing in the hobby market right now is the lack of good software courses, books, and software itself," Gates begins. Under such conditions, "Will quality software be written for the hobby market?" Gates outlines his answer: quality software will only emerge if professional programmers work full-time to develop and improve it. Building on this assumption, Gates advances arguments for the professionalization of programming work and the consequent economic compensation that programmers deserve. However, software sharing, in Gates's view, makes compensation—and, as a result, the professionalization of software development—next to impossible. Contrasting the time, effort, and expense he and Allen had put into developing BASIC with the low profits the system generated (he estimates that less than ten percent of Altair BASIC owners actually paid for the program), Gates proceeds to berate hobbyists who share software:

As the majority of hobbyists must be aware, most of you steal your software. Hardware

must be paid for, but software is something to share. Who cares if the people who worked on it get paid?

Is this fair? One thing you don't do by stealing software is get back at MITS for some problem you may have had. MITS doesn't make money selling software. The royalty paid to us, the manual, the tape and the overhead make it a break-even operation. One thing you do is prevent good software from being written. Who can afford to do professional work for nothing? What hobbyist can put 3 man-years into programming, finding all bugs, documenting his product and distribute for free? The fact is, no one besides us has invested a lot of money in hobby software. We have written 6000 BASIC and are writing 8080 APL and 6800 APL, but there is very little incentive to make this software available to hobbyists. Most directly, the thing you do is theft.

What about the guys who re-sell Altair BASIC, aren't they making some money on hobby software? Yes, but those who have been reported to us may lose in the end. They are the ones who give hobbyists a bad name, and should be kicked out of any club meeting they show up at. (Qtd. in Manes and Andrews 91–92)

A clarion call for the corporatization of code, Gates's letter makes four main assertions: first, that there is such a thing as a hobbyist software "market"; second, that quality software will not be written for the hobbyist software market unless professionals do it; third, that professionals will not write software for the hobbyist market unless they are well compensated; fourth, that professional programmers won't get paid if hobbyists continue the hacker tradition of sharing software. On the basis of these assertions, Gates draws a damning conclusion about both the future of technology and the viability of hobbyist culture: "One thing you do is prevent good software from being written." Later I will show that these assumptions are both false and disingenuous; Gates was not only wrong, he was insincere. For now, I want simply to note that Gates's strategic manipulation of logic allows him to reach two extraordinarily self-serving conclusions—that sharing software is stealing ("Most directly, the thing you do is theft"), and that people who share—or "steal"—software will be caught and presumably punished (they "may lose in the end").

Equating the time-honored tradition of software sharing with criminal behavior, Gates's threatening letter set the tone for increased legal restrictions on the production and distribution of software in the coming decade. As such, Gates's letter marked a corresponding change in the status of computing code itself: once understood as *information* that could—and should—be shared and freely modified, computer programs became licensed *products* whose distribution could—and should—be regulated by law. Over the next decade, software came increasingly under the twin

purviews of intellectual property law and corporate restrictions on source code distribution. Between copyrighting programs and preventing customers from seeing the actual code, software companies purposefully enacted the sort of proprietary vision Gates the teenaged hobbyist had so presciently articulated. The spectacular corporate success of Microsoft speaks to the brilliance of Gates's conceptual strategy.

There is a word for what Gates's logic eventually produced: "blackboxing." Blackboxing refers to the now-commonplace corporate custom of keeping source code a closely-guarded secret while still providing customers with usable software. Blackboxing is possible technologically because there is a division between *source code*, or the code programmers write, and *object code*, or the code computers read. As we have seen, modern programmers typically write software in "high-level" programming languages such as C or C++, languages that have been designed to reflect the conceptual structures of programming problems rather than the "low-level" instruction sets that computers execute, or process, when running a program. The version of a program that is written in programming language is called "source code." "Object code" is the binary version of the program, the ones and zeroes the computer itself reads. A concrete illustration may be useful here to explain the difference between the two. Here is the source code for a short program, written in C:

```
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("Hello World!\n");

    return 0;
}
```

This program tells the computer to print "Hello World!" on the screen. But the computer cannot read this program as written. In order to read or execute this program, the source code has to be translated into sequences of binary code. That translation is effected by a compiler, which converts it into a piece of *object code* (also known as an "executable" or simply a "binary"). Compiling the above source code would produce an executable binary file composed of object code; running that binary file would—as expected—cause the computer to print the phrase "Hello World!" on the

screen. Real-world programs are a great deal more complex than the one above, of course, but they nonetheless operate according to the same principle: no matter how complicated the program, the mechanism through which binary code comes into being is the same. In all cases, source code files and object code files ultimately exist as separate, autonomous, and independent entities.

The elemental split between source and object code has fueled decades of debate about what code is, about how code should be distributed, and about what it means to own it. Without source code files, curious programmers cannot find out how programs function—nor can they change how programs work, or fix bugs if they find them. Here is a visual representation of compiled binary code derived from the C program referenced above:

```
1111000000000000000000000000100010111111111110011001101
0000001000100100010000011101011000101111111000010000
1101100010111111110001000101000011111100001110001101
1101110001111111101010000010001010101000000101010111
1010000000000011101011000001100010100111001100000001
0000000000000101000111110100000000000011101000010110
1111011111111111111111000111000100000001011001111110
101110101100001010111010110001101101110001111111010
0000000101010111010000000000100000001011100011111111
0110100001101010011010000000110111101111111011111111
```

This is only a brief excerpt from the binary code. Although “Hello World!” is a short and simple C program, a complete binary representation of its object code would run for many printed pages. Even in the more compact hexadecimal (base 16) format, the object code is no more hospitable to the human eye. Here is a small section of the hexadecimal object code:

```
BEC8B4D0CFF4904780E8B118A45088802FF010FB6C0EB0B51FF
508E8BF120000595983F8FF8B451075058308FF5DC3FF005DC3
78B7C24108BC74F85C07E218B74241856FF742418FF742414E8
FFFFFF83C40C833EFF74078BC74F85C07FE35F5EC3538B5C240C
C34B565785C07E268B7C241C8B7424100FBE065746FF74241C5
5FFFFFFF83C40C833FFF74078BC34B85C07FE25F5E5BC38B4424
300048B008B40FCC38B4424048300088B088B41F88B51FCC38B
48300048B00668B40FCC3A120AE4000566A1485C05E7507B800
00EB063BC67D078BC6A320AE40006A0450E80613000059A3189
0085C05975216A0456893520AE4000E8ED12000059A3189E400
```

Because it is difficult to decompile object code (i.e., to deduce from an executable binary the source code used to construct it), the “Hello World!” program can be blackboxed simply by keeping its

source file secret—a fact behind much of the power wielded by the software industry today.²⁷ If a software company chooses to keep its source code under lock and key (as almost all do), the text of its programs will remain a mystery to all but the programmers who wrote it. While end users may acquire functional programs from the company in binary format (usually by purchasing CD-ROMs or by downloading executables over the Internet) the source code underlying them remains proprietary and hidden. Microsoft Word is a classic example: millions of people use the ubiquitous word-processing program every day, yet relatively few programmers have ever seen a line of its source code. The measures taken to use secrecy as a guarantor of future profit can be extreme: Pekka Himanen comments that “information is guarded to such an extent that when one visits an information-technology company, sometimes one cannot avoid the impression that all these locks protecting information make the building similar to a maximum-security prison” (45).

1.3 Rebel Code: Free Software and Open Source

The proprietary system of software production and distribution holds obvious advantages for software companies: selling blackboxed binaries can generate enormous revenues, and secret source code can be used as the basis for further exclusive product development. But insofar as blackboxing impedes software sharing, it represents a real threat to both programming culture and the quality of the code. As a result, this system of software development has given rise to a number of pragmatic and ethical objections—objections that have fuelled a crusade among leading hackers to keep software free. There are a variety of personalities and political positions within this crusade, and I will outline the most significant of these shortly. But first I want to discuss

²⁷In theory, special computer programs called “disassemblers” or “reverse compilers” can convert binary executable code back into source code. However, given that different “high-level” commands can produce the same binary executable file, the re-created source code often bears little similarity to the original. Reverse compilers also cannot reconstruct programmers’ code comments, which are often vital to understanding the structure of code. Wary of code theft, many commercial software firms employ an additional technique known as “obfuscation” that renders reverse compilation efforts useless; furthermore, most commercial software licenses legally prohibit attempts to reverse compile the program’s object code. It is therefore practically difficult—and often illegal—to recover meaningful source code from compiled binaries. For more on reverse compilation, see C. Cifuentes’s doctoral dissertation *Reverse Compilation Techniques*.

what all opponents of blackboxing have in common: a deep, abiding commitment to what they affectionately call hacker culture.

By the mid-1960s, most computers were being used to aid scientific research and to process business data. In research labs and corporate data processing centers, programming had, as we have seen, a distinct tradition and a unique set of constraints. At the same time, a new computing culture, with its own distinct rituals and values, was beginning to emerge in computing labs around the country: that of the computing hacker. At academic centers such as MIT, young technophiles were turning their creative and intellectual energy toward computing and laying the groundwork for a culture that would become a vital force in computing over the next several decades. That culture, which grew up around the microcomputer revolution of the 1960s and 1970s and gradually merged with the “hobbyist” countercultural movements of the 1970s and 1980s, is centered on a deeply-rooted set of beliefs about what it means to program, what it means to be an artist, even about what it means to be alive. Hackers live programming as artists live their art; they see themselves as a community of committed, creative people who are devoted both to perfecting their craft and to living life as an extension of that craft.

Burrell Smith, the brain behind the Apple Macintosh computer, notes that programmers share a distinct psychology. That psychology, which he half-lovingly, half-jokingly dubs the “hacker mentality,” owes more to a state of mind than to a specific set of computer skills: “Hackers can do almost anything and be a hacker. You can be a hacker carpenter. It’s not necessarily high tech. I think it has to do with craftsmanship and caring about what you’re doing” (qtd. in Himanen 7). Eric Raymond concurs, noting that what he calls “the hacker attitude” can be found “at the highest levels of any science or art” (qtd. in Himanen 7). Hackers commonly combine their artistic approach to code with other artistic talents, notably literature and music. Programmers who consider themselves to be hackers see their “hacker mentality” as having more to do with an attitude toward life than with their computing expertise. That attitude is a special combination of aesthetic sense and problem solving. As Eric Raymond explains it in a discussion of Unix hacking,

To do the Unix philosophy right, you have to be loyal to excellence. You have to believe that software is a craft worth all the intelligence and passion you can muster . . . Software design and implementation should be a joyous art, and a kind of high-level

play. To do the Unix philosophy right, you need to have (or recover) that attitude. You need to *care*. You need to *play*. You need to be willing to *explore*. (Qtd. in Himanen 6)

A programming mentality that extends beyond computing to such artistic pursuits as literature and music is a mentality that sees the precise, specialized work of writing code as an artistic activity akin to more traditional creative modes such as making music or writing novels. It is also a mentality that understands its particular expertise as importantly secondary to its style of reasoning and its attitude toward creating; programming, in the programming mentality, is paradoxically not essential to it. What is essential to it is a unique way of working. Pekka Himanen elaborates this concept in his book *The Hacker Ethic*, which sees hacker culture as emblematic of a new work culture that rejects the Protestant ethic of asceticism. As Himanen defines it, the “hacker ethic” speaks to an individual’s capacity for total spiritual absorption in the creative process: to his powerful commitment to getting a thing done right on the one hand, and on the other hand to his equally powerful commitment to play, to making work into a game.

Linus Torvalds’s experience writing Linux offers an exemplary instance of this mentality. During the months Torvalds spent writing the initial Linux kernel code, he scarcely left the tiny Helsinki bedroom where he worked. His computer sat on a desk next to his bed, and he describes rolling right out of bed into his chair to program, and then rolling right back into bed again when he was exhausted. He worked without reference to time—black curtains on the windows prevented him from knowing whether it was night or day—and interrupted his programming only to sleep and eat (Torvalds 64). His absorption was total, his enjoyment was complete, and his mental life became more real, and more rewarding, than the more worldly existence he had temporarily vacated. Torvalds speaks of the fascination of programming, noting that “you get to create your own world, and the only thing that limits what you can do are the capabilities of the machine.” He stresses, too, that the best programmers seek creative, elegant solutions to problems: instead of generating a mundane, generic solution, a great programmer “would know to write a beautiful program that attacks the problem in a new way that, in the end, is the right way” (Torvalds 74).

As hackers have fought to preserve their culture in the face of blackboxing, two individuals have emerged as leaders: Richard Stallman and Eric Raymond. Stallman and Raymond both feel

strongly that code should never be kept secret, that anyone should be free to read it—and even to revise and rewrite it. Both men have launched grassroots movements to raise awareness about the ethical and practical problems with corporate blackboxing and to promote the cause of what Stallman calls “free software” and what Raymond, whose libertarian dynamism leads him to view the issues somewhat differently than the more controlling, more evangelical Stallman, calls “open source software.” Where Stallman focuses on the injustice of blackboxing, which he sees as a theft of the vital, life-giving culture of software sharing, Raymond concentrates on the practical benefits of sharing source code, pointing out how blackboxing ensures buggy code while code sharing creates the conditions under which errors crop up more rarely and are more easily and quickly fixed.

Ethical objections to software blackboxing find their most ardent exemplification in the controversial and uncompromising figure of Richard Stallman, founder and president of the Free Software Foundation. Stallman, who entered Harvard in 1970 to study mathematics and physics, quickly joined the community of talented computer programmers working at MIT’s AI Lab, a world-famous Artificial Intelligence research center. Working—and often even living—at the AI Lab for the better part of a decade, Stallman spent most of his time refining and extending the Incompatible Time-Sharing System (ITS) that ran on the Digital PDP-10 minicomputer,²⁸ one of his most important contributions was the powerful and versatile Emacs text editor, which is still in widespread use today. When the AI Lab community dissolved in the early 1980s after most of its members had been hired away by commercial software developers, an outraged and devastated Stallman channeled his prodigious programming talents into developing free software. In 1984 he embarked on the enormously ambitious GNU Project, aiming to recreate his lost community by writing a completely free UNIX-like operating system and development platform.²⁹ Although the

²⁸The name “Incompatible Time-Sharing System” (ITS) is a typical hacker pun, its object being the Compatible Time-Sharing System (CTSS) created by MIT professor F. J. Corbató for IBM’s 7094 computer. CTSS, widely associated with IBM-style bureaucracy, was repellant to hackers and ITS became their choice alternative.

²⁹With the demise of the PDP-10, ITS, which was tightly bound to that architecture, no longer had a platform on which to run. Stallman chose to emulate UNIX because UNIX could be ported to almost any computing platform, and it was clearly the system most favored by hackers as a replacement for ITS.

GNU Project's much-delayed Hurd kernel remains under development,³⁰ many GNU utilities are found today in the free software system known as Linux.³¹ More to the point, the GNU project became the cultural catalyst Stallman wanted it to be, drawing programmers and users from around the world into close, mutually beneficial contact through the rich tradition of code sharing.

As much a moral statement as it is a means of providing pragmatic alternatives to blackboxed code, Stallman's project is built around an ideal of liberty. Indeed, when he speaks of "free software," he does not mean software distributed for no cost—although much free software *is* free in that sense—but software that is not bound by restrictive, proprietary licensing requirements. Stallman repeatedly insists that "free software" is a matter of freedom, not cost.³² To be considered "free" in Stallman's terms, software should meet the four requirements outlined in the Free Software Foundation's "Free Software Definition": it should give you "the freedom to run the

³⁰Together with the GNU Mach microkernel and the C library, Hurd lies at the core of the GNU system. A set of servers running atop Mach, Hurd implements basic system services such as the file system, interrupt and exception handling, network protocols, and thread scheduling. Eschewing the monolithic kernel approach favored by Linux developers, GNU designers have opted for an object-oriented microkernel design. At the time of writing, the Hurd kernel is not yet ready for production use, but is making significant progress. Up-to-date information about Hurd is available at this Web site: <http://www.gnu.org/software/hurd/hurd.html>.

³¹Strictly speaking, "Linux" describes only the operating system kernel started by Linus Torvalds in 1991. However, the term "Linux" is also often used as shorthand for the entire operating system built around this kernel. Because GNU software has been instrumental in Linux development, and because a great many GNU utilities are included with Linux distributions, Richard Stallman has argued that the operating system should properly be called "GNU/Linux." In this document I use "Linux" to indicate the Linux kernel, and GNU/Linux to indicate an extended operating system built atop that kernel.

³²In April 2001, when Stallman gave a talk at MIT entitled "Copyright and Globalization in the Age of Computer Networks," he refused to allow his talk to be broadcast over the Internet. He explained why: "The software they use for web broadcasting requires the user to download certain software in order to receive the broadcast. That software is not free software. It's available at zero price but only as an executable, which is a mysterious bunch of numbers. What it does is secret. You can't study it; you can't change it; and you certainly can't publish it in your own modified version. And those are among the freedoms that are essential in the definition of 'free software.' So if I am to be an honest advocate for free software, I can hardly go around giving speeches, then put pressure on people to use non-free software. I'd be undermining my own cause. And if I don't show that I take my principles seriously, I can't expect anybody else to take them seriously." These remarks come from an edited transcript of Stallman's talk, available on the Web at http://media-in-transition.mit.edu/forums/copyright/index_transcript.html

program, for any purpose”; “the freedom to study how the program works, and adapt it to your needs”; “the freedom to redistribute copies so you can help your neighbor”; and “the freedom to improve the program, and release your improvements to the public, so that the whole community benefits.”³³ Because the freedom to study or modify software necessitates access to source code, Stallman vigorously opposes licensing arrangements—whether copyrights or patents—that prevent access to source code or that forbid its modification and redistribution and has written his own licence, the General Public Licence (GPL), designed to keep code free.³⁴

Where ethical objections to blackboxing revolve around lofty philosophical ideals, pragmatic objections to blackboxing generally center around the more mundane issues surrounding the quality and cost of proprietary software. Computer programs are rarely (if ever) released free of errors, and consumers of blackboxed software must endure extant flaws until such time as the companies concerned release modified or upgraded versions—months, or even years, may pass before outstanding bugs are fixed. Pragmatic objections to software blackboxing are articulated most powerfully and fervently in Eric Raymond’s “The Cathedral and the Bazaar,” an essay that is frequently cited as a foundational document for the modern open source software movement. In this essay, Raymond posits two basic models for software development: the “cathedral” and the “bazaar” models of his title. Under the “cathedral” model, a small team of developers works upon a piece of software until they have perfected it; only then do they release it to a wider audience. The “cathedral” process, slow, inefficient, and unnecessarily centralized in Raymond’s opinion, is the one employed by the GNU Project as well as by commercial developers; for all his emphasis on free software, Richard Stallman is, to Raymond’s mind, a cathedral builder. The bazaar model, by contrast, embodies the open-source philosophy of “release early, release often”: early versions of barely working code are released on publicly-accessible servers; the code is then maintained, developed, and deepened through the collaborative efforts of programmers all over the world. Emphasizing the importance of “open source” to programmers, to users, and

³³The Free Software Definition is available on the Free Software Foundation’s Web site at <http://www.fsf.org/philosophy/free-sw.html>

³⁴I take up the issue of software as intellectual property again in Chapter Two. A discussion of Stallman’s General Public License may be found in Chapter Three.

to the future of software, the open-source movement advocates the distribution of source code along with precompiled binaries; it actively encourages users to become co-developers; and it asks programmers who read, analyze, and improve the source code to return their enhancements to the program's user-base. Raymond's bazaar development model thus presupposes Stallman's model of intellectual property, but it circumvents Stallman's tendency toward central planning by advocating the rapid circulation of code. Limiting or hampering that rapid circulation, in Raymond's view, also impedes the highly productive open-source development model. The Linux kernel, the Apache Web server, and the Sendmail e-mail delivery program are all examples of "bazaar"-style software development. In their extraordinarily high quality they lend weight to Raymond's claim that the fast-paced open-source development model often results in high-quality code: because open source software is frequently and continuously updated, it tends to have fewer bugs and more usable features than code produced and maintained under the cathedral model.³⁵

As closely tied as the open source and free software movements are, philosophical tensions exist between the two movements. Open-source advocates focus on marketing their philosophy as a viable corporate business model, for instance, and many major corporations—notably IBM—have thus made major investments in the Linux kernel, the Apache web server, and other open source projects. Those open-source proponents are frequently embarrassed by the anti-corporate overtones of Stallman's "free software" rhetoric—indeed the very name "open source" was coined in an intentional effort to avoid the problematic word "free." Open-source even has various alternative licenses to Stallman's GPL—these include the Mozilla Public License, the Perl "Artistic License," and the X-Window System License.³⁶ In 1997, the guidelines for licensing free software were

³⁵Ironically, even Microsoft seems to agree with this proposition. In November 1998, confidential Microsoft memoranda (authored in August of that year) were leaked to Raymond, who published annotated versions on his Web site. Known as the "Halloween Documents," they outline suggestions for Microsoft's strategic response to the open-source initiative. The memo worries about the impact of open-source upon Microsoft revenue, noting that "recent case studies provide very dramatic evidence . . . that commercial quality can be achieved / exceeded by OSS [open-source software] projects." The Halloween Documents may be read on-line at <http://www.opensource.org/halloween/>

³⁶For a comprehensive list of public software licenses, see Appendix A of Donald K. Rosenberg's *Open Source: The Unauthorized White Papers*.

formalized as the Open Source Definition. This Definition “allows greater liberties with licensing than the GPL does. In particular, the Open Source Definition allows greater promiscuity when mixing proprietary and open-source software” (DiBona *et. al.* 3). In response, Stallman worries that these “greater liberties with licensing” will entail reduced liberties for software users, and charges that “the rhetoric of ‘Open Source’ focuses on the potential to make high quality, powerful software, but shuns the ideas of freedom, community, and principle” (Stallman, “The GNU Operating System and the Free Software Movement” 69–70).

Despite these philosophical differences, however, what both the free software movement and the open source movement have in common is a deep and abiding sense of programming as an art, and of code as an essentially literary aesthetic form. I will develop that claim in the next chapter; for now, I want simply to note that the politics and poetics of contemporary computing culture all depend on the political, economic, technological and social potentials contained within the von Neumann architecture. That architecture, combined with the high-level programming techniques it made possible, enabled mechanical systems of plugs and cables to be abstracted into language; that abstraction in turn created the conditions for both the flourishing proprietary software market we know today and for competing notions of computer programs as forms of aesthetic expression that should be freely shared and available to all.

Chapter Two of this dissertation discusses how an aesthetic, explicitly literary attitude toward textuality has become central to computing culture; Chapter Three argues that we can use advanced modern programming philosophies to make sense of Joyce’s compositional approach to *Finnegans Wake*. My argument in these chapters is that the seemingly separate fields (and thought systems) of literary studies and computing are actually crucially embedded in one another, and that study of one can illuminate the other. The remaining sections of this chapter examine the institutional and political divides that prevent such a mutually beneficial interpenetration of cultures and thought systems from taking place. It will be my contention that the humanities has figured “the computer” in a way that has done more to sustain a debilitating divide between the two fields than to integrate them in mutually enabling, creative ways. I will argue here that far from generating useful knowledge or deep understanding of computing within the humanities,

“cybertheory” or “cyberstudies”—as the academy calls its theoretical approach to computer-related topics—produces a fantasy about computers that is largely divorced from the culture, history, and technology of computing itself. Furthermore, I will contend that cybertheory appropriates the computer to serve political agendas that are in many ways antithetical to the issues with which computing culture is itself concerned; this political dislocation in itself perpetuates the divide between the two fields.

1.4 Consensual Hallucinations: Producing a Metaphysics of Code

William Gibson coined the term “cyberspace” in his 1984 novel *Neuromancer*, using it to describe the dystopic realm of a fully networked world where existence is synonymous with computerized communication. Literary and cultural critics quickly took up the term, importing it into their theory and their thought. By the late end of the decade, “cyberspace” was an up and coming interpretive category, significant enough to warrant its own academic conference, held in May 1990 at the University of Texas at Austin. The proceedings of The First Conference on Cyberspace in turn became the basis for the first scholarly anthology on the subject, *Cyberspace: First Steps*. Edited by Michael Benedikt, organizer and arbiter of the conference, it collected many of the papers delivered at that initial gathering. As such, *Cyberspace: First Steps* both announced the arrival of a new field of inquiry and set out to set the tone and the terms of that inquiry.

Cyberspace: First Steps takes Gibson’s novel as both inspiration and deterrent: Gibson’s vision of cyberspace as “the mutual connective fabric of the conceptual universe,” the “consensual hallucination” of “billions of legitimate operators,” carried the possibility of scholarly creativity, of a world that was whatever one theorized it to be; his depiction of cyberspace as profoundly dystopic cut off that theoretical prospect at the roots, suggesting that to be enclosed within cyberspace was every bit as oppressive as enclosure within any other totalizing system. The origins of cybertheory lie at this impasse. As Benedikt explains it in his introduction, cyberspace is “an unhappy word, perhaps, if it remains tied to the desperate, dystopic vision of the near future found in the pages of *Neuromancer* (1984) and *Count Zero* (1987)—visions of corporate hegemony and

urban decay, of neural implants, of a life in paranoia and pain—but a word, in fact, that gives a name to a new stage, a new and irresistible development in the elaboration of human culture and business under the sign of technology” (1). From the beginning, cybertheory’s project has been to envision this “new stage” by idealizing the computer, to produce an “irresistible” utopian vision of all that a fully networked world could be.

At the heart of that utopian vision was the term cyberspace itself, a metaphor taken from a fantasy novel that became the architectural principle of a new scholarly discourse. In *Cyberspace Textuality: Computer Technology and Literary Theory*, Marie-Laure Ryan describes the theoretical excitement embedded within early invocations of the term “cyberspace.” The very word, she notes, “captures the growing sense that beyond—or perhaps on—the computer screen lies a ‘New Frontier’ both enticing and forbidden, a frontier awaiting exploration, promising discovery, threatening humanistic values, hatching new genres of discourse, altering our relation to the written word, and questioning our sense of self and of embodiment” (1). Describing cyberspace in frankly science fictional terms, Ryan captures the manner in which scholarship centered on this new analytical category cast itself as a sort of armchair intergalactic adventure. Cybertheory is rife with the language of exploration and settlement; the work of the cybertheorist is that of the pioneer: her principle activities are to map the philosophical terrain and to lay the intellectual foundation for cyberspace’s theoretical development. Benedikt thus calls cyberspace a “common mental geography” that has the potential to become a “collective memory or hallucination, an agreed-upon territory of mythical figures, symbols, rules, and truths, owned and traversable by all who learned its ways, and yet free of the bounds of physical space and time” (2–3). *Cyberspace: First Steps* clearly evokes in its title the image of the inaugural footprint, the act of laying claim to new territory by making a mark upon it. The allusion to U.S. astronaut Neil Armstrong’s 1969 moonwalk—itself a territorial series of first steps—is unmistakable. So is the fantastic character of the field, which developed as a quasi-analytical, deliberately speculative adjunct to Gibson’s own fantasy fiction.

The power—and credibility—of cybertheory lay, ironically, in its frankly fictional nature. As the sci-fi feel of the language surrounding the discourse suggests, cybertheory had virtually no

pretension to comprehend or care about the actual workings of technology. It was concerned almost exclusively with conceptualizing cyberspace as a realm beyond the material world—beyond even the technology that brought it into being—where, by definition, anything was possible. By the mid- to late 1990s, “New Frontier” fervor had centered itself on the most pressing problems of contemporary literary and cultural theory; the objects of cybertheoretical analyses were largely the stock categories already heavily favored by cultural studies (race, class, gender, sexuality, and, more broadly, the body and identity), and the aims of these analyses were to show how cyberspace offered solutions to problems—and problematics—that could simply never be resolved in a world bound by material constraints.

Cybertheory thus found its analytical niche in the symbolic resolution of real world problems. Centering its conceptual efforts on the central questions of its parent methodology, cultural studies, the newest and savviest field on the block advanced its own “cyber-categories” (post-human identity, cyberfeminism, the cyborg, and so on) as the messianic fulfillment of cultural studies’ theoretical and political fantasies. The leaders within the field were explicit about the visionary, imaginative aspect of their work: Michael Benedikt even invokes Gibson’s concept of “consensual hallucination” to describe the constructive work of the cybertheorist, which proceeds according to an extra-rational thought process he dubs “mytho-logic” (7). Where Gibson understood cyberspace as a disturbing sign of complicity with a mechanized hegemony, then, Benedikt and other founding cybertheorists understood it as a description of the subversive, potentially transformative power of their scholarly vision. Producing idealized narratives of what cyberspace could be, theorizing the virtual in order to make it real, cybertheory is the academic arm of science fiction.

As in other space stories—*Star Trek*, *Star Wars*, *Alien*—anything is possible in cyberspace. In cyberspace, ideas don’t have the gravitational pull of reality. The result is to give political visions analytical purchase. Ironically, the groundlessness of the “space” metaphor allows analytical agendas to ground themselves. As such, cybertheory has cast itself as the last, best hope of cultural studies. Conceived in response to the rise of cultural studies in the late 1980s, it both set itself out as a form of cultural studies and as the answer to some of the central problems cultural studies posed. As cultural studies sought to problematize identity by foregrounding race, class, and gender

as the primary theoretical categories, cybertheory set out to be the scene within cultural studies where those theoretical problems could be symbolically resolved.

Cybertheorists have thus devoted their efforts almost exclusively to mapping the categories, politics, and *modus operandi* of cultural studies onto the post-Internet technological landscape. As such, cybertheory has primarily attempted to register the effects of networked computing technology upon race, class, gender, the body, and other sub-categories of identity, and to theorize the utopian modes of “posthuman” subjectivity and corporeality that its proponents envision emerging in cyberspace. Regrettably, the field contains little sustained engagement with computing technology as a material or historical entity in its own right. This willful ignorance of the history and technology of computing has powerfully distorting effects—chief among them the capacity for distortion itself. This is best viewed by looking at the sheer range of visions cyberspace has been used to anchor. In their wildly divergent, mutually contradictory guises, the literature of cyberspace speaks eloquently, if inadvertently, to both the conceptual utility of the concept and to its methodological failings.

Optimistic critics are busily dedicated to the belief that computers are the portals to a freer, better future. Positive predictions for cyberspace center on the belief that the Internet can unmake rigid social boundaries separating races, classes, nations, genders, sexualities, ages, creeds, and so on; such readings get much of their power from the art of powerful—if frequently convoluted—assertion. Donna Haraway’s essay “A Cyborg Manifesto: Science, Technology, and Socialist-Feminism in the Late Twentieth Century” is commonly felt to be the founding moment of the “positive” or celebratory readings of computers, and it models the art of celebratory assertion so convincingly that hordes of critics have followed in her wake. In her “Manifesto,” Haraway uses the image of the cyborg—a being part human, part chip; part woman, part “integrated circuit”—to propose a socialist feminism that has since become the basis of what feminist theory calls “postfeminism,” a type of radical gender politics that ties together a number of strands in current poststructuralist and feminist theory and that is ultimately interested in theorizing what Haraway herself named as the ultimate goal of her “ironic vision”: “a world without gender.” As Barbara Kennedy explains it, “Cyberfeminism has been crucial to the current élan of cross-cultural,

theoretical and differential plateaux of contemporary feminisms . . . Post-feminism seeks to rethink the feminist voices of the 1990s, to present *situational* ethics, where we need to move beyond debates of binary thinking in which gender is perceived as immutably masculine or feminine: we should be concerned to go beyond established notions of gendered identity or subjectivity” (Bell and Kennedy 284). The computer’s power to anchor fantastic visions of a new world order—or rather a new world (dis)order—has thus become the basis for an entire school of feminist, or post-feminist thought (see especially work by Chela Sandoval, Judith Squires, and of course Haraway herself). Likewise, the computer underpins similarly liberatory visions of popular culture and subcultures (as exemplified in work by theorists such as Vivian Sobchack, Arturo Escobar, and Andrew Ross), queer sexualities and embodiment (as exemplified in work by theorists such as Timothy Leary, Gareth Branwyn, and Sandy Stone), and space and colonization (as discussed in the work of Ziauddin Sardar and David Bell). Routledge’s 2000 anthology, *The Cybercultures Reader* (edited by David Bell and Barbara Kennedy), is a testament to the breadth and energy of the bright side of cybertheory, not to mention its blithe self-confidence (the book announces itself as *the* cybercultures reader, not *a* cybercultures reader).

Other critics mirror this logic, yet reverse its optimistic emphasis, arguing that the computer is about to cost us our identities and even our culture. Relying on a watered-down, essentially Marxian idea of the machine as that which inevitably mechanizes, and so dehumanizes, its user, the pessimistic strand of cybertheory foretells a future dystopia that accepts Gibson’s equation of technological expansion with personal diminishment. Alarmist, even bitter visions of what this all-encompassing mediation will mean for our capacity to live meaningful lives are to be found in the work of theorists like David Noble, who are adamant in their belief that computers are destroying such basic and essential entities as education and even books. From the outset, then, cyberstudies has treated the computer as a grounding mechanism for a series of competing analytical agendas. Within cyberstudies, computing has become the basis for dreams of liberation, for nightmares of imprisonment, for cyborg manifestos, for conservative calls to preserve tradition. As Mark Poster writes in *The Mode of Information*, “The literature on recent developments in electronic technology and their impact upon technology has generally gravitated around

three poles: prophecies of salvation (we are entering the age of the post-human, and our mental and physical faculties will be enhanced); prophecies of doom (the advent of the post-human is inevitable, but it will mean the loss of all that is worth preserving in our cultural heritage); and Luddite calls to resistance (something can be done to defend our humanity against the steady advance of the machine)” (114). The wide range of ideas about what “cyberspace” is and what it can be, the range of readings of what it is doing to our bodies, our minds, our culture, and our future, point to the imaginative power of “cyberspace,” a concept that has provided us less with practical insight or actual knowledge than with seemingly endless critical energy.

The point is not that one reading of the computer is more “right” or “wrong” than another, but, rather, that so many readings have been written from so many perspectives about what the computer means for the future of the body, the subject, identity, culture, and politics. Unburdened by the constraints of technological understanding, floating free of its own historical context, the computer is a profoundly productive metaphor, one that can stimulate and sustain any number of fantastic interpretations precisely because it is itself a cipher, a very visible, increasingly central part of our culture whose real mechanisms are little known and less understood. Paul Virilio has described the manic quality of the interpretive energy that surrounds the computer as “frantic interpretosis,” a phrase that precisely captures both the unending, unstoppable quality of this emerging discourse and the desperation, even panic, that underwrites it. Within cultural studies, there are many sites of such “frantic interpretosis”: the body, nation, and the Renaissance stage, to take a few prominent examples, have all been read so many ways as so many things that they have ceased to carry much meaning in themselves, becoming instead the means by which larger cultural phenomena (race, gender, class, materiality, modernity, nationalism, identity) can be interpreted. Through cybertheory, the computer has become one such site; its particular symbolic work has been to locate or ground what I will call here our looming post-postmodernity, the growing sense that once “identity” shifts onto the “platform” of the computer, we will either lose sight of cultural phenomena such as the body, race, gender, or class, or become free to manipulate those categories however and whenever we want.

A common cybertheoretical theme is to perfect the transformative promise of cultural studies

by projecting its agenda onto an unreal plane. As early as 1991, Michael Benedikt explicitly acknowledged the imaginative dimension of scholarship centered on cyberspace: “All the authors address themselves to the topic with extraordinary seriousness, acumen, and enthusiasm, even though—and perhaps because—the varieties of cyberspace they imagine, describe, and sometimes criticize, do not yet exist. Indeed, the very definition of cyberspace may well be in their hands (or yours, dear reader)” (23). Since its inception, the goal of cyberstudies has remained constant: to develop an interpretive mode unfettered by reality, to create a space within criticism for futuristic imagining, to perfect the project of cultural studies by producing an interpretive analogue of science fiction. The subject of that fiction is the computer. The plot is the story of how the computer engineers a new global culture. As we have seen, even analyses of computers that are hostile to the utopian vision of cybertheory participate in this plot, accepting as a matter of faith that the computer will become—may have already become—not only the primary transmitter of culture, but the location of culture itself. In this, they follow the lead of Gibson himself, whose founding gesture was to define cyberspace as a realm whose dysphoric impact was predicated on its capacity to displace—and so become—the way we live.

It is crucial to see how deeply readings of cyberspace, whether optimistic or pessimistic, positive or negative, depend on an unexamined and largely uninformed idea of the computer. Like the vampire in *Dracula* or the creature in *Alien*, the computer’s power to scare and thrill depends on its status as part of the “unknown.” Whether prophesying doom and gloom or predicting endless utopian liberations, cybertheorists have one thing in common: they use the computer to advance a preconceived political agenda, and they license themselves to do so by strategically ignoring both how computers work and what animates computing culture. In this, too, they follow Gibson, who freely admits that his portrayals of science are not based on careful research or technical understanding: “Most of the time I don’t know what I’m talking about when it comes to the scientific or logical rationales that supposedly underpin my books” (qtd. in Coyne 29).

Taken together, cybertheoretical readings of the computer’s impact on culture are so extreme, and so extremely out of tune with one another, that we ought to wonder just how much real research underpins any of it. But this question has not been asked, as factual knowledge is neither

necessary nor conducive to fantasy. Regardless of what the fantasy is, however, the perspectives I have sketched above *are* fantasies. Indeed, what unifies paranoid and celebratory cybertheory is that few people on either side of the debate have an abiding interest in or deep understanding of either how computers work or what the people who work on them are like. That's tautological: if they had knowledge they could not have the fantasy. As it stands, though, the fantasies that polarize and paralyze the debate about computing in the humanities threaten to paralyze the humanities themselves.

Cybertheory tends to focus on modes of Internet-mediated experience and communication (what Lawrence Lessig calls the "content layer" of the computer), and it does so at the expense of the "code layer" and "physical layer" (these terms are also Lessig's) that, as we have seen, make up the computer's complex operational totality. In this, cybertheory has allowed itself to be seduced by the special effects of the phenomenon it theorizes: as the computer-generated reality of "cyberspace" largely obscures the specific hardware and software that enable those encounters, so the cybertheorist conveniently ignores the complex mechanism that creates the effect of cyberspace. The concept of cyberspace seems, indeed, to be expressly designed for the purpose of keeping computers decontextualized. The very word signals the strategically ungrounded quality of the discourse, whose "space" not only floats free of constraint, but also remains eternally empty. Moreover, as we have seen, the term "cyberspace" situates itself within a metaphorical lexicon of exploration and conquest that has become the defining terminology of the field. The aim is to configure the cultural impact—present and future—of the computer as an endlessly mappable, infinitely expanding, borderless unbounded space. As a term, cyberspace empties the computer of inconvenient content while at the same time permanently dislocating it. Cyberspace, one gathers, is truly the final analytical frontier.

Some cybertheorists have attempted to grapple with the computer as mechanism. But even when critics do try to approach the "physical layer" of technology, or the "code layer" of software, they typically revert to the familiar terminology of cultural studies analogy. A striking example of such rhetorical maneuvering may be found in Deborah Lupton's 1995 essay "The Embodied Computer/User." In a section entitled "The Frightening Computer," Lupton analyzes the

phenomenon of “computerphobia”:

Computers, unlike many other household or workplace machines, appear inherently enigmatic in the very seamlessness of their hardware. Most people have not the faintest idea what lies inside the hard plastic shell of their PC. The arcane jargon of the computing world, with its megabytes, RAMs [sic], MHz and so on, is a new language that is incomprehensible to the uninitiated. It is a well-known truism that the manuals that come with computer technology are incomprehensible and that computer “experts” are equally unable to translate jargon into easily understood language to help users unfamiliar with the technology. (484)

Having demonized computers and computer experts as frighteningly “enigmatic” and “arcane,” Lupton goes on to compare the “incomprehensible” and “frightening” computer to that catch-all staple of cultural studies discourse, the monster: “There is something potentially monstrous about computer technology, in its challenging of traditional boundaries,” she writes. “Fears around monsters relate to their liminal status, the elision of one category of life and another, particularly if the human is involved, as in the Frankenstein monster” (484). Lupton concludes her essay by drawing a dark and menacing analogy with another favorite discursive site of cultural studies: the unstable female body. The final sentence of her essay reads: “As with the female body, a site of intense desire and emotional security but also threatening engulfment, the inside of the computer body is dark and enigmatic, potentially leaky, harbouring danger and contamination, vulnerable to invasion” (487). By calling computers “monstrous,” “female,” “threatening,” and “dangerous,” and indicting computing terminology as “incomprehensible,” Lupton frees her reader and herself from responsibility for knowing anything *about* computers, while at the same time giving that reader language with which to “theorize” computers anyway. The cultural theorist can take comfort from the scare quotes Lupton strategically places around “computer ‘expert,’” which reassure us that such putative “experts” are at best locked within a prison house of language, speaking an “arcane jargon” that cannot possibly translate into “easily understood language.” Bridging that gap becomes the responsibility of the cybertheorist, who accomplishes her end by comparing computers to other “liminal” categories dear to cultural theory. Pronounced incapable of meaning anything on its own terms, the computer becomes both a monster and a female body, sites that, in the logic of cultural studies, have no distinct ontology but are instead only able to produce meaning analogically.³⁷

³⁷Erin O’Connor notes the seemingly boundless theoretical utility of the monster within

The dominance of “content”-centered interpretation has not gone entirely unquestioned. For example, Mark Hansen begins his recent study *Embodying Technesis: Technology Beyond Writing* by challenging the authority of what he terms the “culturalist position” on technology. Correctly charging that cultural theorists “invoke technology not for its own sake but as an enabling means and a material support for a more pressing account of subject constitution,” Hansen notes that theorists pervasively use technology “as a concrete placeholder for the alterity that has become, at least in the post-modern academic scene, a compulsory component of any respectable account of subjectivity” (5). Hansen responds to the theoretical limitations of cultural theory by outlining a corrective materialist direction for future technocultural criticism that will “liberat[e] technological materiality from its illegitimate, rhetorically imposed reduction,” and “reposition [technology] as the motive mechanism of an antiformalist, externally oriented neodeconstructive critical practice responsive to the ‘demands’ of embodied reality” (21). In other words, Hansen proposes to focus on the materiality of technology that technocultural criticism elides, and to use that focus as a basis for a new radically materialist critical practice. Repeatedly urging his reader to “embrac[e] technologies as materially robust entities,” he strives to use technology as a “site for a resistance against the imperialism of theoretical (or linguistic) idealism” (8). Hansen’s response to cybertheory is thus to indict its exclusive focus upon identity and subjectivity as complicit with the “imperialism of theoretical idealism,” and to push for an ever-more material materialism. However, such an insistence upon what Hansen calls “robust materiality” would surely privilege the first, physical layer: the ever-evolving web of microchips, circuits, and networks that ultimately underlie and underpin the very possibility of digital data transmission. As such, it does not account for the

cultural studies discourse. She writes: “Forming the centerpiece of studies on issues as varied as reproduction, imperialism, creativity, spectacle, and the rise of mass culture, monsters seem not only to be everywhere, but also to be able to mean just about anything. Whatever the context, though, the ‘monster’ is a figure for what is transgressive and liminal” (210). Meanwhile, feminist cultural studies frequently figure the female body as a productive site of unauthorized, transgressive meaning—in *Uneven Developments: The Ideological Work of Gender in Mid-Victorian England*, for instance, Mary Poovey proposes that the anaesthetized female body has a “problematic capacity to produce meanings other than and in excess of” what doctors intended. So similar is the work that female bodies and monsters do that they are often theoretically conjoined, as in Mary Russo’s *Female Grotesque: Risk, Excess, and Modernity* or Nina Auerbach’s *Woman and the Demon: The Life of a Victorian Myth*.

ever-increasing theoretical abstraction of computing code from computing hardware. Hampered by a failure to integrate the symbolic component of programming, Hansen's analysis ultimately suffers from the opposite problem to Lupton's.

A central contention of *The Art of Code* is that a nuanced, historically-inflected understanding of computing must acknowledge both the abstractions described in earlier sections of this chapter and the aesthetic, cultural, and political effects of those abstractions. However, a lack of specificity is endemic to discussions of technology that come from within the literary and cultural studies traditions. The result is as predictable as it is condemnable: distorted analytical paradigms that underwrite inaccurate and irresponsible histories. David Bell's introduction to *The Cybercultures Reader* unwittingly offers a telling summary of these distorted paradigms. According to Bell, there are four main ways of examining cyberspace. The first is to think of cyberspace "cartographically or schematically," as "the sum of the hardware that facilitates its practice." But such a hardware-oriented reading, Bell warns, would have to acknowledge the origins of the Internet in "the war machine"; in fact, "we might see traces of military-industrial ideologies still at work in the technologies we now befriend and entrust" (2). That, it is implied, would be bad. The second approach is to consider "the political economy of cyberspace"; however, this would also be bad because it would involve "seeing in the systems a coding which mutates the logic of industrial (global) capitalism—domination, expansion, incorporation, consumerism—into digital, viral form" (2). Because such readings yield undesirable insights that complicate any attempt to embrace the computer as an agent of liberatory critical practice, they are not in themselves particularly desirable.

Bell sketches out two alternative approaches, "the traces of which have to be set aside from the ones sketched above":

On the one hand we have the impacts of science fiction on the ways cyberspace works for us, and on the other we have the experiential, subjective sense of the hallucinatory (wired) world we are engaging with. Let's begin with science fiction. An important component of a *cultural* approach to cyberspace is to find it in our imaginations, to read its symbolic forms and meanings, to cross-reference to the ways in which it is represented. As many of the essays in this book make clear, we need to read cyberspace at the intersections of technology and representation, and see the two as mutually implicated in constituting our approach...

What we find in cyberspace is the result of this complex interplay, which works at the level of subjectivity to produce something akin to Gibson's consensual hallucination; for if we ask the question 'Where are we when we are in cyberspace?', we have to move beyond the simple answer that, physically, we are seated in front of a monitor, our fingers at work on the keyboard. We are there, to be sure, but we're simultaneously making ourselves over as data, as bits and bytes, as code, relocating ourselves in the space behind the screen, between screens, everywhere and nowhere. Moreover, if we believe certain strings of cyberhype, when we are in cyberspace we can be who we want to be; we (re)present ourselves as we wish to . . . we can be multiple, a different person (or even not a person!) each time we enter cyberspace, playing with our identities, taking ourselves apart and rebuilding ourselves in endless new configurations. (3)

Although Bell insists that "we need to read cyberspace at the intersections of technology and representation," his alternative methodologies perform an extraordinary displacement of technology in favor of representation, subjectivity, and fantasy. Bell's liberatory visions are derived from science fiction (particularly William Gibson's characterization of cyberspace as a "consensual hallucination" in *Neuromancer*), and from the realm of personal experience, from what he calls our "experiential, subjective sense of the hallucinatory (wired) world." In these idealized paradigms, technology merely provides a point of entry into cyberspace: from his seat in front of the monitor, the cybertheorist moves into "the space behind the screen," and ultimately exists "between screens, everywhere and nowhere." Like Neo in *The Matrix*, the cybertheorist has fallen down the rabbit hole, and, like Neo, the cybertheorist will reinvent herself "as data, as bits and bytes, as code," in the process appropriating "actual" data, bits, bytes, and code as metaphors for the "endless new configurations" of subjectivity made possible "in the space behind the screen, between screens," everywhere and nowhere. In Bell's ideal analytical world, technology loses all historical and material grounding as actual technologies evaporate in the movement of cyberspatial transformation, where they are replaced by allegorical technologies representing the endlessly malleable body and the multiple personalities of the cybertraveler. Concerned more with eliding the computer's historical origins than with investigating them, Bell's reader consistently shirks scholarly responsibility in order to avoid linking cyberspace to global capitalism and the military-industrial complex. It is remarkable that nowhere in the hefty *Cybercultures Reader* do we find any account of the people who actually imagined and "architected" the Internet.

The costs of such flagrant disregard for the history of the computer lead to conflation and distortions of the kinds found in Herbert Sussman's recent short essay "Machine Dreams: The Culture of Technology." In this methodological manifesto, Sussman encourages nineteenth-century scholars to imagine what he calls "Victorian technoculture" as an important antecedent to contemporary computing technology. "[T]here are the stirrings of a new project in Victorian studies," announces Sussman, and that project involves "recuperating the technological imagination, the 'technological feeling' of the [Victorian] age" (198). As Sussman correctly points out, the Victorians were fascinated with machines, engines, and contraptions of all shapes and sizes; but when Sussman effectively collapses two hundred years of technological development into a sentence, declaring that "we have seen in our own society the transformation from an economy of iron and steel, continuous with the Victorian factory and the Manchester of Engels, to a 'post-industrial' age; from the machine as snorting locomotive or smoking blast-furnace to the machine as sleek, intelligent rival of the human, from production in fluff-filled mills and superheated foundries to immaculate, if still toxic, microchip plants" (198), he essentially makes the Victorian technological experience coextensive with our present. For Sussman, locomotives, blast-furnaces, personal computers, and microchips join seamlessly into one technocultural narrative.³⁸ Sussman even lists the main technological anxieties of Victorians in ways that conveniently echo both the dominant concerns of Victorianist criticism over the past two decades and the prevailing themes of contemporary cybertheory. His list includes "panoptical surveillance; issues of gender in the masculine ideal of self-control and the attractions of the robotic female; machinery as spectacle; [...] a deconstructive sense of the machine figured in debates about prosthesis; and especially the challenge of the unprecedented 'self-acting' or intelligent machine to the idea of the human" (199). The list is instructive. In Sussman's world, technology signals a range of anxieties that are strikingly consistent—and strikingly postmodern—over time. To hear

³⁸ Another remarkable instance of this type of argument appears in Tom Standage's *The Victorian Internet: The Remarkable Story of the Telegraph and the Nineteenth Century's On-line Pioneers*. Standage's book effectively situates the Victorians as the originators of the Internet. Sussman excitedly echoes these connections between wired Victorian and wired Victorianist, announcing that "the e-mail of Victorianists now speeds through the material base of undersea phone wires laid down by the Victorians themselves" (200).

him tell it, the Victorian technocrat is not substantially different from the postmodern theorycrat, nor was the era of the steam engine substantially different from the age of the microchip. Sussman's argument may not, finally, tell us much about Victorian technoculture, but it tells us a great deal about the reductive nature of cultural studies argument.

One must question the usefulness of proceeding with definitions that ignore, or wilfully obscure, important historical and technological differences. Lumping together superheated foundries with overclocked microchips—or suggesting that Victorians tapping out telegrams are no different from Victorianists tapping out e-mail—ultimately disregards the specific contexts of technological innovations, the specific cultures that produced them, and the specific material and aesthetic natures of the technologies themselves. When placed under scrutiny, neither Hansen's insistence on the "robust materiality" of technology nor Sussman's blithe continuum between technologies past and present can begin to offer an adequate conceptualization of either digital computer technology or the aesthetic mentality that has led computer programmers to shape that technology for more than half a century.

1.5 A Manifesto for Informed Theory

The argumentative stasis regarding computers in the humanities finds its antithesis in the markedly productive, even revolutionary, debates and theories about computing being conducted among hackers themselves. Whereas a number of people in the academy are using computers to forward interpretations that finally fail to take computers into account, there are a number of programmers who have found themselves called upon to provide interpretations of the computer research they have undertaken. The difference between the two groups is as amusing as it is telling. Where the one group is made up of politicized theorists who feel no responsibility to grasp the actual technology or issues they expound upon, the other has repeatedly bumped up against politics despite its best efforts to avoid doing so. Ironically, those who are most qualified to explain the politics of computing are the hackers who have had to explain—often under duress—their beliefs, their work, and the way their beliefs and work shape one another. This happens commonly enough

that Eric Raymond and Linus Torvalds, two of the most influential such spokesmen of recent years, have coined a term for it: the accidental revolutionary. Each uses it in the title of a recent book: Eric Raymond's *Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary* began as a series of self-published web essays designed to document the anthropology of hacker culture. Torvalds was cajoled into writing his recent memoir, *Just For Fun: The Story of an Accidental Revolutionary*, by the public perception of him as the brilliant, anarchic mastermind behind the open source movement—a perception Torvalds persistently tries throughout the book to deflate, dismantle, explode, undermine, ridicule, and finally destroy.

As Raymond and Torvalds use it, the term “accidental revolutionary” speaks to priorities as well as attitudes: where cybertheorists consciously style themselves as radical, political, transgressive thinkers, as people whose ideas about computers will revolutionize our ideas about, well, everything, “accidental revolutionaries” find transgression thrust upon them. Torvalds has tellingly referred to himself as the “hood ornament” of the open source movement, while Raymond refuses to inhabit the role of Hacker Laureate by writing as idiosyncratically and widely as ever—his Web site includes sections on his pro-gun sentiments and his libertarian politics, and he has recently begun writing a “weblog” that deals with everything from sex to terrorism. While “accidental revolutionaries” such as Raymond and Torvalds don't particularly enjoy the limelight or the label, they have striven to offer the leadership and the philosophy that their fellow hackers have demanded of them. The stance of the accidental revolutionary is thus characteristically ironic and earnest by turns, and the “gurus” of the movement are as self-effacing as they are important.

Not surprisingly, given their history and their counter-culture, hackers are absorbed by questions that have a great deal to do with what it means to defy the incredibly powerful marketing juggernaut that has grown up around computers, especially around operating systems. These questions have to do with articulating a continued, ongoing rationale for resisting the huge, hugely lucrative economic system that has allowed software giants like Microsoft, Apple, Oracle and others to make billions of dollars selling something that hackers believe should be free. As such, hackers are equally concerned with both philosophy (why give up a chance to get rich?) and with pragmatics (how do you do meaningful work outside the system, and how do you share it with

others?). Hackers' philosophical pragmatism, or pragmatic philosophy, is a heady mixture of whys and hows that give deep, consequential meaning to the hacker's work while also explaining precisely what hacking *is* and why the non-computing world should care about the work hackers do. Richard Stallman has been a pioneer in this respect, having entered the political fray almost a decade before Torvalds and Raymond, and his "GNU Manifesto," which I will discuss below, exemplifies hacker philosophy, combining as it does a set of reasons why programmers should join the free software movement and listing off the kinds of code he would most like hackers to contribute to his ongoing GNU project.

Cyberstudies does not devote much space to the questions that preoccupy hackers. For the hacker, the central issues have to do with where code comes from, who owns it, who writes it, and how good it is. For the cybertheorist, there is just the impersonal, artifactual monolith: the computer. No sense of culture surrounds that monolith, apart from the vaguely amorphous culture that monolith is said to redefine. No sense of history surrounds it, apart from the vaguely defined past and amorphously free or imprisoned future that monolith is said to separate. And this means that no sense of specificity surrounds the computer, either. In cyberstudies there is little sense that computers differ from one another in what they do and how they do it, nor is there a sense that there are such things as operating systems and platforms, even though these are the things that more than anything else shape one's experience with a computer (the same PC can run a plethora of different operating systems; at a certain point, the hardware has less to do with one's relative liberation or constraint than software). Where both cybertheorists and hackers might be seen as aestheticizing the computer, then, there is a pronounced difference between their modes of aesthetics and their effects. For the cybertheorist, aestheticizing the computer (metaphorizing it) is a means of making it signify politically. For the hacker, adopting an aesthetic attitude toward coding practice inevitably pushes code into the political realm. Taking an aesthetic attitude toward programming quickly becomes palpably political. Taking an aesthetic attitude toward The Computer quickly becomes palpably bogus. As such, the cybertheorist's desire to make pronouncements about the nature of computers without actually studying computers results in all manner of buggy arguments.

The phenomenon I am talking about here, as well as the divide between the cybertheorist's

fantastical computer and the hacker's actual one, can best be seen through a simple exercise in comparing and contrasting analogous documents from cyberstudies and hacker culture. In what follows, I will read a foundational text of cybertheory, Donna Haraway's "Cyborg Manifesto," against its sister document in hacker culture, Richard Stallman's hortatory, impassioned "GNU Manifesto." Aside from the lucky coincidence of their names, there are compelling historical and theoretical reasons to put these pieces side by side. Haraway's essay was written in 1983, and an early version of it was first published in the "Orwell 1984" volume of *Das Argument*. Its final version appeared in her book *Simians, Cyborgs, and Women: The Reinvention of Nature* in 1991. Stallman's manifesto was published online in 1985, and acquired its footnote text in 1993 (adding footnotes in response to his readers' responses and questions was all the revision Stallman was willing to do). Written, disseminated, and revised during the same stretch of years, both essays purport to bring before their audience a state of emergency and to set out a plan of action for implementing or creating a better future. The one is centered on addressing the moment in the history of computing when profit-seeking corporations and their restrictive licensing procedures stopped the free circulation of code within the hacker community. The other is centered on using the idea of code to theorize how the capitalist system of patriarchal domination oppresses women. Stallman's is a pragmatic approach to liberation: he wants to free software from economic constraint by writing lots of free software. Haraway's is a philosophic approach to freedom: her desire is to free women theoretically by releasing their discourse from the constraints of traditional rhetoric (whether feminist or patriarchal). Comparing the two can tell us a great deal about the nature, and the costs, of the schism between cybertheory and programming practice.

Apart from their historical and nominal contiguity, Haraway and Stallman's essays share a deeply activist belief in the absolute necessity of unfettered freedom, particularly freedom of expression. Both are profoundly committed to and concerned with forming and maintaining communities in the face of what they see as intolerable economic and technological control (what Haraway calls "the commodification of everything"). Both are profoundly committed to and concerned with how populations marginalized by this control (women, hackers) can use the tools most natural to them to fight back. Stallman encourages his fellow hackers to resist commercial

licensing and join him in writing superior free code; Haraway encourages women to resist the rigid structures of patriarchal capitalism by developing their innately fluid relationship to all boundaries—economic, philosophic, ideological, national, sexual, ethnic, or linguistic. Both see computer technology as the means of effecting a better, freer future—for Stallman, real software will combat real problems and inequities in the present system of writing and distributing software; for Haraway, the metaphor of the cyborg will combat real problems and inequities in the present global distribution of power, wealth, and prestige. In short, both essays are the very calculated utterances of very intentional revolutionaries (Stallman is perhaps the one hacker who deliberately and resolutely identifies his activism not as an accident, but as a conscious choice and a serious, lifelong commitment).

Despite these broad similarities, however, each author pursues revolution in his or her own way. The most obvious difference between the two is, indeed, succinctly expressed in their own statements of purpose. Stallman announces: “So that I can continue to use computers without dishonor, I have decided to put together a sufficient body of free software so that I will be able to get along without any software that is not free.” Haraway, by contrast, says that “This chapter is an argument for pleasure in the confusion of boundaries and for responsibility in their construction. It is also an effort to contribute to socialist-feminist culture and theory in a postmodernist, non-naturalist mode and in the utopian tradition of imagining a world without gender, which is perhaps a world without genesis, but maybe also a world without end” (151). In other words, Stallman is building a system; Haraway is trying to dismantle one.

This basic structural difference extends to the authors’ respective styles and subject matter. Haraway’s manifesto is a heady post-structuralist cocktail of several distinct political discourses—socialism, radical feminism, Marxist materialism. It is also a post-structuralism whose most important feature is, arguably, its obscurity. Indeed, as if her “Cyborg Manifesto” were first of all a manifesto for obfuscation, Haraway begins by announcing not the problem she seeks to address, nor the solution she wants to propose, but the complexity of her own prose. The essay’s first subsection is titled, “An ironic dream of a common language for women in the integrated circuit,” and the first sentence of that first subsection announces that

This chapter is an effort to build an ironic political myth faithful to feminism, socialism, and materialism. Perhaps more faithful as blasphemy is faithful, than as reverent worship and identification. Blasphemy has always seemed to require taking things very seriously. I know no better stance to adopt from within these secular-religious, evangelical traditions of US politics, including the politics of socialist-feminism. Blasphemy protects one from the moral majority within, while still insisting on the need for community. Blasphemy is not apostasy. Irony is about contradictions that do not resolve into larger wholes, even dialectically, about the tension of holding incompatible things together, because both or all are necessary and true. Irony is about humor and serious play. It is also a rhetorical strategy and a political method, one I would like to see more honoured within socialist-feminism. At the center of my ironic faith, my blasphemy, is the image of the cyborg.

While the overall gist of Haraway's project can be intuited by someone familiar with the basic parameters of her discourse (it would be a stretch to suggest the essay is truly clear to anyone, and I suspect Haraway herself would disavow complete control over her meaning), someone new to cultural theory will be unable to identify the discourses Haraway plans to blaspheme, and will thus be utterly lost from the outset. By contrast, Stallman's essay makes crystal clear reading for a programmer, and even, certain especially technical passages aside, pretty clear reading for anyone else. Where Haraway opens with a catalogue of categories that she intends to unmake, Stallman opens with a catalogue of the components that have already been made, and a list of those yet to be created.

So far we have an Emacs text editor with Lisp for writing editor commands, a source level debugger, a yacc-compatible parser generator, a linker, and around 35 utilities. A shell (command interpreter) is nearly completed. A new portable optimizing C compiler has compiled itself and may be released this year. An initial kernel exists but many more features are needed to emulate Unix. When the kernel and compiler are finished, it will be possible to distribute a GNU system suitable for program development. We will use T_EX as our text formatter, but an nroff is being worked on. We will use the free, portable X window system as well. After this we will add a portable Common Lisp, an Empire game, a spreadsheet, and hundreds of other things, plus on-line documentation. We hope to supply, eventually, everything useful that normally comes with a Unix system, and more.

Neither of these languages is comprehensible to an outsider, but the quality and kind of that incomprehensibility differs profoundly. Haraway adopts an obfuscating, obfuscatory language that cannot readily be pinned down. This is her way of articulating a feminism that is not prescriptive (*a la* Catharine MacKinnon) but that, to her mind, allows readers to move fluidly across normally

rigid boundaries. Stallman adopts a precise, professional language that is intended to pin down the exact state of the GNU project at the time that he writes. This is his way of telling fellow programmers exactly what is needed and of prescribing for them exactly what they must do. Simply put, then, the difficulty of Haraway's language comes from her desire to unsettle our sense of language and the world it represents; the difficulty of Stallman's language comes from his determination to communicate difficult design concepts with as much accuracy as possible.

The stylistic and thematic differences between Haraway and Stallman's approaches have far-reaching effects for the relative powers of their futuristic visions. On a practical level, Stallman will know when he has achieved his vision of freedom; Haraway will not know when she has fully dismantled the patriarchal, capitalist, and discursive obstacles that block the way to hers. On a philosophical level, Stallman's vision harks backward to the ideals of liberty and free speech enshrined in the constitution and the First Amendment. (When Stallman talks about "free software" he means not software that is given away at no cost—although much free software is free in that sense—but software that is not bound by restrictive licenses. As he succinctly puts it, software should be "free as in speech, not as in beer.") Haraway, by contrast, looks forward to a post-patriarchal feminist-socialist utopia whose values and ethics are—and must always be—undefined. Most damning of all is what these differences mean for their political goals. Stallman's essay actively works in the service of the thing he is arguing for by stating the problem clearly, recruiting programmers to the cause with well-defined reasons, and by describing projects he would like programmers to take on. Over the years, thousands of programmers have read Stallman's call to arms and have joined his cause. Haraway's essay, by contrast, impales itself on the sharp inconsistency of its own uninformed cyber-rhetoric.

Haraway's dominant image is the cyborg, which she defines as "a cybernetic organism, a hybrid of machine and organism, a creature of social reality as well as a creature of fiction," and her main vehicle for communicating the power of this hybrid organism is the notion of "code." "The cyborg is a kind of disassembled and reassembled, postmodern collective and personal self," Haraway writes; "This is the self feminists must code" (163). "Coding," for Haraway is an image for rewriting, or even for overwriting, the self that has itself always already been "coded" by

what she terms “the informatics of domination,” those categories of meaning that forcibly limit our conception of what a “self,” or a cyborgian post-self, might be: “communications sciences and modern biologies are constructed by a common move—the translation of the world into a problem of coding, a search for a common language in which all resistance to instrumental control disappears and all heterogeneity can be submitted to disassembly, reassembly, investment, and exchange” (164). Coding is, in other words, Haraway’s word for both the dominant mechanisms of control in the information age and the means of wresting control away from oppressive ideologies: “The phallogocentric origin stories most crucial for feminist cyborgs are built into the literal technologies—technologies that write the world, biotechnology and microelectronics—that have recently textualized our bodies as code problems on the grid of C3I. Feminist cyborg stories have the task of recoding communication and intelligence to subvert command and control.” In Haraway’s essay, “coding” operates as both the descriptor for massive social control and the liberatory metaphor for feminist self-creation.

For Haraway, cyborgs are the special creatures of these massive codifications: “Modern medicine is . . . full of cyborgs, of couplings between organism and machine, each conceived as coded devices, in an intimacy and with a power that was not generated in the history of sexuality. . . And modern war is a cyborg orgy, coded by C3I, command-control-communication-intelligence, an \$84 billion item in 1984’s US defence budget” (151). Born of the untoward blending of body and machine, cyborgs are thus the things best equipped to overturn the codified informatics of domination from within: “Technological determination is only one ideological space opened up by the reconceptions of machine and organism as coded texts through which we engage in the play of writing and reading the world,” she notes; “No objects, spaces, or bodies are sacred in themselves; any component can be interfaced with any other if the proper standard, the proper code, can be constructed for processing signals in a common language” (152, 163). “Code,” for Haraway, is at once ideology and the actual code written into prosthetics and pacemakers and artificial intelligence and weapons; “code,” one might say, encodes the largely unwitting complicity of computer technology with the largely oppressive and dangerous powers that be. The ability of “coding” to convey the idea of a cyborg playfully and effectively undermining the system

from within thus depends on Haraway's extraordinarily unproblematic notion of code as a kind of writing that is presently entirely in the hands of the too-entitled, too thoughtless few, the scientists, technologists, capitalists, and warmongers whose special access to code gives them corresponding access to political and economic power. Hence the special force of Haraway's designation of the cyborg as "the self feminists must code."

But what does it really mean to speak of coding a self? And what does it mean to speak of coding a self in the very moment that coders are themselves trying to write a free code that will let them be themselves? Haraway's equation of coding with domination is based on utter ignorance of the "oppression" that coding was undergoing at the very moment she was writing. As Stallman points out in his contemporaneous manifesto, when corporations make it illegal to give away code, they deprive programmers of the fundamental gesture of friendship in hacker culture: sharing one's carefully crafted code with others. Stallman's free software project is as much about salvaging an endangered culture as it is about staging an outraged protest against the proprietary, stingy economics of corporations. Code is, to Stallman's mind, and, indeed, to the minds of everyone participating in the open-source movement (which is now, thanks to Linux, the largest collaborative project in the history of the world), far more aligned with art and free expression than with capitalism, nationalism, sexism, or any of the other *-isms* Haraway stacks onto it. Code is also, to Stallman's mind, and to the open source philosophy he speaks for, a fragile, susceptible thing whose vulnerability to corporate exploitation means it must be steadfastly protected (hence, for example, Stallman's special copyleft licensing procedure, which ensures that software licensed under it will always be free).

Haraway's vision of cyborgian feminists resisting the encoded oppression of capitalist patriarchy by metaphorically coding new, improved versions of themselves begins to look unrealistically ambitious alongside Stallman's far more literal plea for programmers to reclaim themselves and their culture by recovering their dying tradition of writing code for the love of it and then giving that code freely to fellow programmers and, eventually, the world. Indeed, by assuming so blithely that code belongs to the victors, Haraway manages to reinforce the very equations she claims she wants to dismantle. Participating in the sort of logic that Stallman is trying to resist, i.e.,

the belief that code is automatically the sole, proprietary possession of the dominant economic few, Haraway's manifesto is manifestly misguided. It secures its vision of transgressive women at the expense of a technological reality it knows nothing about, and so ironically plays into the hands of controlling corporate interests by granting it *a priori* all the control it desires eventually to have. Haraway's "transgressive" reclamation of coding for the purpose of feminist liberation is thus more nonsensical than activist, more reactionary than rebellious. One can only speculate about what the essay might have looked like if its author had informed herself of the actual, complex politics of computing culture during the 1980s. But one thing is sure: if Haraway had investigated the reality behind her metaphors, her metaphors would have acted very differently indeed.

In this chapter, I have demonstrated two distinct, hermeneutically apposite modes of theoretical engagement with the computer: while programmers have historically layered methodological abstractions onto the computer in order to facilitate the work of coding, recent cultural theorists have layered metaphysical abstractions onto the computer as a means of advancing their political projects. The one has obfuscated the other: as cultural theorists have appropriated the computer for their own analytical ends, they have effectively obscured programmers' own philosophical engagement with their work. As a result, they have neglected to identify an important new addition to their own field; by mapping politicized interpretive frameworks onto a machine that they assume has no ties to aesthetic or literary history, they have failed to see the very real, vital interplay between programming and literary cultures. In the next chapter, I will explore that interplay, showing how the historical evolution of software technology enabled—some might say necessitated—a corresponding evolution in programmers' conception of their work.

Chapter 2

“At the Edge of Language”: The Art of Code

Code¹

*an ode to Grace Murray Hopper, 1906–1988,
maker of a computer compiler and verifier of COBOL*

Poet to poet. I imagine you
at the edge of language, at the start of summer
in Wolfeboro, New Hampshire, writing code.
You have no sense of time. No sense of minutes, even.
They cannot reach inside your world,
your grey workstation
with when yet now never and once.
You have missed the other seven.
This is the eighth day of creation.

The peacock has been made, the rivers stocked.
The rainbow has leaned down to clothe the trout.
The earth has found its pole, the moon its tides.
Atoms, energies have done their work,
have made the world, have finished it, have rested.
And we call this Creation. And you missed it.

The line of my horizon, solid blue
appears at last fifty years away
from your fastidious, exact patience:
The first sign that night will be day
is a stir of leaves in this Dublin suburb
and air and invertebrates and birds,
as the earth resorts again
to its explanations:
Its shadows. Its reflections. Its words.

You are west of me and in the past.
Dark falls. Light is somewhere else.
The fireflies come out above the lake.
You are compiling binaries and zeroes.
The given world is what you can translate.
And you divide the lesser from the greater.

Let there be language—
even if we use it differently:
I never made it timeless as you have.
I never made it numerate as you did.
And yet I use it here to imagine

¹I am grateful to Eavan Boland for permission to cite “Code” here in full.

how at your desk in the twilight
legend, history and myth of course,
are gathering in Wolfeboro, New Hampshire,
as if to a memory. As if to a source.

Maker of the future, if the past
is fading from view with the light
outside your window and the single file
of elements and animals, and all the facts
of origin and outcome, which will never find
their way to you or shelter in your syntax—

Let it make no difference.
We are still human. There is still light
in my suburb and you are in my mind—
head bowed, old enough to be my mother—
writing code before the daylight goes.
I am writing at a screen as blue
as any hill, as any lake, composing this
to show you how the world begins again:
One word at a time.
One woman to another.

Eavan Boland's paean to computer programming is an exceptional moment in the history of poetry. First, in apostrophizing someone most poets and readers of poetry probably have never heard of, the poem demands biography. Who, it wants us to ask, is Grace Murray Hopper? Second, in addressing Hopper "poet to poet," Boland's poem operates from the premise that writing computer code is not only analogous to writing poetry, but effectively *the same*. The implication is startling: Boland's "Code" essentially announces the expansion of the poetic canon to include a mode of writing—and a particular writer—that many see as the furthest possible thing from literary art. Code, Boland asserts, is not *like* poetry; it *is* poetry. And programmers are not *like* poets; they *are* poets.

It is not clear what Grace Murray Hopper, a mathematician who did much of her programming as an officer in the U.S. Navy, would have had to say about Boland's claim. A pioneering figure in computer programming during the 1940s and 1950s, Hopper was born in New York City in 1906, received a degree in mathematics and physics from Vassar College in 1928, and joined Vassar's mathematics faculty as an instructor in 1931. In 1934 she became the first woman to

earn a Ph.D. in mathematics from Yale, and in 1941 she won promotion at Vassar to the rank of associate professor. In December 1943 she resigned her academic position to join the Navy WAVES (Women Accepted for Voluntary Emergency Service). The Navy promoted her to the rank of lieutenant in July 1944, and issued her wartime assignment: a Harvard University project dedicated to computing mathematical tables for the Bureau of Ordnance. Hopper's commanding officer at Harvard was Lieutenant Howard Aiken, and she got her first exposure to computing when she helped him program his electromechanical computer, the Automatic Sequence Controlled Calculator—also known as the Harvard Mark I.

After the war, Hopper reverted to inactive status in the Navy reserve, accepted a Harvard research fellowship in engineering sciences and applied physics, and continued to pursue her burgeoning interest in computing. In 1949 she joined the Eckert-Mauchly Computer Corporation founded by the engineers responsible for building the ENIAC digital computer at the University of Pennsylvania's Moore School, where she remained through the company's subsequent incarnations as Remington Rand, Sperry-Rand, and UNIVAC. In 1966, Hopper returned to active service in the Navy, working there for the next two decades to standardize its plethora of computer programming languages. By the time of her retirement in 1986, Hopper had been promoted to the rank of rear-admiral; at seventy-nine, she was the oldest officer on active duty in the American armed forces. Hopper died on January 1st, 1992 (not in 1988, as Boland claims).

It is hard to imagine Hopper, who spent her life steadily climbing the then-masculine ladder of academic, military, and corporate rank, recognizing herself in the feminist romanticism of Boland's poem. Rear Admiral Hopper would most likely not have had much patience with this depiction of her as a sort of techno-earth mother or digital goddess. Nonetheless, Boland's choice of Hopper as the image of the programmer-poet is peculiarly apt. In the 1950s, Grace Murray Hopper conducted important research into compilers, programs that would eventually translate human-authored "high-level" code into machine-readable instructions.² Fueled by her—correct—conviction that

²A point of clarification: as I mentioned in Chapter One, Hopper's "compiler" was not what is now known as a compiler. Modern compilers translate high-level source code into low-level object code, while Hopper's "compiler" only allowed programmers automatically to assemble subroutines into a larger program. Nonetheless, Boland refers to Hopper as "the maker of a computer compiler"

high-level programming languages would one day replace hand-coded machine language as human beings' dominant mode of interaction with computer systems, she wrote compiler software for the BINAC and UNIVAC systems and published influential scientific papers on the subject. Deeply interested in advancing the craft of software development, Hopper traveled extensively to promote advanced programming techniques; so successful were her efforts that William Aspray and Martin Campbell-Kelly consider her "the driving force behind advanced programming techniques for commercial computers, and the world's foremost female computing professional" of the 1950s (121). "Probably no one did more to change the conservative culture of 1950s programmers than did Grace Hopper," they conclude (187).

Nonetheless, it is strangely appropriate that Boland would structure her tribute to the author of compiler software through a series of translations between forms of language and communication; insofar as the poem renders Hopper's computing achievements in poetic form, it is itself a compiler. Drawing out an aesthetic relationship between poetry and code, Boland's poem ultimately makes the correspondence between the two modes of creation the beautiful thing that her poem describes. Indeed, Boland's poem is a paean of appreciation twice over. On one level, it is a feminist memorial to women's work, an ode "from poet to poet" spoken across time and space by "one woman to another" that posits parallels between its subject, Grace Murray Hopper, "maker of a computer compiler and verifier of COBOL," and its author, Eavan Boland, maker of a poem about a woman programmer whose work resembles the work of writing poetry. On another level, "Code" is an ontological experiment, a poem about what makes a poem a poem and a poet a poet that posits an essential symmetry between a poem and a program, the language poets write "one word at a time," and the language programmers write and compile into binaries: "Let there be language," writes Boland, "even if we use it differently." The one meaning is largely transparent: the poem is a feminist celebration of women's fundamental connectedness across time, space, and craft, a testament to a unifying femaleness that subsumes lesser distinctions such as age, geography,

and uses the term in its modern form, to refer to a program that translates source code into object code ("You are compiling binaries and zeroes. / The given world is what you can translate"). Given the centrality of Boland's compilation/translation analogy to her poem, I follow her logic in this reading, despite its historical inaccuracies.

and profession. It is easy to read Boland's analogies between computer programmer and poet as merely reinforcing this vision of female interconnectedness, and yet this would ignore another, more obscured meaning: "Code" is itself a sort of code, a poem about how poetic computing code can only be fully read by those who already understand—and write—code as poetry. Boland makes a parallel between poetic process and programming the basis for her imaginative, reverent identification with Grace Murray Hopper ("I never made [language] timeless as you have / I never made it numerate as you did. / And yet I use it here to imagine"). In making this parallel, Boland draws on an analogy that has long been central to computing culture: among programmers, writing code is frequently understood as writing poetry, and the programmer himself—if he is any good—is often thought of as an artist.

For Boland, seeing code and poetry as not only analogous, but as two economical and aesthetically pleasing ways of condensing meaning, is far from troubling. Indeed, it is the means of enabling one woman to identify with another, an identification that in turn becomes the means of celebrating women's professional and artistic achievements. But Boland's portrayal is exceptional, at least in literary circles. As this chapter will show, computer programmers have long embraced the idea of computing code as an aesthetic form that possesses literary qualities; yet twentieth-century writers and literary critics have only begun to entertain a concept of poetry capable of encompassing the kind of creation that is computing code.³ However, rather than adding to the expanding body of literature that considers possibilities for poetry in an age of "New Media," this chapter pursues the sinuous analogies uncovered in Boland's poem in order to trace the historical and theoretical connections between computer programming and literary practice. In so doing, I hope both to expand critical conceptions of poetic practice, and to demonstrate how models of reading and writing that literary scholars have come to regard as redundant or reactionary actually retain a great deal of aesthetic and political vitality in programming circles.

The chapter is divided into five main sections. The first deals with computer scientists,

³Some notable recent studies have considered the interpenetration of technology and poetry; foremost among them are Marjorie Perloff's *Radical Artifice: Writing Poetry in the Age of Media*, Michael Davidson's *Ghostlier Demarcations: Modern Poetry and the Material Word*, and Carrie Noland's *Poetry at Stake: Lyric Aesthetics and the Challenge of Technology*.

computer programmers, and historians of computing culture whose writings and theoretical models implicitly or explicitly present computer programming as an aesthetic endeavor. I will show that despite the enormous changes and advances that have taken place in computing technology and programming methodology over the past half-century, programmers and computing theorists have consistently employed aesthetic and literary concepts to explain the nature and meaning of their work. The enormously influential work of Donald Knuth, who has dedicated himself to “the art of computer programming” since the early 1960s, will be particularly important in framing this discussion.

The chapter’s second section examines ongoing legal debates about software copyright and patents in order to focus on the troubled ontological status of contemporary computing code. As I will show, this debate hinges on the question of whether writing code constitutes a creative activity akin to literary production, or whether programming is more properly understood as an engineering practice with purely utilitarian value. The purpose of this section will be to demonstrate that the ontological status of computing code is not a mere question of metaphor or semantics; it is central to one of the most heated and economically significant legal debates of our time.

While the first two sections of the chapter concentrate on *defining* computing code, the last three focus on the common historical practice—at least in computing culture—of *reading* code as a textual, aesthetic entity. The third section includes a discussion of John Lions, whose *Commentary on Unix 6th Edition with Source Code* has been called a “literary criticism” of the Unix operating system; it is followed by an analysis of Donald Knuth’s “literate programming” methodology, which enables programmers to generate functional object code for computers while simultaneously producing beautifully typeset, intuitively structured code for the human reader. The chapter concludes with a study of the crossover genre of Perl poetry, where poems written in the programming language Perl are simultaneously programs that can be executed (a Perl poem is not formally “valid” unless both machines and human beings can read it).

Each section of this chapter aims to illuminate a particular aspect of the close-knit relationship between code and literature; the overall trajectory of the sections constitutes in turn an attempt to trouble oppositions that have traditionally worked to prevent literary critics from recognizing

code as a literary genre. As a whole, the chapter will allow me both to question the unexamined assumption that the work of code is not on the same level, or even the same terrain, as writing poetry, and to argue for new ways—and new languages—for conceptualizing the “literary.”

2.1 Programming Aesthetics

The aesthetic history of code originated in 1843 when Ada Augusta Lovelace (daughter of Lord Byron) published one of the first theoretical articles on computing. Lovelace wrote her article in collaboration with the remarkable English inventor Charles Babbage, whose most important theoretical invention forms its subject. Babbage, often heralded as the man who designed the first computer, made his most important contribution to the history of computing during the 1830s when he designed a sophisticated machine called the Analytical Engine. The Analytical Engine succeeded an earlier Babbage invention, the Difference Engine, with this crucial distinction: while the Difference Engine could only calculate navigational tables, the Analytical Engine could perform any programmable operation its operator could specify. Babbage never managed to build a working model of the Analytical Engine. Even so, its design comprised many elements of the modern digital computer, including a central processing unit, a memory unit, and input–output devices. As such the machine signaled to those who understood it the advent of a new technological era. Ada Lovelace, Babbage’s friend and advocate, was intensely interested in Babbage’s machine, and in 1843 she collaborated with Babbage to produce an extensively annotated translation of Italian engineer Luigi Menabrea’s 1842 article on Babbage’s Analytical Engine, “Notions sur la machine analytique.” Lovelace described the programmable Analytical Engine in the notes, writing that

The distinctive characteristic of the Analytical Engine, and that which has rendered it possible to endow mechanism with such extensive faculties as bid fair to make this engine the executive right-hand of algebra, is the introduction into it of the principle which Jacquard devised for regulating, by means of punched cards, the most complicated patterns in the fabrication of brocaded stuffs. It is in this that the distinction between the two engines lies. Nothing of the sort exists in the Difference Engine. We may say most aptly that the Analytical Engine *weaves algebraical patterns* just as the Jacquard loom weaves flowers and leaves...

[T]he Analytical Engine does not occupy common ground with mere “calculating machines.” It holds a position wholly its own; and the considerations it suggests are most interesting in their nature. In enabling mechanisms to combine together *general* symbols, in successions of unlimited variety and extent, a uniting link is established between the operations of matter and the abstract mental processes of the *most abstract* branch of mathematical science. A new, a vast, and a powerful language is developed for the future use of analysis, in which to wield its truths so that these may become of more speedy and accurate practical application for the purposes of mankind than the means hitherto in our possession have rendered possible. Thus not only the mental and the material, but the theoretical and the practical in the mathematical world, are brought into more intimate and effective connexion with each other. We are not aware of its being on record that anything partaking of the nature of what is so well designated the *Analytical Engine* has been hitherto proposed, or even thought of, as a practical possibility, any more than the idea of a thinking or of a reasoning machine. (qtd. in Rheingold 34)

Noting that the Analytical Engine establishes “a uniting link . . . between the operations of matter and . . . abstract mental processes,” Lovelace recognized that as a programmable device, the digital computer constitutes a technology whose purpose is *mimetic* rather than simply *mechanical*: what impressed her about Babbage’s work was the prospect of a machine that could materialize thought—of an “engine” that was “analytical.”

In the passage quoted above, Lovelace struggles to describe what she saw as mathematics’ remarkable ability to “endow mechanism with such extensive faculties,” to externalize—and mechanize—that highest and most human of functions: abstract reasoning. She speaks of the Analytical Engine as a sort of cognitive loom that “weaves algebraical patterns.” She speaks of the Analytical Engine’s combinatorial weaving of “general symbols” as a type of language formation (“a new, vast, and a powerful language is developed for the future use of analysis”). She speaks of the uniqueness and superiority of the machine capable of generating the language that will “wield . . . truths” beneficial to “mankind.” For Ada Lovelace, “the idea of a thinking or of a reasoning machine” conjures Romantic images of poetic genius—the Analytical Engine “does not occupy common ground with mere ‘calculating machines.’ It holds a position wholly its own.” Byron’s daughter effectively casts Babbage’s machine as a technological Byronic hero and casts the work of controlling and instructing that machine as an aesthetic act (“not only the mental and the material, but the theoretical and the practical in the mathematical world, are brought into more

intimate and effective connexion with each other”). While J. A. N. Lee writes that Lovelace “[saw] the syntactic format of poetry as an analog to the format of programs” (80), I propose here that Lovelace’s aesthetic conception of programming signals more than a mere analogy. In her writings, Lovelace repeatedly tries to make programming *itself* into an art form.

Although Ada Lovelace’s actual scientific contributions to Babbage’s computing projects are a matter of dispute among historians,⁴ her *figural* contribution to programming history is foundational. Just as Howard Aiken “consciously saw himself as Babbage’s twentieth-century successor” (Campbell Kelly and Aspray 71), early programmers self-consciously resurrected the aesthetic language of Byron’s daughter to describe their craft. Although the idea that computer programming is a creative art has persisted throughout modern programming culture—often sustaining that culture, giving it identity, and allowing it to thrive—it is important to note how that aesthetic idea has evolved alongside computing architectures themselves. While Lovelace aestheticizes computer programming by applying Romantic metaphors to Babbage’s mechanical calculating, programmers from the 1950s onward grounded their aesthetic models in the stored program computer based around the von Neumann architecture (described in Chapter One). Under the architecture of the modern digital computer, as we have seen, code became an ontological entity in its own right, independent of hardware or of any other evidently material manifestation. Writing in *Computer Power and Human Reason* (1976), Joseph Weizenbaum discusses the

⁴Joan Baum eulogizes Lovelace as “a remarkable lady, overlooked in the history of science, a Victorian woman working presciently in a man’s field” (xiv); Howard Rheingold calls her the “founding parent of the art and science of programming” (31); and the U.S. Department of Defense even named its ADA programming language after her. Recent computing historians have been more skeptical about Lovelace’s accomplishments, often blaming distortions of fact upon celebratory feminist historiography. Doron Swade criticizes the tendency to lionize Lovelace, saying that “[s]he is celebrated as a woman who had apparently defied the oppression of her sex to make a mark in a man’s world, and the need for such champions has regrettably distorted her contribution” (169). Martin Campbell-Kelly and William Aspray comment that “the extent of Lovelace’s intellectual contribution to the *Sketch* has been much exaggerated in recent years... Scholarship of the last decade has shown that most of the technical content and all of the programs in the *Sketch* were Babbage’s work” (57). Babbage scholar Bruce Collier is most blunt: he calls Lovelace “the most overrated figure in the history of computing” (qtd. in Swade 168). Swade summarizes the debate about Lovelace’s understanding of mathematics and her contributions to Babbage’s projects; see 155–171.

aesthetic implications of this shift:

There is a distinction between physically embodied machines, whose ultimate function is to transduce energy or deliver power, and abstract machines, i.e., machines that exist only as ideas. The laws which the former embody must be a subset of the laws that govern the real world. The laws that govern the behavior of abstract machines are not so constrained. . . . A computer running under control of a stored program is thus detached from the real world in the same way that every abstract game is. . . . The computer, then, is a playing field on which one may play out any game one can imagine. (111–13)

Computer Power and Human Reason is a deeply reactionary work, a product of Weizenbaum's virulent antipathy to the field of Artificial Intelligence and to emergent cultures of computer hackers. Nevertheless, Weizenbaum is prepared to admit the aesthetic and imaginative potential of the stored program architecture. Recognizing that architecture's crucial distinction between physically embodied machines (computing hardware) and abstract machines (the digital processes running within that hardware), Weizenbaum identifies those abstract machines ("machines that exist only as ideas") as extensions of the human imagination. While physically embodied hardware is governed by physical laws, the abstract "machines" (or digital processes) running on that hardware are limited only by the limits of human invention. Despite Weizenbaum's resolute opposition to the idea that computers could exhibit human faculties such as intelligence or reason, he admits that computer programmers can—and do—understand code as a medium for aesthetic creation. The Romantic spirit of his vision thus echoes the spirit of Lovelace's original conception of the computer as a poetic machine.

The metaphoric work that Ada Lovelace had to do in order to convert Babbage's engines into poetic constructs was extreme; at times her language seems strained to the breaking point by its efforts to aestheticize Babbage's unbuilt Analytical Engine. But as mechanisms for software abstraction took effect during the second half of the twentieth century, attempts to aestheticize code did not seem nearly as contrived; what was for Lovelace a flight of fancy is for modern programmers simply the proper language for describing the work that they do. In what follows, I will trace the evolution of the aesthetic analogy in twentieth-century computing culture. That analogy has two main phases, corresponding to the shift from machine language to more abstract high-level programming languages. As coding techniques were abstracted into language, the

aesthetic imagery surrounding programming evolved from a generalized aesthetic language to a specifically literary one. By the time high-level programming languages came into widespread use during the 1960s, it had become customary to conceptualize programming as a form of literary composition.

Since the early days of stored-program computing, programmers have not only compared their work to art, they have believed that their work *is* art. Early programming aesthetics were born out of necessity—given the significant constraints of memory and processing power in early computers, programmers needed to make programs perform complex tasks with as few instructions as possible. Writing elegant, compact code was quickly elevated to the status of an artform: John Backus, who wrote the FORTRAN programming language at IBM between 1954 and 1957, observed that early programmers who wrote in the dense, difficult machine language or assembly language “rightly regarded their work as a complex, creative art that required human inventiveness to produce an efficient program” (“History of FORTRAN” 25). Correspondingly, corporations sought to hire programmers who manifested a creative, artistic sensibility. Nathan Ensmenger notes that during the 1950s, “creativity and a mild degree of personal eccentricity” were widely believed to be signs of programming talent. (39). Though they were technically “organization men,” programmers were very far from following either corporate norms of hierarchical behavior or, in some cases, from feeling loyalty to corporations themselves. Modeling themselves after creative artists, computer programmers enjoyed more individual autonomy than was normal within the 1950s corporate structure; corporations, in turn, tolerated programmers’ bohemian, aesthetic ideals in the belief that an aesthetic sensibility was best suited to the job of programming.

While the association of code with art permeated early corporate programming culture, it crystallized most clearly in the figure of the computer hacker. Unlike corporate programmers, hackers largely spent their time on minicomputers—smaller, transistor-based computers that started to appear in the late 1950s—in academic computing centers, and were thus even more free than their corporate counterparts to explore the aesthetic potential of code. In *Hackers*, Steven Levy gives a detailed history of the art of code during this era, showing how the term “hacker” has always indicated a mastery of computers and programming that goes beyond mere technical

proficiency and reaches into the realm of the artistic. Levy's description of a young hacker named Peter Deutsch approaching MIT's TX-0 computer in 1959 makes this aesthetic association clear: "Something about the orderliness of the computer instructions appealed to him: he would later describe the feeling as the same kind of eerily transcendent recognition that an artist experiences when he discovers the medium that is absolutely right for him. *This is where I belong*" (17, original emphasis). These early programming enthusiasts fully understood that the flexibility and potential of the computer was limited only by the flexibility and potential of its programmer: a computer could do anything a programmer could make it do. The belief that "you can create art and beauty on a computer" was deeply embedded in the "hacker ethic," the foundational manifesto of hacker culture (30). For hackers, "code had a beauty of its own" (Levy 30).

The programming aesthetic Backus and Levy describe in early corporate and hacker programming cultures is firmly grounded in the mechanics of machine and assembly language programming. However, as I described in Chapter One, programming techniques during the 1960s moved away from these abstruse, difficult methods and toward more accessible, flexible, and hardware-independent "high-level" languages. This shift also marked the moment that corporations began formally to "manage" the programmer by defining and delimiting his work.⁵ In *Windows on the Workplace*, Joan Greenbaum notes how one important technological development, IBM's unified System/360 architecture, marked the advent of a new understanding of the nature and culture of programming:

The first step that management took to gain control over the programming workforce was to divide the conceptual work of programming from the more physical tasks of computer operations. Although this division was put into effect in the aerospace industry in the mid-1950s and subsequently used by companies that had defense contracts, it wasn't until the mid-1960s that it spread elsewhere. By 1965, when IBM began installing the general-purpose System 360, both the more expensive hardware (a large mainframe computer) and the easier to use software (an operating system that could be controlled through commands rather than operators working switches), gave upper and middle managers room to begin enforcing the separation of programming from operations. Operators were to stay in the "machine room" tending the computer,

⁵In his 2001 dissertation *From "Black Art" to Industrial Discipline: The Software Crisis and the Management of Programmers*, Nathan Ensmenger gives an invaluable review of the managerial literature surrounding 1950s and 1960s corporate software development—see especially pp. 12–43, and pp. 93–165.

while programmers were to sit upstairs and write the instructions. Those of us in the field at the time remember feeling that a firm division of labor had been introduced almost overnight. (44–45)

These new managerial distinctions between programmers and operators, writers and mechanics, worked to contain what had become a fairly unruly, iconoclastic group within the corporate structure. With System/360, IBM introduced a hardware design that not only streamlined programming, but in so doing made it possible for corporations to streamline, control, and discipline programmers themselves. This managerial potential would only grow as machine-independence became an important design factor in high-level programming languages.⁶

As programmers lost contact with hardware, they became writers who no longer needed to understand how to manipulate the intricate idiosyncrasies of specific mechanical systems. This in turn represented a threat to the culture of programming that had grown up over the previous two decades: as hardware operation became separate from software production, programmers risked losing to bureaucratic compartmentalization the bohemian independence of their craft. However, programmers responded to these new programming techniques by adapting the aesthetic imagery that had grown up around early coding practice. Once the work of manipulating machine code had been abstracted into the work of manipulating *language*, the act of programming computers began to take on specifically literary connotations. As IBM project manager Fred Brooks put it in his 1975 book *The Mythical Man-Month*: “The programmer, like the poet, works only slightly

⁶Writing in 1998, Yale computer scientist David Gelernter criticizes the by-then thirty-year-long managerial effort to eliminate aesthetics from programming practice. Noting that “[a] good programmer can be a hundred times more productive than an average one, easily,” Gelernter argues that “[t]he gap has little to do with technical or mathematical or engineering training, much to do with taste, good judgment, [and] aesthetic gifts.” Corporations ignore this fact at their peril. Criticizing software managers for their failure properly to appreciate the aesthetic component of software development, and for their tendency to “discipline” the aesthetic turn wherever it is found, Gelernter argues that “[t]he fact that software’s biggest hits are exactly the systems that are repeatedly praised for *elegance* . . . ought to be a clue to the flummoxed industry that elegance has something to do with good software, that there is a connection somewhere between aesthetics and success” (*Machine Beauty* 26). To facilitate the production of better software, Gelernter proposes dissolving educational barriers that define “computer science” as a scientific or engineering discipline and isolate it from aesthetics. “Great technology is beautiful technology,” he writes. “If we care about technology excellence, we are foolish not to train our young scientists and engineers in aesthetics, elegance, and beauty” (*Machine Beauty* 129).

removed from pure thought-stuff. He builds his castles in the air, from air, creating by exertion of the imagination.” Dependent on language rather than on machines, programmers could now easily understand their dematerialized “exertion of the imagination” as an explicitly literary form of composition.

The first true high-level programming language, FORTRAN, appeared in 1957, and was followed by literally hundreds of others. Writing in 1991, programming language historian Jean Sammet comments on the role of literary aesthetics in driving this linguistic explosion:

In my judgement, it is the personal needs and interests of people far more than the functional needs which have led to the creation of more than 1,000 languages over a 35 year period. First, there are enormous differences of personal taste on style and syntax. One need only look at COBOL and APL as representative of the extremes of readable/verbose and concise/cryptic notation to see that point, and there are strong defenders of each approach. In fact, when one looks at the rationale for creating a new language, one often finds a statement along the lines of “my new language will be easier to use or better to write or read than any of the preceding ones.” However, the new language designer generally has very little evidence to support that claim, and *in the last analysis it almost always boils down to a question of personal style and taste.* (“Programming Language History” 49, my emphasis)

It has become commonplace for programmers to comment on the “expressive” potential of different programming languages, just as literary authors might compare the expressive possibilities of French and English or of different poetic forms. Indeed, programmers have often criticized the aesthetic merit of various languages. Legendary programming theorist Edsger Dijkstra claimed that programming COBOL “cripples the mind,” and other programmers called the language “ugly” (Shneiderman 30). As a counterpoint, consider Alan Perlis describing the language Algol 60: “This language proved to be an object of stunning beauty,” he remarks. “Where Algol 58 was considered quite properly to be a draft, Algol 60 was appreciated, almost immediately, as a rounded work of art” (Wexelblat 88). For Eric Raymond, the expressive potential of LISP (an Artificial Intelligence language developed by John McCarthy in the late 1950s) was the key to the flourishing of hacker culture during the 1960s and 1970s: “LISP freed . . . hackers to think in unusual and creative ways. It was a major factor in their success, and remains one of hackerdom’s favorite languages” (11). Although the move from machine language to higher-level languages had what some programmers considered to be negative consequences—chief among them being

loss of contact with computing hardware—it allowed for the creation of a new programming sensibility. By the 1960s, the programmer’s “complex, creative art” had taken on the contours of a specifically *literary* enterprise. In his classic 1971 study, *The Psychology of Computer Programming*, Gerald Weinberg notes how poetic conceptions of code had played a crucial role in shaping the programming mentality: “At first blush, poetic language would seem to have no place in programming, yet it plays a role which should be increasingly recognized. . . . A programmer would not really be a programmer who did not at some time consider his program as an esthetic object” (209). The shift from the “complex, creative art” Backus recognized in the 1950s to the “poetic language” Weinberg identifies in the 1970s is largely a result of the move from machine and assembly code to high-level programming language.

The most extensive treatment of the programmer’s art is the multi-volume, four-decade-long project of Stanford computer scientist Donald E. Knuth: *The Art of Computer Programming*. In 1962, publishing company Addison Wesley approached the 24-year-old Knuth, then a Ph.D. candidate in mathematics and already an accomplished computer programmer, and asked him to write a book on compilers. Knuth immediately sketched out a twelve-chapter book dealing with computer programming techniques and began work—but the book quickly took on a life of its own. By 1966 he had drafted almost three thousand pages. Almost forty years and three monumental volumes later, Knuth’s magnum opus is nowhere near completion. Volume 4 of the projected seven-volume series has been underway since 1975, has itself been divided into three subvolumes, and—according to Knuth’s Web site—is tentatively scheduled to appear in 2007. Despite its unfinished state, *The Art of Computer Programming* was chosen by *American Scientist* magazine as one of the twelve most important scientific monographs of the twentieth century (Morrison and Morrison 61). The 1974 Turing Award, the 1979 National Medal of Science, and the 1996 Kyoto Prize have cemented Knuth’s reputation as one of the monumental figures in modern computer science. *The Art of Computer Programming* has achieved an almost biblical status in the programming world, mostly thanks to Knuth’s meticulous, exhaustive research into his subject matter. But Knuth’s work is not merely a definitive reference guide or the key to all computing mythologies; what distinguishes his writing is his abiding concern with the philosophy

and aesthetics of computer programming, and his oft-repeated insistence on the literary qualities of computer code. The first sentence of the first volume of *The Art of Computer Programming* reads: “The process of preparing programs for a digital computer is especially attractive, not only because it can be economically and scientifically rewarding, but because *it can be an aesthetic experience much like composing poetry or music*” (v; my emphasis). In the preface to his 1992 book *Literate Programming*, Knuth discards the musical metaphor in favor of the analogy between programming and literary production: “At first, I thought programming was primarily analogous to musical composition—to the creation of intricate patterns, which are meant to be performed. But lately I have come to realize that a far better analogy is available: Programming is best regarded as the process of creating *works of literature*, which are meant to be read” (ix). In a 1993 Open University program on BBC Radio 5, Knuth solicited submissions for a computer program written in the form of a sonnet; he has been quite frank about his hope that the Pulitzer Prize committee will one day award a Pulitzer for the best-written program (*Digital Typography* 561). If Knuth has encouraged people to envision computer code as poetry, his readers have returned the compliment: the jacket cover of the third edition of *The Art of Computer Programming* reports that “A programmer in China compared the experience [of reading *The Art of Computer Programming*] to reading a poem.” One of the achievements of *The Art of Computer Programming*—and there are many—was thus to specify the longstanding analogy between programming and artistic creation. For Knuth, writing code is a specifically literary artistic endeavor, and if there is one genre that resembles programming more than any other, that genre is poetry.

Following Knuth’s lead, historians of hacker culture have begun to emphasize how hackers have come to experience code as a form of literature, and how they have thus come to regard composing code as akin to writing poetry. For example, Neal Stephenson’s *In the Beginning . . . Was the Command Line* uses poetic imagery to explain why Unix users retain such reverence for their abstract, bare-bones operating system when most computing users have adopted fancy “user-friendly” operating systems like Macintosh and Windows. Arguing that one can come to appreciate Unix as a poetic artifact, as opposed to a commercial product, Stephenson claims that the development of the Unix operating system over the past thirty years is best understood as an

ongoing, eternally unfolding epic poem: “Unix . . . is not so much a product as it is a painstakingly compiled oral history of the hacker subculture. It is our *Gilgamesh* epic. . . . What made old epics like *Gilgamesh* so powerful and so long-lived was that they were living bodies of narrative that many people knew by heart, and told over and over again—making their own personal embellishments whenever it struck their fancy. The bad embellishments were shouted down, the good ones picked up by others, polished, improved and, over time, incorporated into the story” (88). Like *Gilgamesh*, Unix has been written and rewritten over time. It is the collective result of a history of collaborative creation, innovation, recording, and revision, one whose polished and perfected lines contain within them an entire culture’s history.

One result of this epic poetic vision of programming is a pronounced and heated disagreement between hackers and corporate software vendors about the legal status of code. The belief that code is a form of artistic expression—one that in turn “encodes” the history of its creation—is, for many programmers, also the belief that code should be open to all, that, like a poem, everyone should be free to read it, admire it, criticize it, learn from it, and even rewrite it. The literary aesthetics of code come into direct conflict, however, with the corporate practice of blackboxing, in which the “literary” high-level source code is kept secret in order to give software companies a competitive business edge. Glyn Moody expresses this conflict succinctly in his book *Rebel Code*: “Hackers rebel against the idea that the underlying source code should be withheld. For them these special texts are a new kind of literature that forms part of the common heritage of humanity: to be published, read, studied and even added to, not chained to desks in inaccessible monastic libraries for a few authorized adepts to handle reverently” (4). The conflict between corporate blackboxing and programmers who believe source code should be freely available to all is more than a mere philosophical quibble. As I will show in the next section, the debate about whether code is literature grounds one of the most economically significant legal struggles of our time.

2.2 Intellectual Property Law and the Ontology of Code

The founding principle of Knuth's writing on programming—that code has an aesthetic and literary appeal, that those who speak its languages might appreciate its merits as a lover of literature might appreciate *Paradise Lost* or *Hamlet*—issues an unexpected challenge to literary scholars, not to mention librarians. Knuth has joked about the confusion that the title of his major work has generated, noting the existence of bibliographic references to “The *Act of Computer Programming*,” and pointing out that Cornell University librarians shelved his books in its Fine Arts Library, where they “apparently sit neatly on the shelf, without being used” (*Literate Programming* 2,7). One can take a certain delight in imagining a puzzled librarian shelving a category-defying text like *The Art of Computer Programming* under “Art” rather than under “Computer Programming.” But behind Knuth's lighthearted quips lies a complex debate about the ontological status of software and the nature of the work that programmers do.

This debate finds its most immediate articulation today in the application of intellectual property law to computer programs. The central question driving the debate—whether software should be protected by copyright law, by patent law, by both, or by neither—ultimately comes down to one of ontology: what exactly *is* a computer program? Is it a machine, a process, an act of creative expression, or a hitherto unimagined amalgam of all three? In his written conclusion to *Folsom v. Marsh* (1941), Justice Story argued that intellectual property debates often hinge around such fine ontological distinctions: “Patents and copyrights approach, nearer than any other class belonging to forensic discussion to what many be called the metaphysics of the law, where distinctions are, or at least may be, very subtle and refined, and, sometimes, almost evanescent” (qtd. in Koepsell 43). Calling for detailed critical, philosophical, and legal attention to “subtle and refined” distinctions, Story's conclusion nonetheless assumes that intellectual property law rests on a solid metaphysical ground and that its system of classification is sound. Some forms of intellectual property may prove recalcitrant, Story allows, but a thorough philosophical examination will reveal their true (and, implicitly, *one* true) ontology. However, no form of intellectual property has troubled the “metaphysics of the law” and challenged the categorical

foundations of intellectual property law more than computer software.

The primary legal mechanisms for defining, categorizing, and protecting intellectual property—patent and copyright law—have long legal histories: scholars have traced the evolution of patents to medieval Italy (Heckel 68) and modern copyright law dates to Britain’s 1710 Statute of Anne (Koepsell 46). Drawing upon European precedent, the U.S. Constitution gave Congress the power “To promote the Progress of Science and useful Arts, by securing for limited Times to Authors and Inventors the exclusive Right to their respective Writings and Discoveries” (Article I, Section 8, Clause 8). Although the Constitution here amalgamates “Writings and Discoveries,” the products of literary authorship and scientific invention have historically been handled by distinct sets of laws: copyright and patent, respectively. Until relatively recently, the legal territories of copyright and patent law have been fairly clear and self-evident. Section 101 of the U.S. Patent Act defines statutory patent subject matter as follows: “Whoever invents or discovers any new or useful process, machine, manufacture, or composition of matter, or any new and useful improvement thereof, may obtain a patent therefore, subject to the conditions and requirements of this title” (qtd. in Nalley 45). Aiming to foster innovation, to advance knowledge, and to discourage enduring monopolies, patent law rewards the inventor of a useful, nonobvious, and novel invention with a limited state-sanctioned monopoly on its use. In return for this limited period of patent protection—currently twenty years—the inventor agrees to disclose the technical details of the invention. Individuals or corporations who wish to make use of an invention during its term of patent protection must in turn negotiate licensing arrangements with its inventor.

Copyright law, by contrast, protects original expression, its primary goal being to safeguard writers, artists, musicians, publishers, or other copyright holders against unauthorized duplication of their creative work. Copyright protection gives its holder exclusive rights to reproduce, publish, distribute, and sell a creative work for a set period of time.⁷ Although the Constitution specifically stipulates that intellectual property law should protect “writings,” the U.S. Copyright

⁷This limit was originally 14 years, but has varied considerably since then: it once rose to 99 years, but was scaled back to 75 years. However, the Sonny Bono Copyright Term Extension Act of 1998 extended all copyright expirations for a period of 20 years. At the time of writing, the constitutionality of this Act is being challenged in the Supreme Court in *Eldred v. Ashcroft*.

Act extends copyright protection to “original works of authorship fixed in any tangible medium of expression, now known or later developed” (qtd. in Koepsell 50). Under this provision, courts have successfully extended the scope of copyright law throughout the twentieth century to encompass emergent media of creative expression such as sound recordings, radio broadcasts, and film.

Neither copyrights nor patents can be invoked to protect ideas in and of themselves: patent law protects the *instantiation* of ideas in material inventions or processes; copyright law protects the original *expression* of ideas in original creative work. An author or inventor may not, for instance, copyright the “stream of consciousness” literary technique or patent the first law of thermodynamics; however intellectual property law can protect specific works or inventions that *embody* those techniques or laws: James Joyce’s *Ulysses*, for example, or the internal combustion engine. The specific ontological distinctions that determine patent or copyright have historically been reasonably clear: courts merely have to decide whether an entity is an invention that performs a functional, utilitarian action, or a creative work that serves an expressive, artistic purpose.

Computer programs defy such easy categorization. As written texts, computer programs are perfect examples of copyrightable expression, and yet they also have a distinct utilitarian value in that they enable computers to perform work. The very aesthetics of programming confound the distinctions between expression and functionality upon which intellectual property law revolves—in his 1974 essay “Computer Programming as an Art,” for example, Donald Knuth comments that code is most artistic when its formal and functional aspects are perfectly harmonized: “The ideal situation occurs when the things we regard as beautiful are also regarded by other people as useful” (*Literate Programming* 9). Elliot Turner Nalley summarizes the crux of the debate thus: “Computer software looks like text. It is composed of words arranged in lines written on a page (a literal page when printed out; a virtual page when viewed on a computer monitor). It is the expression of an idea and, therefore, an appropriate subject matter for protection by copyright.” However, “[c]omputer software works like a machine. It is most easily perceived as such when it behaves as a literal machine, as part of a mechanical process that performs a visible, physical act” (43). As I will show, the question of whether software is a “text” or a “machine” has crucial implications for the aesthetics of computer programming described in the first section

of this chapter. The programmer's tendency to conceptualize his work as a literary art is not merely metaphorical: there are powerful legal issues underwriting it. Those issues in turn have far-reaching implications for the prevailing tendency within literary studies to ignore the textual and aesthetic qualities of computer software. Simply put: by leaving the question of code's ontological status to lawyers and corporations, literary critics have thus far absented themselves from what may well be the most politically and economically profound literary debate of our times. I will develop this point shortly, but first I want to provide some background about the history of software's legal status.

Originally, computer programs did not enjoy any legal protection at all—nor did they need it. As I explained in the previous chapter, early computer companies did not draw clear distinctions between hardware and software, distributing operating systems and other programs freely with their computers on the premise that hardware was the valuable aspect of the system, while software was merely a component that enabled the customer to make use of the system. In 1969, IBM unbundled software from hardware, creating a large-scale corporate software economy; in the 1970s, the cost of computing hardware fell rapidly, to the point where “hobbyist” computers (the forerunners of today's personal computers) started making their way into the home. Software, once freely shared among corporations and users, became *intellectual property* during this time, and lawmakers and corporations alike began to seek ways to protect it. During the 1960s and early 1970s, some companies invoked trade secrecy law to protect their programs. Reasonably effective in the days when software companies developed and sold custom-designed programs to a limited number of corporate customers, trade secrecy law became less and less useful during the 1970s. As a burgeoning software market emerged, software vendors abandoned the goal of tailoring programs to the needs of individual clients, and with the invention of the personal computer, more and more people began using computers outside the centralized control of a corporate structure. In such a climate, trade secrecy laws could protect only the most specialized and restricted of programs.

In 1972, the Supreme Court heard and rejected its first software patent suit, *Gottschalk v. Benson*. At issue was a computer program that implemented a new method of turning decimal numbers into pure binary numbers. Robert Merges explains Justice Douglas's rationale for denying

the patent:

Justice Douglas understood algorithms implemented in computer software as pure mathematical abstractions. This made his decision in *Benson* an easy one; because algorithms are timeless abstractions plucked in their entirety from nature's library, they are unpatentable. This is, after all, the treatment that patent law gives to all abstract truths, all "scientific principles," all "products of nature." Once algorithms are equated with "pure mathematics," they are assigned to a category of discovered universal truths. (2229)

While Douglas's rationale seems conceptually flawed—as Merges points out, it is difficult to equate "a piece of software code, written in the 'C' programming language for some prosaic function such as calculating grade point averages or checking a bank balance, with the discovery of Newton's laws of gravitation, or the Pythagorean Theorem" (2229)—it set a precedent for regarding software as unpatentable material. And yet the government faced substantial corporate pressure during the 1970s to protect the product of the nascent software industry. In 1974, responding to the pressing need to find a definitive way to include software in the intellectual property matrix, Congress convened the National Commission on New Technological Uses of Copyrighted works (CONTU). The primary purpose of this commission was to study and define the nature of software, and to determine how legislature should handle this new form of property.

In 1978, after several years of hearings and discussions, CONTU published its final report. Ultimately, the commission resolved the problems of software definition very simply. In a section entitled "Foundations for the Recommendations," the CONTU report argued that computer programs, although a new form of textuality, were analogous to other forms of written communication:

Computer programs are a form of writing virtually unknown twenty-five years ago. . . . They are prepared by the careful fixation of words, phrases, numbers and other symbols in various media. The instructions that make up a program can be read, understood, and followed by a human being. (23)

Advising some necessary exceptions to standard copyright law—such as the consumer's right to "copy" a computer program by loading it into the computer's memory, and the right to make a backup duplicate version—the CONTU report recommended that Congress define computer programs as "literary works" and protect them under copyright law. In turn, the CONTU report

notes that these recommendations were in accord with decisions reached by similar commissions in other countries—both the World Intellectual Property Organization and Great Britain’s Committee to consider the Law on Copyright and Designs had drawn similar conclusions about the nature of software and the use of copyright law to protect it (27–28).

The most remarkable section of CONTU’s report, however, is the prescient dissent produced by one of its commissioners, nonfiction author John Hersey. Strenuously disagreeing with the Commission’s definitions and recommendations, he argued that “copyright is an inappropriate, as well as unnecessary, way of protecting the usable forms of computer programs” (69). While Hersey’s rationale for his position occasionally degenerates into reactionary ranting about the danger posed to humanity should governments start admitting computer programs into the category of “literary works,”⁸ his dissent, in stark contrast to Justice Douglas’s *Benson* decision, displays a remarkable understanding of the software production process and of software’s complex ontological status. Noting that “we are dealing here with an entirely new technology, one with a highly intricate multiplicity of means of fixation, of transformation, of movement from one medium (of communication) to another (of mechanical function) and back again” (81), Hersey sets out to analyze these shifts, to understand the nature of software and the compositional process behind its production, and to explain how software can best be accommodated within the matrix of intellectual property law.

Hersey’s ultimate recommendation to Congress was that “[t]he Copyright Act of 1976 should be amended to make it explicit that copyright protection does not extend to a computer program in the form in which it is capable of being used to control computer operations” (92). However, Hersey includes an important qualifying claim—software, in his view, should not be copyrightable in “*the form in which it is capable of being used to control computer operations.*” Hersey thus

⁸In one such section, Hersey refers to the “subtle dehumanizing danger . . . of the Commission’s position on programs.” He continues: “To call a machine control element a copy of a literary work flies in the face of common sense. Ask any citizen in the street whether a printed circuit in a microprocessor in the emission control of his or her car is a copy of a literary work, and see what answer you get. But if your government *tells* the citizens in the street that this is so, and makes it law, what then happens to the citizen’s sense of distinction between works that speak to the minds and senses of men and women and works that run machines—or, ultimately, the citizen’s sense of the saving distinction between human beings themselves and machines themselves” (90).

focuses his energy on the binary manifestation of computer software, which he understands as more machinic than textual, more utilitarian than expressive. However, elsewhere he admits that binary code can be understood as the utilitarian end product of expressive acts. He summarizes the software development process as follows:

In the stages of the planning and preparation of software, its creators set down their ideas in written forms, which quite obviously do communicate to human beings and may be protected by copyright with no change in the present law.

But the program itself, in its mature and usable form, is a machine control element, a mechanical device, having no purpose beyond being engaged in a computer to perform mechanical work.

The stages of development of a program usually are: a definition, in eye-legible form, of the program's task or function; a description; a listing of the program's steps and/or their expression in flow charts; the translation of these steps into a "source code," often written in a high level programming language such as FORTRAN or COBOL; the transformation of this source code within the computer, through intervention of a so-called compiler or assembler program, into an "object code." This last is most often physically embodied, in the present state of technology, in punched cards, magnetic disks, magnetic tape, or silicon chips – its mechanical phase. (70)

Object code, or what Hersey elsewhere calls the "mature program" (70), is the result of compiling high-level language into binary format. While Hersey is willing to concede that the earlier, "written forms" of software that "communicate to human beings" can be considered acts of expression, he identifies an ontological shift at some point during the process of planning, writing, and compiling software, a point where code loses its expressive power and becomes "a machine control element, a mechanical device." Explaining the alienation of expressive act from functional end-product, Hersey comments that "the 'writing' of the author is spent in the labor of the machine" (73). What is needed, however, is "an appropriate definition of the cutoff point, the point at which a program ceases being a copyrightable writing and becomes an uncopyrightable mechanical device" (92). Hersey leaves this crucial moment of transition undefined. Having argued that binary code cannot constitute an expressive form, but that earlier, less "mature" forms of computer programs *can* constitute "copyrightable writing," Hersey's dissent, for all its stridency, creates more ambiguity and indeterminacy than it resolves. In amending the U.S. Copyright Act in 1980, Congress followed CONTU's main guidelines rather than Hersey's dissent; it was thus that CONTU's central

recommendation—that computer programs should be defined legally as “literary works”—became law.

Despite Congress’s decision, the debate over software’s ontological status has continued unabated for more than twenty years,⁹ with the positions of Justice Douglas (that computer programs are “laws of nature”), of John Hersey (that “mature” computer programs are machines), and of the remaining CONTU commissioners (that computer programs are copyrightable expression) shaping a fierce and heated debate. Underpinning the debate, naturally, are economic considerations about safeguarding revenue in what has become a multi-billion dollar industry. Although the 1980 Copyright Act amendment made programmers into authors and guaranteed that their work could not be copied without permission, the amendment did nothing to protect the processes or functions contained within computer programs. Companies thus found themselves vulnerable to predatory “reverse engineering”—another company could develop and market a version of the same product as long as it wrote its own code for the job. It was thus that corporations continued to press for software patents as a way to define and delimit software.

In *Diamond v. Diehr* (1981), the Supreme Court granted a patent on a new process for curing synthetic rubber. As unlikely as it sounds, this was a landmark case in software patent history: a software program was part of the rubber curing process, and it, too, was protected by the patent. In its ruling, the Supreme Court distinguished between pure algorithms and industrial processes employing those algorithms:

[T]he respondents here do not seek to patent a mathematical formula. Instead, they seek patent protection for a process of curing synthetic rubber. Their process admittedly employs a well-known mathematical equation, but they do not seek to preempt the use of that equation. (Qtd. in Koepsell 64)

While this seems fairly innocuous, this decision had far-reaching consequences. In order to enhance their chances of gaining software patents, lawyers after *Diamond v. Diehr* drafted patents so as to emphasize the role of hardware and underlying processes (Merges 2230). In turn, both the

⁹My discussion here centers on debates within the United States. In most countries in the world, the confusions caused by software’s dual status as text and machine have not been made the basis for legal debate. In Europe and in Australia, for example, software is afforded precisely the same legal status as literary texts.

Patent Office and the Federal Circuit Court have upheld software patents on the most tenuous of arguments. The Patent Office has even approved patent claims on the basis that computer programs can be stored on magnetic storage disks, the patentable material being the particular “composition of [electronic] matter” on the disk. Without directly challenging the Supreme Court or the 1980 Copyright Act amendment, then, the Federal Circuit Court and the Patent Office have gradually eroded legal precedent and turned computer programs into legitimate patentable material. The trickle of patent applications received prior to 1980 turned into a flood in the wake of *Diamond v. Diehr*: Lawrence Lessig notes that software-related patent applications rose from 250 in 1980 to 21,000 in 1999 (*Future of Ideas* 208).

In the last decade, the courts have extended patent protection to business practices involving software—in a 1993 patent dispute over a financial data processing system, the U.S. Court of Appeals declared that “as ‘machine’ or ‘process’ [software is] proper statutory subject matter” (Nalley 45). As a result of the e-commerce boom, patent applications for software-related business practices more than doubled between 1997 and 1999 (Lessig, *Future of Ideas* 208). Moreover, the Patent Office has granted patents for innovations that, to the minds of many, do not fall into the category of “nonobvious” innovation; Garfinkel *et. al.* comment that “patents have been granted for ideas so elementary that they could have been answers to problems in a first-year programming course” (36). For example, Internet retailer Amazon.com secured a notorious patent for its “1-Click” ordering function, which not only makes shopping at Amazon.com easier and more pleasant, but ensures that shopping elsewhere online will never be so simple—unless other Web sites are prepared to license the technology from Amazon.com. Other companies that wish to use one-click ordering (such as Apple Computer in its on-line Apple Store) must first license it from Amazon.com. What Amazon.com has patented is not the software, but the business process *underlying* the software; therefore, any independently-developed competing one-click ordering software package infringes upon Amazon.com’s legal rights.

For Lawrence Lessig, one aspect of software intellectual property law is particularly ironic: “What was most striking about this explosion of law regulating innovation,” he writes, “was that the putative beneficiaries of this regulation—coders—were fairly uniformly against it”

(*Future of Ideas* 208). Indeed programmers have been so deeply concerned by the implications of copyright and patent law that they have formed grassroots organizations—such as the Free Software Foundation (described in Chapter One) and the League for Programming Freedom—to oppose the very notion of software as proprietary intellectual property. So serious is Richard Stallman on this point that in 1985 he employed legal experts to create a special license that would help protect and perpetuate free software. Known as the General Public License, or GPL, this new license embraced what Stallman calls the principle of “copyleft,” or “all rights reversed.” When a programmer licenses code under the GPL, anyone can use, modify, and redistribute that code—so long as the improved code is also licensed under the GPL. Any larger project that incorporates GPL’d code must itself be re-released under the GPL. “Copyleft” is thus Stallman’s way of ensuring a perpetual link between free software and the ideals of individual liberty that underpin the Free Software Foundation’s work: “The idea of GNU is to make it possible for people to do things with their computers without accepting [the] domination of somebody else, without letting some owner of software say, ‘I won’t let you understand how this works; I’m going to keep you helplessly dependent on me and if you share with your friends I’ll call you a pirate and put you in jail’” (qtd. in Moody 28). By licensing under “copyleft,” the programmer surrenders his creation to the larger body, immersing it in a collaborative environment. Other programmers can read the code and change it (as Raymond puts it, “anyone can hack anything” [87].) Copyleft thus anticipates and even requires a norm of collective authorship. No piece of open source code is ever single-authored: though something may begin as one person’s work—the original versions of Stallman’s Emacs editor and Torvalds’s Linux kernel are classic examples—it is never meant to *remain* one person’s work. Indeed, submitting one’s work to the open-source community ensures that one’s work will become the work of many within short order. As bugs are found and fixes are posted, the list of contributors to the project grows (this list is recorded as part of the code’s “history”), and soon many authors are writing a work that began traditionally as the brainchild of one.¹⁰

¹⁰Programmers have a clever way of retaining a sense of origin within this collaborative climate. As Raymond notes, even though open source projects are collaboratively written, and even though they are not owned in the way property, or even intellectual property, is owned, everyone still knows

The cycles in software's definition tell us a great deal about how economic and political considerations underpin ontological debates about the status of computing code. Those debates have also had a crucial impact upon the concepts of software authorship described in the first section of this chapter. If a piece of software is copyrighted, the actual texts of the source code and object code are protected by copyright law. However, as we have seen, there is nothing in copyright law to prevent another programmer or company from writing new source code to achieve the same effect—just as there is nothing to prevent a literary author from imitating the stylistic effects of Virginia Woolf or Toni Morrison. Patent law, by contrast, protects the *utilitarian* aspects of software, meaning that any code that duplicates the operations of a patented program violates the software patent. Although software activists like Stallman oppose copyright as a matter of principle, they see patents as by far the more dangerous form of intellectual property protection because they can strip away the programmer's right to expression. For the last two decades, American courts and the Patent Office have privileged large corporations' vested interest in securing software patents over programmers' traditions of authorship and code sharing. As the courts have struggled to comprehend the confused and difficult ontology of computing code, programming traditions and attempts at authorial self-definition have been all but ignored.

Literary critics could conceivably have a vested interest in such debates—after all, who knows more about what constitutes “the literary,” and who is better trained to argue for an expanded

who the owner really is. There are two kinds of owners. Founders of open source projects are owners: Torvalds and Eric Allman own Linux and Sendmail, even though thousands of people have helped author those programs. Inheritors and adopters of open source projects can become owners: if a founder decides he can no longer devote the time and energy necessary to maintain a project, he hands the project over to someone who can run things as well or better than he can. In *The Cathedral and the Bazaar*, Raymond explains how he inherited a POP3 program called popclient from Carl Hacker, and developed it into a much more sophisticated multi-protocol mail-retrieval program called fetchmail. Ownership, in this model, is distinct from authorship, though closely tied to it. It names a proprietary relationship, but that relationship is not oriented around concepts of economic value or the primacy of the creator, so much as it is around an ideal of management. The owner of an open source project is the person who coordinates its continual revision, the editor-in-chief of hundreds, even thousands, of contributing editors. A founder is an owner only as long as he facilitates the communal effort to perfect his creation. Torvalds is still considered to be the owner of Linux because he has spent the last ten years vetting and implementing improvements on the operating system he began in 1991.

literary canon and a flexible concept of textuality? However, literary critics have largely absented themselves from the legal furor over software. So far, indeed, are they from accepting it that they have yet to conceive of it—despite the fact that software has been defined under the law as a “literary” form for more than two decades.¹¹ It is an irony of modern criticism that literary scholars are now prepared to read just about any non-textual thing as a textual construct—bodies, clothes, and even garbage have all fallen under the critic’s purview. An even greater irony is that when the almost uniformly left-wing literary critical culture neglects to consider software as a textual entity shaped by real material conditions and situated in actual political contexts, it becomes complicit with a corporate agenda that would deprive software of textual status in order to secure prohibitive patents. In the next three sections, I want to look further at code through the lens of a literary critical sensibility, examining three moments in computing culture when a literary, literate idea of code has had powerful practical implications for computing: the tendency of computer programmers to learn their craft by doing essentially New Critical analyses of code; the move toward a “literate programming” methodology that expressly defines writing code as writing literature; and the trend among Perl programmers to combine English and the Perl programming language into hybrid texts that read as poems at the same time that they compile as programs.

¹¹In his essay “There Is No Software,” Friedrich Kittler comments on the nature of programming languages; however, he does so in order to argue *against* a privileged concept of “literary” language: “Programming languages have eroded the monopoly of ordinary language and grown into a new hierarchy of their own. This postmodern Tower of Babel reaches from simple operation codes whose linguistic extension is still a hardware configuration, passing through an assembler whose extension is this very opcode, up to high-level programming languages whose extension is that very assembler” (82). Although Kittler grasps the essential concept of abstraction that underpins programming theory, his essay lacks any cultural or aesthetic understanding of programming practice. By highlighting those factors, this chapter has attempted to argue against Kittler’s repeated insistence that programming languages—along with other forms of new media—are helping to erode language as a phonetic and graphic entity. Within programming culture itself, the movement has been quite the opposite to the one Kittler emphasizes: programmers understand programming not as a deconstruction of language but as a contribution to literary history.

2.3 Long Close Readings: Literary Criticism and Code

The first section of this chapter argued that evolutions in computing architecture and programming methodology during the 1950s and 1960s allowed programmers to conceptualize writing code as a literary endeavor. During the 1970s, a new dimension was added to programmers' sense of code as a complex art: it was then that computer scientists began to think of code as something that could be profitably read as one would read a work of literature, as something whose form and content are best revealed and appreciated by way of detailed line-by-line exegesis. Formalist analysis of code became a pedagogical norm during the 1970s: computer science professors began putting source code on their course reading lists, students of computer science began studying source code on their own and in reading groups, and computer science professionals began reading source code to improve their programming skills.

Close reading code became central to computing culture during these years; it was both a useful training technique and an aesthetically pleasing experience, a way of learning one's craft that centered—like so much artistic schooling—on the patient study and scrupulous imitation of the masters. So central has close reading code become to the field—to its sense of artistry, of history, of passionate commitment—that it has indeed become a cornerstone of contemporary computing culture. This section discusses how the concept of close reading entered the field of computing, focusing in particular on the code that made the concept possible: the Unix operating system. Without Unix, there would be no concept of code as a literary text that can be profitably read, explicated, and appreciated; without Unix, there would be no “art of code.”

Unix was the first operating system compact enough to be understood and mastered by a single individual. Written in 1969 by Ken Thompson and Dennis Ritchie of Bell Labs, Unix was surprisingly accessible. At just under 10,000 lines, it was short—earlier operating systems consisted of millions of lines of code. Written in the high-level programming language C, it was intelligible—earlier operating systems were written in the far less comprehensible assembly language. As such, Unix definitively changed both the shape of programming (by raising it to a higher standard of efficiency) and the shape of programming pedagogy (by lending itself to

close reading). As Unix programmer Peter Reintjes recalls, “We had been taught previously that an operating system was something only to be undertaken by an army of programmers. Once written, it would not admit analysis, composed as it was of millions of lines of assembly language instructions. Happily, Ken Thompson and Dennis Ritchie proved this wrong, producing the kernel for the Sixth Edition of Unix in less than 10,000 lines of code that would not merely admit, but invite, analysis” (qtd. in Lions).

Thompson and Ritchie made that invitation to analysis explicit in 1974, when they publicized their creation in a now-classic article entitled “The Unix Time-Sharing System.” Presenting Unix as a “general-purpose, multi-user, interactive operating system,” they demonstrate its applicability to real-world situations such as “the preparation and formatting of patent applications and other textual material, the collecting and processing of trouble data from various switching machines within the Bell System, and recording and checking telephone orders.” However, in describing Unix as “an essentially personal effort,” Thompson and Ritchie take pains to distance their creation from operating systems written under corporate agendas. In fact, their paper contains a subtle critique of the managerial technocracy that supposedly existed to give order and direction to the programming enterprise. “Perhaps paradoxically,” Thompson and Ritchie write, “the success of Unix is largely due to the fact that it was not designed to meet any predefined objectives” (373). Unlike impossibly complex and monolithic operating systems like IBM’s disastrous OS/360, Unix was the product of programming experimentation, flexibility, and creativity: “Our goals throughout the effort, when articulated at all, have always concerned themselves with building a comfortable relationship with the machine and with exploring ideas and inventions in operating systems” (374). Thompson and Ritchie hope that once users grasp the Unix design philosophy, they “will find that the most important characteristics of the system are its simplicity, elegance, and ease of use” (365).

Ritchie and Thompson stress how important the collaborative development model was in shaping Unix, noting that “Since all source programs were always available and easily modified on-line, we were willing to revise and rewrite the system and its software when new ideas were invented, discovered, or suggested by others” (374). Indeed, Unix’s openness was another key to its success. Written in C, the system was not inextricably bound to one hardware architecture, and

by 1974 it had been ported to several different types of computer, thus giving it a huge temporal and lateral advantage over operating systems that had been custom designed to work on specific machines. This not only meant that Unix could be used by all programmers in the present, but that it would not become obsolete when current technology was displaced by more advanced hardware. In short, Unix was built to be flexible, to work anywhere, anytime, on just about anything. Unix was built to last—and it has.

The article inspired programmers around the world to write to the authors for a copy of the operating system, which at the time included the source code. People flocked to Unix. They saw its potential to become the dominant operating system of the future, and they saw that their own professional futures would most likely depend on knowing Unix. And so programmers around the world embarked on a careful, deep study of the Unix source code. It was the first time in computing history that hackers could sit down and read an entire operating system to see how it worked, and it was also the first time they could follow another programmer's complete train of thought. The activity was irresistible, educational and pleasurable at once. Unix quickly became the new lingua franca of programming culture: within a year or two, an entire culture had grown up around the Unix source code. That culture was a literary culture, and its principal activity was close reading code.

The literary culture of reading code began at Berkeley when Ken Thompson spent a sabbatical there in 1975. Thompson, who graduated from Berkeley in 1966 with a degree in electrical engineering, found in the newly formed West Coast Unix User's Group a number of programmers, computer science students, and faculty who were eager to study the code he and Ritchie had written. Thompson responded to that eagerness by forming an ad hoc UNIX reading group. Over the course of a week, the group met to read Unix source code under Thompson's guidance. Bob Fabry, a Berkeley computer science professor who attended Thompson's reading, remembers the sessions as transformative: "We all sat around in Cory Hall and Ken Thompson read code with us. We went through the kernel line by line in a series of evening meetings; he just explained what everything did ... It was wonderful."¹² Eric Allman, author of the Sendmail open-source mail

¹²Andrew Leonard describes Thompson's Berkeley readings in his online "Free Software

transfer program (a program that handles the vast majority of e-mail sent across the Internet today), was a Berkeley undergraduate at the time. He attended Thompson's nightly code readings and still treasures the old marked-up printouts of Unix code where he had scribbled notes from Thompson's reading.

The ad hoc reading-group-style approach to learning Unix was formalized in 1976 by John Lions, an Australian professor of computer science at the University of New South Wales. Like so many other programmers around the world, Lions was thoroughly taken by Ritchie and Thompson's article. He urged his colleagues in the Computer Science Department to apply for a Unix license, thinking that switching to Unix would give them better connectivity and overall performance than they had with their current systems. When the copy of Unix arrived, Lions read the code as a matter of course. Shortly afterward, an unlikely bond between Unix and literary criticism was forged.

Lions had long felt that it was necessary to read good code if one was to learn to write good code. Unix, whose own authors stressed the importance of collaboration and openness, offered a perfect opportunity to implement his belief. As I have explained above, Unix was the first operating system that one person could read all the way through with the aim of fully grasping every component and of seeing how the whole was put together. It was also exceptionally well-written, and contained a great deal of virtuoso programming as well as many masterful solutions to common programming problems. Lions had sent away for Unix with the idea of possibly converting the New South Wales system over to the newer, more powerful and more flexible operating system. When Unix came, it served that need and more. Lions jumped at the opportunity to have his students study the new and promisingly robust Unix operating system. He began assigning sections of Unix source code to his students, and Unix quickly became the centerpiece of his pedagogy.

What Lions was proposing with his Unix reading assignments was an essentially literary approach to programming. Just as an aspiring writer benefits from reading prose that is far more

Project"—see http://www.salon.com/tech/fsp/2000/05/16/chapter_2_part_one/index1.html

accomplished than anything he is capable of producing, so the student of programming, Lions reasoned, would gain a sense of form and style from humble, attentive study of masters such as Ritchie and Thompson. According to Greg Rose, one of Lions's former students, his professor "felt that all of the other courses making up the computer science degree program at the time, with the exception of the hardware course, involved teaching the students to write and debug programs. No other course required the students to be able to READ programs. . . . With biting sarcasm, John expressed this by saying 'the only other big programs they see were written by them; at least this one is written well'" (Lions). Such was Lions's belief in the pedagogical powers of good code that he envisioned his students reading their way beyond mere appreciation to technical mastery. He not only required them to study the Unix source code closely, but encouraged them to find—and even fix—the few awkward routines and occasional bugs that marred an otherwise near-perfect text (Lions's hope was optimistic to say the least—another former student who is now a Unix programmer confesses that she and her peers didn't find nearly as many bugs as Lions thought they should).

Lions was serious enough about his vision that he wrote a textbook to help it along, and in 1977 he made his *Commentary on the Unix Operating System* available to the world by publishing an ad for it in *Unix News*. Programmers and computing science students around the world read the ad, mailed Lions a check, and proceeded to have both their studying technique and their appreciation of what it meant to write code revolutionized. Lions's book consisted of extensive annotations to the Unix source code, along with periodic assignments to the reader. The commentary was explicitly cast as an exercise in reading comprehension, aesthetic appreciation, and authorial training, and was sold along with a companion volume containing the code itself. A student of Unix could thus open both brightly bound orange books at once, lay them side by side, and study the code while absorbing Lions's notes on it—a method familiar to anyone who has ever read, for example, Joyce's *Ulysses* alongside Gifford's annotations. The literary quality of this style of exegesis was noted from the outset: as the programmer Peter Reintjes put it, "We had acquired what amounted to a literary criticism of computer software" (Lions).

Reintjes does not exaggerate. Lions explicitly casts programmers as writers, referring to

Thompson and Ritchie as the “authors” of Unix, and he just as explicitly casts their code as an aesthetic achievement worthy of imitation. Observing that Unix is written to a “standard of near perfection,” he tells his reader that “Ken Thompson and Dennis Ritchie have created a program of great strength, integrity and effectiveness, which you should admire and seek to emulate.” And he frames the study of great code as an elite form of literary study that is itself a creative act: “Reading other people’s programs is an art which should be learnt and practiced—because it is useful!” Lions leaves no doubt that he envisions his commentary as an exegetical aid to the essentially literary work of learning, and learning to appreciate, the art of code.

Most practically, Lions’s book allowed programming students to see how two top programmers handled the technical aspects of writing complicated working code. As Berny Goodheart of Tandem Computers, Sydney, explains,

It is important to understand the significance of John’s work at that time: for students studying computer science in the 1970s, complex issues such as process scheduling, security synchronization, file systems and other concepts were beyond normal comprehension and were extremely difficult to teach—there simply wasn’t anything available with enough accessibility for students to use as a case study. Instead, a student’s discipline in computer science was earned by punching holes in cards, collecting fan-fold paper printouts, and so on. Basically, a computer operating system in that era was considered to be a huge chunk of inaccessible proprietary code. Unix changed all this. (Lions)

Peter Reintjes echoes this sentiment, stressing how valuable it was to be able to study a real, working program: Unix, he recalls, gave “students the chance to read production code written by excellent practitioners. It [was] first and foremost an opportunity to peer over the shoulders of Ken Thompson and Dennis Ritchie and see them at work. And it must be stressed that this was production code; the Sixth Edition of Unix was used for many applications in addition to computer science research. Textbook examples of code may be elegant, but they often ignore difficult aspects of real world programming, such as error- and interrupt-handling.” In Unix, Reintjes continued, “we had discovered the first piece of software that would inspire rather than annoy us . . . we were making the single most significant advancement of our education in computer science by actually reading an entire operating system.” As these testimonials suggest, the practical benefits of Lions’s commentary were always also aesthetic ones. The elegance of Unix was not the sterile elegance of

perfectly crafted textbook examples, but was instead the elegance of a program that managed to be both beautifully shaped and entirely pragmatic. The beauty of Unix's "10,000 lines of elegantly written code" (the words are programmer Mike O'Dell's) is the beauty of code that works as well as it looks.

Hence the enormous pleasure programmers experienced upon reading Unix with the aid of Lions's commentary. Michael Tilson, president of UniForum Association and CIO of the Santa Cruz Operation, Inc. remembers his first encounter with the Lions book vividly: "When this book was first published, I was astonished by how much pleasure I got from reading what should have been a dry piece of technical documentation. John Lions had created a truly brilliant technical work. The Unix operating system kernel code was itself an elegant work, and even today it remains worthy of study. John added a line-by-line analysis that was equally elegant. The source code and the annotations were perfectly suited to each other, and I haven't seen anything to equal this achievement since" (ix). Mike O'Dell, Vice President and Chief Scientist at UUNET, recounts a similarly breathtaking reading experience: "John Lions's marvelous exegesis revealed the inner beauty that Ken and Dennis set down. His commentary was spare but incisive, mirroring the shape of the code, filling in the curiosity just as the questions formed in my mind." For both programmers, Lions's ability to bring out the artistry of Unix was a work of art in and of itself.

Neither the power of Unix nor that of Lions's commentary has diminished over time. Dennis Ritchie, one of the original authors of Unix, explains in his introduction to the current edition of the Lions commentary that

The material in this monograph is indeed dated. You will not find here anything about graphics, about networks, about anything that's happened since 1975. You will find linear searches, primitive data structures, C code that wouldn't compile in 1979 let alone today, and an orientation towards a machine that's little more than a memory. You will see signs of sloppiness and naivete. But you will also see in the code an underlying structure that has lasted for a long time and has managed to accommodate vast changes in the computing environment. (Lions)

This underlying structure—what Mike O'Dell calls "the soul of the system" (Lions)—has been the basis for the continuing popularity of Lions's commentary on Unix version 6.

Programmers agree that even as Unix has evolved, both its original expression and Lions's original exegesis of that expression are incomparable and timeless achievements. Peter H. Salus, former executive director of Usenix and the Sun User Group, notes that the Lions book is as useful today as it was when it was first written—despite the fact that Unix has changed a great deal since then: “The code is now out of date. As much of the commentators note, the comments are not. To learn about operating systems, one must read and understand code” (Lions). Moreover, as Mike O'Dell observes, the commentary is useful for appreciating the beauty of coding language itself, which to his nostalgic eye was more mellifluous back then, more capable of expressing the programmer's meaning with nuance, economy, and grace: “I looked back at my original copies not long ago and marveled at how much things have changed, not all for the better. The C language back then was a much more graceful language. It lay on the page with a grace and beauty later bespoilt by the nattering legalisms of casts and politically-correct type definitions. The modern C language, for all its improved utility and portability, seems to have become mostly points and sharp edges, where the old language was flowing soft curves, molding itself around the concepts with a marvelous economy” (Lions). O'Dell talks about programming language as if it were the language of poetry. Both men discuss the commentary as a precious document in code's literary history.

Among today's literary theorists, explication and appreciation are considered to be the least demanding, least useful, and least prestigious of critical methodologies. At best, they are associated with the antiquated work of former critical giants; at worst, with the bad habits of undergraduate English majors. Either way, it is understood among contemporary literary scholars that “proper” literary criticism does more than explicate and appreciate a text. So strong is this belief that in recent years, “the literary” has all but disappeared from literary criticism, which either makes literature a means of doing social or political analysis or ignores it altogether in favor of a variety of “cultural” or “material” texts. In computing culture, however, the unfashionable methodologies of New Criticism have found a practical purpose and even a professional prestige that they no longer enjoy—no longer *can* enjoy—in their discipline of origin. Among Unix programmers, explication is a tough, difficult, precise art (for programmers, the challenge is to compress as much meaning into a line of code as possible; explicating those lines is to decompress

and translate at once). Likewise, among programmers, appreciation is a specialized, acquired skill (to see a line of code as beautiful, one must be able to see why, of all the ways the line could have been written, this way is the most compact, the most effective, and hence the most elegant).

It is for this reason that Unix's own authors have come to see Lions' commentary as a foundational moment in the history of programming. A text that both brought out the beauty of the Unix code and modeled the equally artful discipline of close reading, the commentary was, according to Dennis Ritchie, "The document" that "educated a generation" (Lions). As Ken Thompson puts it, Lions's commentary is quite simply "the best exposition of the workings of a 'real' operating system" (Lions). Thompson and Ritchie's language is telling; it acknowledges that in writing an "exposition" of Unix—in writing a kind of extended, annotated plot summary of the program—Lions's made it possible for others to understand Unix, to use it, and, crucially, to keep writing it.

Improving on existing code has always been a central component of hacker culture. A good programmer writes good code; a great programmer makes already good code better. Lions's commentary explicitly sought to train its readers to locate places where a program could be improved, and to take the time to write a tighter, more elegant alternative script. For example, he tells the reader to rewrite the code for the procedures "malloc" and "mfree": "The code for these two procedures has been written very tightly. There is little, if any, 'fat' which could be removed to improve runtime efficiency. However it would be possible to write these procedures in a more transparent fashion," he notes. "If you feel strongly on this point, then as an exercise, you should rewrite 'mfree' to make its function more easily discernible." A central lesson of Lions's annotations, something he performs in his own notes and prods his readers to apply on their own, is that there is no end to either the reading or the writing of a good program. His conclusion acknowledges the eternally unfolding character of the programming odyssey in a mock epic tone that perfectly captures the hacker's spirit of serious, unending play: "Now that you, oh long-suffering, exhausted reader have reached this point, you will have no trouble in disposing of the last remaining file, 'mem.c' (Sheet 90). And on this note, we end this discussion of the Unix Operating System Source Code," he writes. "Of course there are lots more device drivers for your

patient examination, and in truth the whole Unix Time-Sharing System Source Code has hardly been scratched. So this is not really THE END.”

Lions’s was the exposition that kept on giving, that not only made it possible to learn Unix effectively and efficiently, but also provided an impetus for programmers to revise and improve the operating system. In other words, Lions’s literary approach to Unix made it one of the earliest open source movements, a collaboratively authored effort that over time built upon the original work of Ritchie and Thompson. Greg Rose thus describes Lions’s book as an education not, finally in imitation, but in moving beyond imitation to innovation. As more and more programmers read Lions’s commentary and took to heart his encouragement to improve its already elegant code, he observes, “Enhancements from institutions around the world began to be exchanged, and contributed in no small way to the growth of Unix . . . These two volumes made it far easier to get started with this kind of experimentation, and contributed greatly to the success of Unix during the late 1970s and early 1980s.”

The vitality of this collaborative critical practice, this culture of innovative and revisionary close reading, became eminently clear when the law made it impossible to continue sharing the Unix source code freely. In 1979, ten years after Unix was written, Unix Version 7 was released with the proviso that henceforth the source code underlying the operating system could not be released to anyone—not even to computing science students for teaching purposes. The effect on computer science education was devastating and profound. As Andrew Tanenbaum, a professor of computer science at the Free University of Amsterdam, recalls, “Between 1979 and 1984, I stopped teaching practical Unix at all and went back to theory” (Moody 33). A terrible irony of the situation was that it had been brought about by the Lions commentary itself. According to Dennis Ritchie, the commentary was just too good: “The very value and vividness of the Lions commentary compelled caution, and so the license for the 7th Edition of Unix 1977 forbade using its source code as teaching material” (Lions). Western Electric, eager to protect its commercial interests, effectively crippled computer science education; now that Lions’s commentary had made the source code for Unix comprehensible, the company was no longer willing to continue its tradition of releasing a copy of the code to each licensee. The “blackboxing” of Unix stalled

computer science education rather the way a liberal arts education would be stalled if students had to study literature and art history by reading theories about representation rather than by studying works of art themselves.

Nonetheless, the culture of creative license that had grown up around Unix continued to thrive. Although it was now technically illegal to circulate, to reproduce, or to teach from Lions's book, the commentary continued to make the rounds of the programming world in the form of second, third, fourth, even fifth generation photocopies. Everyone had a copy, though few had bound copies of the original, and people studied it just as they always had—except that now they did it on the sly. Pirate photocopies circulated in the Unix community right up until 1996, when the book was finally officially published with AT&T's blessing (AT&T now owns what used to be Western Electric). Bell Labs continued to use the commentary internally along with later Lions commentaries on later versions of Unix (the people at Bell liked Lions's work so much that they brought him in for a year, during which time he did a lot of annotation-related work for them). It is no exaggeration to say that nearly every aspiring Unix programmer since the late 1970s has studied Lions's annotations and learned from them, even long after the code itself was obsolete. Today the Lions commentary is affectionately known as “the most famous suppressed manuscript in computer history.”

The blackboxing of Unix, and the accompanying suppression of Lions's commentary, also inspired a number of programmers to write Unix clones of their own. Richard Stallman's GNU project, founded in the mid-1980s, aimed to produce an entire Unix-like operating system from scratch in order to release the source code and so recreate the vital collaborative culture that had grown up around Unix (GNU stands for “GNU is Not Unix”). Meanwhile, in Amsterdam, Andrew Tanenbaum, desperate to find a way to continue to employ a pedagogy centered on close reading code, simply sat down and wrote his own miniature Unix—which he dubbed “Minix.” Tanenbaum used Minix in his classroom the way he had formerly used Unix, and, following in Lions's footsteps, in 1987 he distributed it along with his textbook *Operating Systems: Design and Implementation*. Tanenbaum's gesture of pedagogical desperation in turn had spectacular—if unexpected—recuperative effects: a young Linus Torvalds studied Tanenbaum's book in university, using it as what he termed the “scaffolding” for the operating system that would become the

open-source GNU/Linux. This system in turn has enabled the continuation of pedagogical models inspired by Lions. Consider, for example, Daniel Bovet and Marco Cesati, two professors of computer science whose teaching practices make close textual analysis an integral aspect of university courses on operating systems. In their recent book *Understanding the Linux Kernel*, Bovet and Cesati write that “In the spring semester of 1997, we taught a course on operating systems based on Linux 2.0. The idea was to encourage students to read the source code” (xi). Bovet and Cesati wrote *Understanding the Linux Kernel* to encourage close textual engagement with Linux kernel code itself—an engagement that gets its authority from a clear historical sense of where the code came from and how it has evolved over time. “The book will give valuable insights to people who want to know more about the critical design decisions in a modern operating system,” they write. This background in turn makes the text of the code all the more meaningful. The code is more accessible: “Our work might be considered a guided tour of the Linux kernel: most of the significant data structures and many algorithms and programming tricks used in the kernel are discussed; in many cases, the relevant fragments of code are discussed line by line” (xi). The thought behind the code is more significant: “It is not specifically addressed to system administrators or programmers; it is mostly for people who want to understand how things really work inside the machine! Like any good guide, we try to go beyond superficial features. We offer background, such as the history of major features and the reasons they were used” (xii). And the training it offers is thus more profound, at once an education in programming and a socialization into a culture: “All people curious about how Linux works and why it is so efficient will find answers here. After reading this book, you will find your way through the many thousands of lines of code, distinguishing between crucial data structures and secondary ones—in short, becoming a true Linux hacker” (xi). Bovet and Cesati thus seek to make the workings of the kernel accessible to the non-specialist, and to give a historical, contextualized interpretation of the kernel. Aiming to impart a sense of programming culture and authorship through a close reading of the texts computer programmers have produced, they essentially ask the reader to take a journey through their book; to become immersed in what may seem an alien, foreign environment; but, with the help of careful exegesis, to come to understand the Linux kernel code as having a background, a

history, authors, and a surrounding culture.

The story of Unix's evolution into Linux is thus the story of how a literary ethic of coding created and continues to create a great deal of continuity within programming culture across time, space, and platform. Together, Ritchie, Thompson, and Lions created a distinctly literary, literate computer science pedagogy in which programmers learn to code by becoming literary critics of code. The art of code is explicitly framed by programmers in the terms that we associate with new criticism: to read a program properly, one must not only be able to discern what the program does, but must also be able to judge how well it does what it does. A program's merit is assessed much the way a poem's would be, in terms of structure, elegance, and formal unity.

2.4 Literate Programming

Shortly after copyright restrictions forced the Lions commentary underground, Stanford computer scientist Donald Knuth began to advocate a style of programming that had a great deal in common with Lions's annotative technique. Called "literate programming," Knuth's concept sought to improve programming by promoting an expressly literary approach to writing code. In a 1984 essay Knuth defines literate programming as "an attempt to make further progress in the state of the [programming] art" (*Literate Programming* 99), one that aims to increase what we might call "code literacy" by having programmers incorporate into their code explanations of what they are doing and why. For Knuth, the key to literate programming lies in a literary approach to programming: "I believe that the time is ripe for significantly better documentation of programs, and that we can best achieve this by considering programs to be *works of literature*" (*Literate Programming* 99). Knuth is serious about his analogy, and develops it at length:

The practitioner of literate programming can be regarded as an essayist, whose main concern is with exposition and excellence of style. Such an author, with thesaurus in hand, chooses the names of variables carefully and explains what each variable means. He or she strives for a program that is comprehensible because its concepts have been introduced in an order that is best for human understanding, using a mixture of formal and informal methods that nicely reinforce each other. (*Literate Programming* 99)

For Knuth, the best code is written not from the pragmatic perspective of an engineer, but from the artistic perspective of an author. Economy of style, clarity of expression, and formal elegance are as essential to good programming as they are to good writing.

Essentially advocating that all code be written the way Lions wrote about Unix, Knuth transforms Lions's model for comprehending code (supplementing a primary text—a program—with a secondary guidebook) into the model for writing code in the first place. In literate programming, the analysis, the annotation, and the exegesis of code are inextricable from its composition. Generalizing to all programming the techniques of reading and commenting that had become so central to Unix culture, Knuth thus raised a pedagogical strategy to the level of theory. Knuth's essential concept was to focus not on the computer but on the human reader: instead of writing source code and interspersing occasional secondary comments in the program listing, literate programmers concentrate on explaining their programs and inserting code as part of that description. In redefining the audience for computer programs, Knuth also redefined programming philosophy. What Unix programmers saw as a useful technique for learning to program—sharing, reading and annotating code—became, in Knuth's hands, an entire programming methodology, one whose operative rationale is that writing code ought always to be understood as a literary art.

Knuth followed this strategy when he wrote $\text{T}_{\text{E}}\text{X}$ and METAFONT, two programs that together comprised his digital typesetting program. Publishing the $\text{T}_{\text{E}}\text{X}$ and METAFONT programs in two volumes of his *Computing & Typesetting* series, *T_EX: The Program* and *METAFONT: The Program*, Knuth printed copies of the programs' source code annotated in the manner of Lions's book. The result was a significant improvement over Lions's own major innovation: where Lions annotated the code of other authors, Knuth supplies his own annotations, a time-savvy move of consideration that invited others to contribute to the program's authorship (in explicating his own text, Knuth reveals the intentions underlying his code and so enables other programmers to fix glitches and improve the code's syntax). Treating authorship as a type of accountability, Knuth stresses that better, clearer programs result when the writer has to criticize his own code. “[I]t seems to me that at last I'm able to write programs as they should be written,” he writes. “My programs are not only better explained than ever before; they are also better programs, because the new methodology

encourages me to do a good job” (*Literate Programming* 100).

T_EX and METAFONT were not just *examples* of literate programming, however; they were designed to *aid* literate programming. In writing them, Knuth developed what he called the WEB system of programming. T_EX and METAFONT also completed the original WEB project, forming the digital typesetting component that allowed a programmer to transform “literate” code into a beautifully formatted and printed text. In 1986, Knuth explained the symbiotic relationship between T_EX, METAFONT, and literate programming thus:

As I wrote the programs for T_EX and METAFONT, I wanted to produce programs that would represent the state of the art in computer programming, and this goal led to the so-called WEB system of structured documentation. I think that WEB might turn out to be the most important thing about all this research—more important in the long run than T_EX and METAFONT themselves—because WEB represents a new way to write software that I think is really better than any other way. The use of WEB has made it possible to write programs that are so readable, I think there are already more people who understand the inner workings of T_EX than now understand any other system of comparable size. Furthermore, I think it’s fair to claim that WEB has made T_EX and METAFONT as portable, as maintainable, and as reliable as any other pieces of software in existence. (*Digital Typography* 555)

Ingeniously integrating the processes of compilation and publication, Knuth’s WEB system provides a way for programmers simultaneously to build working programs for computers and produce typeset editions of their work for human readers.¹³ Knuth describes the workings of WEB as follows:

A WEB user writes a program that serves as the source language for two different system routines. One line of processing is called *weaving* the web; it produces a document that describes the program clearly and that facilitates program maintenance. The other line of processing is called *tangling* the web; it produces a machine-executable program. The program and its documentation are both generated from the same source, so they are consistent with each other. . . . Suppose you have written a WEB program and put it into a computer text file called COB.WEB (say). To generate hardcopy documentation for your program, you can run the WEAVE processor; this

¹³The combined potential of T_EX and WEB allow for sophisticated textual manipulation of source code. As Wayne Sewell explains, “T_EX automatically handles details such as microjustification, kerning, hyphenation, ligatures, and other sophisticated operations, even when the description part of the source is simple ASCII text. WEB adds functions which are specific to computer programs, such as boldface reserved words, italicized identifiers, substitution of true mathematical symbols, and more standard pretty-printer functions such as reformatting and indentation” (42).

is a system program that takes the file COB .WEB as input and produces another file COB .TEX as output. Then you run the T_EX processor, which takes COB .TEX as input and produces COB .DVI as output. The latter file, COB .DVI, is a ‘device independent’ binary description of how to typeset the documentation, so you can get printed output by applying one more system routine to this file. (“Literate Programming” 101–102)

Although Knuth’s original WEB program used Pascal, Knuth has since extended the concept to CWEB, which combines T_EX with the C programming language. Again, Knuth describes how the digital typesetting language can work in harmony with a powerful coding language:

The typographic tools provided by T_EX give us an opportunity to explain the local structure of each part by making that structure visible, and the programming tools provided by languages like C make it possible for us to specify the algorithms formally and unambiguously. By combining the two, we can develop a style of programming that maximizes our ability to perceive the structure of a complex piece of software, and at the same time the documented programs can be mechanically translated into a working software system that matches the documentation. (*The CWEB System of Structured Documentation* 1)

As the WEB and CWEB systems of literate programming illustrate, Knuth’s commitment to the ideal of structured, “readable” code was total. A devoted student of the history of bookmaking and print technology, Knuth is as concerned with the typographical appearance of printed code as he is with how the code itself is written. In Knuth’s vision, truly literate programming must be as attentive to the print quality of a program as it is to the content of the program itself; only then will the true beauty of code become available to the reader’s discerning eye, and only then will the programmer take seriously his aesthetic obligations. The logic is that of the book: a great work deserves to be printed in a manner that is worthy of it. Knuth draws no practical or aesthetic distinction between literature and code; each deserves to be printed in the purest, most elegant type available. T_EX and METAFONT were Knuth’s responses to the failure of the publishing industry to honor this simple, indisputable truth.

In the 1960s, photo-optical technology started to replace hot-lead typesetting machines like Monotype. Knuth considered photo-optical technology to be vastly inferior to hot-lead typesetting, especially for complex mathematical texts such as his *Art of Computer Programming*. Appalled by the poor print quality of modern mathematical typesetting, and the general deterioration of the bookmaking art, Knuth set out to remedy the problem by writing his own digital typesetting system

from scratch. The T_EX typesetting program and the METAFONT font-design program were Knuth's answers to what he perceived as the book industry's failure to meet its obligation to beautiful books. Though T_EX and METAFONT are pieces of software, they were designed as a condensation, crystallization, and extension of the entire history of print and printmaking, and are built on a massive amount of research into the print tradition. Furthermore, Knuth wrote his programs with explicitly archival goals in mind: he wanted to "create systems that would be independent of changes in printing technology" (*Digital Typography* 559).¹⁴ Designed to meet the special printing needs of mathematics and built to last, T_EX set a standard for publishing code as text: programs printed in T_EX looked like the sleek literary productions they are. At once an exemplary instance of literate programming and a program designed to enhance literate programming, T_EX thus realized Knuth's methodological vision twice over.

Critical responses to Knuth's concept of literate programming speak eloquently to the power of Knuth's bookish vision. John Bentley, a programming critic known for his column "Programming Pearls," wrote about T_EX as if it were a profoundly absorbing work of fiction:

When was the last time you spent a pleasant evening in a comfortable chair, reading a good program? ... I'm talking about cuddling up with a classic, and starting to read on page one. ... Until recently, my answer to that question was 'Never.' I'm ashamed of that. I wouldn't have much respect for an aeronautical engineer who had never admired a superb airplane, nor for a structural engineer who had never studied a beautiful bridge. Yet I, like most programmers, was in roughly that position with respect to programs. That's tragic, because good writing requires good reading—you can't write a novel if you've never read one. (*Literate Programming* 137)

Hailing Knuth as the man who is almost singlehandedly making programmers aware that there is a "canon" of code, a tradition of quality programming whose great works should be read and appreciated by all serious members of the discipline, Bentley writes of the pleasure he got from

¹⁴Knuth believes that "these goals of top quality and machine independence seem to be achieved" (*Digital Typography* 559). An important archival aspect of T_EX is that its input files may be saved as plain ASCII text, which should remain readable on any computer system for hundreds of years to come. By contrast, the proprietary data formats used by many commercial word processing and typesetting systems remain useful only as long as the specifications for those programs remain constant. Because they change frequently, often without respect for "backward compatibility," proprietary formats cannot store data effectively for more than a few years at a time.

studying Knuth's code as if it were a work of art, the massive accomplishment of a master whose every line leaves something to learn and much to admire:

I recently spent a couple of pleasant evenings reading the five-hundred-page implementation of the T_EX document compiler. I have no intention of modifying the code, nor am I much more interested in document compilers than the average programmer-on-the-street. I read the code, rather, for the same reason that a student of architecture would spend an afternoon admiring one of Frank Lloyd Wright's buildings. There was a lot to admire in Knuth's work: the decomposition of the large task into subroutines, elegant algorithms and data structures, and a coding style that gives a robust, portable, and maintainable system. I'm a better programmer for having read the program, and I had a lot of fun doing it. (*Literate Programming*, 137–38)

The analogy with Frank Lloyd Wright is apt: in 1901, Wright delivered his famous lecture "The Art and Craft of the Machine," which welcomed the machine into American architecture and countered the prevailing idea that machines contributed to a decline in architectural craftsmanship. As did Wright, so does Knuth pay attention not only to the structured elegance of his programs but to the craftsmanship inherent in their presentation.

When Bentley wrote to Knuth asking if he could have a sample of Knuth's code for his "Programming Pearls" column, Knuth challenged Bentley to challenge him: "Why should you let me choose the program? My claim is that programming is an artistic endeavor and that the WEB system gives me the best way to write beautiful programs. Therefore I should be able to meet a stiffer test: I should be able to write a superliterate program that will be noticeably better than an ordinary one, whatever the topic." Bentley obliged, assigning Knuth to write a program to determine the most frequently-used words in a technical paper. The result more than proved Knuth's methodological claim. M. D. McIlroy, reviewing Knuth's program for Bentley's column, called it "a sort of industrial-strength Fabergé egg—intricate, wonderfully worked, refined beyond all ordinary desires, a museum piece from the start" (*Literate Programming* 174). The anecdote is instructive: at once aesthetic artifact and operative code, Knuth's literate programs have become synonymous with the specific literary bent of contemporary computer programming. For many, they exemplify not only how beautiful code can be, but also how effective aesthetics are as a means of urging code toward ever more capable, comprehensive forms.

2.5 Perl Poetry

This chapter revolves around a series of closely related questions: why is it important to see code as text? What do programmers gain by doing so? And what do literary and cultural critics lose when they neglect to consider code as text? I opened my investigation of these questions by reading an English language poet writing about code; I will end by reading programmers who write code that is also poetry. So far, we have seen how useful it has been to programmers to metaphorize their work as literature: whether licensing it under the U.S. Copyright Act, teaching it in the classroom, or promoting a literate approach to programming, programmers have used the code-as-text equation to protect their work (by defining themselves as authors) and to perfect their craft (by treating both the writing and the reading of code as an aesthetic literary act). Perl poetry brings the conceptual fusion of code and text to life. Built on the analogy between programs and poems, Perl poetry is a new genre whose most important achievement is to press that analogy into an actual identity.

Over the past decade literary scholars have become fascinated with emergent forms of electronic textuality. Spurred on by works such as Jay David Bolter's *Writing Space: The Computer, Hypertext, and the History of Writing* (1991), George Landow's *Hypertext: The Convergence of Contemporary Critical Theory and Technology* (1992), and Richard Lanham's *The Electronic Word: Democracy, Technology, and the Arts*, (1993), critical theorists began to assume what the subtitle of Landow's book promised: that technology and critical theory had undergone a "convergence," and that the advent of hypertext represented the fulfillment of poststructuralist prophecies about the nature of language and authorship, and about the future of interpretation.

It's not hard to see why literary scholars were so taken by the idea of electronic textuality during the 1990s. As Landow and others argued, hypertext (by which they meant both the experimental "hypertext fiction" of authors such as Michael Joyce and the hyperlinked, radically interconnected structure of the World Wide Web itself) seemed to formally realize post-structuralist ideals that were already powerfully entrenched within academic literary study and avant-garde poetics. Specifically, it literalized the vision of continuous, unbounded, authorless textuality so

polemically espoused by post-structuralist theorists such as Roland Barthes and Jacques Derrida. As Barthes put it in a representative passage from *S/Z*, “in this ideal [readerly] text, the networks are many and interact, without any one of them being able to surpass the rest; this text is a galaxy of signifiers, not a structure of signifieds; it has no beginning; it is reversible; we gain access to it by several entrances, none of which can be authoritatively declared to be the main one; the codes it mobilizes extend *as far as the eye can reach...* (5–6). The World Wide Web realized the infinite, infinitely expansive “network” that was, at the moment of Barthes’s writing in 1970, largely prospective and metaphorical. As an actual “galaxy of signifiers,” the Web seemed to its early postmodern analysts to exemplify the radically de-centered, wholly democratic form of signification that had long been envisioned and attempted by post-structuralist writers.¹⁵

The combination of literary experimentation and computing technology has a longer history than many contemporary cybercritics tend to acknowledge, however. The “convergence” of

¹⁵The advent of hypertext has in turn given rise to an extraordinarily polarized debate within literary studies about whether electronic textualities will replace the material book as the primary means of gathering, organizing, and disseminating cultural value. Pitting technological visionary (“The computer is going to replace the book as the principal means of assembling, storing, and distributing information, ideas, stories, and knowledge”) against traditional bibliophile (“The book must be saved, and the computer must be banished from the realm of publishing in order to ensure that the book form survives”), the debate is one where feelings run high and reason runs comparatively low. Neither pole is particularly defensible, and both ignore the existence of historically nuanced forms of electronic textualities, such as Donald Knuth’s T_EX and METAFONT programs. With their productive melding of state-of-the-art technology and reverent regard for the history of bookmaking and typesetting, T_EX and METAFONT have contributed immeasurably to the print tradition. Moreover, because Knuth gives T_EX and METAFONT away for free, they give anyone the power to create typographically beautiful documents at almost no cost. Knuth notes how, using T_EX and METAFONT, “a dedicated author now has the power to prepare books that previously were prohibitively expensive”; he also reports receiving scholarly publications requiring special typography (like Eskimo language folk tales or critical editions of Sanskrit texts) that “probably would never have existed” if not for the availability of his typesetting system (*Digital Typography* 17). For more on the book vs. electronic textuality debate within the humanities, see the essays collected in Geoffrey Nunberg’s 1996 collection *The Future of the Book*. For a representative collection of essays fetishizing the book as a “mythic and material object,” see Jerome Rothenberg and Steven Clay’s 2000 collection *A Book of the Book: Some Works and Projections About the Book and Writing*. Technophobic attacks on the very idea of electronic textuality may be found in Barry Sanders’s *A Is for Ox: Violence, Electronic Media, and the Silencing of the Written Word* and Sven Birkerts’s *The Gutenberg Elegies: The Fate of Reading in an Electronic Age*.

avant-garde art with the field of computing was not, as Landow indicates, wholly a phenomenon of the 1990s; indeed, experimental writers were drawing on computing language as far back as the 1960s, when French poets began exploring the potential of computing languages to open up their ongoing investigations into the limits of meaning, the nature of signification, and the essence of poetic form. In the early 1960s, François Le Lionnais, a founding member of the French experimental literary group Oulipo (Ouvroir de Littérature Potentielle), composed a poem using the twenty-four-word lexicon of the Algol 60 programming language (Mathews and Brotchie 47). Inspired by Le Lionnais, Noël Arnaud built on his fellow poet's conception. Using the restricted vocabulary of Algol 60 to literary ends, he produced an entire volume of code-poetry in 1968 entitled *Poèmes algol*. With a foreword by Le Lionnais, the volume exemplified Arnaud's vision of a supremely compact, minimalist verse. Here is one of Arnaud's Algol poems (Mathews and Brotchie 47):

Table

Begin: to make format,
go down to comment
while channel not false
(if not true). End.

As "Table" makes clear, Arnaud and his fellow Oulipian poets were inspired less by the Algol language itself (despite Alan Perlis's remark that the language was "an object of stunning beauty") so much as by its potential for enforcing formal rigor and economy of expression. Arnaud employed similar constraints in other works—his poem "Adverbities of Eros," for instance, is composed entirely of adverbs and adverbial phrases—and other Oulipians sought similar formal constraint, experimenting with limiting the numbers of words they could use to complete a poem, or confining their poetic vocabularies to specific, narrow pools of words. For example, Harry Matthews's story *Their Words, For You* limits its vocabulary to 185 words; likewise, Jacques Jouet's poems in *The Great-Ape Love-Song* are composed entirely from the ape language of the *Tarzan* books (Mathews and Brotchie 241). While Le Lionnais and Arnaud wrote poems in Algol, then, they were principally concerned with Algol as a formal device for setting parameters on their

art. Ultimately, they were less interested in the idea of programming as a literary artform than in Algol as a ready-made restricted vocabulary with which to experiment.

Algol was something Oulipo poets used; while the language was materially interesting to them as a signifying system, it did not appeal to them either as a language for commanding computers or a means of expressing those commands in a formally rigorous, syntactically elegant way. By contrast, the Perl programming language has inspired programmers themselves to explore the formal and expressive properties of computing language. Invented in 1986 when programmer and linguist Larry Wall set out to solve some limitations with the Unix text processing tool *awk*, Perl (Practical Extraction and Report Language) quickly grew from a programmer's tool into a fully-fledged programming language. When Wall released Perl to the hacker community on December 18, 1987, he summarized its underlying design philosophy in two maxims: "There's more than one way to do it," and "Easy things should be easy, and hard things should be possible" (Brate 259–60). The hacker community quickly found Wall's creation to be a language of unprecedented versatility and flexibility; consequently Perl has become one of the most widely-used "glue languages" of the Internet. When Wall released Perl 3.0 in 1989, he licensed it under the terms of Richard Stallman's GNU Public License, thus ensuring that Perl would always remain freely available to the programming community.

Larry Wall wrote the first Perl poem on a workday morning in March 1990. Sharon Hopkins, having noticed the JAPH signature programs being written by Randal Schwartz,¹⁶ suggested that Wall try to write a poem in Perl. By noon, Wall had written this simple Perl "haiku":

```
Print STDOUT q
Just another Perl hacker
Unless $spring
```

Wall's Perl poem is a classic haiku—three lines long, with a 5–7–5 syllable scheme, specifying the

¹⁶JAPHs are short e-mail signature files written in Perl. When run through the Perl interpreter, they print out the phrase "Just Another Perl Hacker." Programmers often vie with one another to create especially ingenious or obfuscated JAPHs.

season of its composition.¹⁷ Wall's Perl poem is also a program. When you run it, it instructs the local printer to print "Just another Perl hacker." The rules of Perl poetry were established by Wall's doubly meaningful, doubly formal verse. All Perl poetry obeys the rules of poetic syntax, while at the same time obeying the compositional rules of the Perl programming language. To qualify for inclusion in the genre, a Perl program must be legible on two levels: to the trained aesthetic eye that scans it, and to the Perl compiler that interprets it.

Wall touched off a trend with his haiku. There are now Web sites devoted to the genre, there have been a number of Perl poetry contests, and some of the better known Perl poets have had their work published in print. Fittingly, Sharon Hopkins, the woman who originally thought of Perl poetry, has become the reigning laureate of the genre. Her Perl poem "listen" has been published in both *The Economist* and *The Guardian*:¹⁸

```
#!/usr/bin/perl
```

```
APPEAL:
```

```
listen (please, please);
```

```
open yourself, wide;  
    join (you, me),  
connect (us,together),
```

```
tell me.
```

```
do something if distressed;
```

```
    @dawn, dance;  
    @evening, sing;  
    read (books,$poems,stories) until peaceful;  
    study if able;
```

```
write me if-you-please;
```

```
sort your feelings, reset goals, seek (friends, family, anyone);
```

```
do*not*die (like this)  
if sin abounds; keys (hidden), open (locks, doors), tell secrets;
```

¹⁷In order for the metrical scheme to work, "STDOUT" must be pronounced "standard out," and "\$" should be pronounced "dollar."

¹⁸I am grateful to Sharon Hopkins for permission to reproduce "listen" in full here.

do not, I-beg-you, close them, yet.

accept (yourself, changes).
bind (grief, despair);

require truth, goodness if-you-will, each moment;

select (always), length(of-days)

Sharon Hopkins, Feb. 21, 1991

listen (a perl poem)

rev. June 19, 1995

Within the Perl programming language, parentheses work to create lists of elements which are then acted upon by functions. “Listen(),” “join(),” “connect(),” “read(),” “seek(),” “die(),” “keys(),” “open(),” “sort(),” “reset(),” “bind(),” “accept(),” and “length()” are all Perl function calls, so that the line “seek (friends, family, anyone)” demonstrates the “seek()” function acting on the elements “friends,” “family,” and “anyone.” Within poetic language, the parentheses create whispered asides, moments that may or may not be part of the official “APPEAL” of the poem’s opening; they may be inward wishes or imaginings. When Hopkins merges the formalities of Perl with traditional poetic convention, the effect is to play with the overlap between the rules of linguistic and programming grammar. Using Perl’s imperative function calls to express heartfelt emotional appeals, and modulating the expressive power of language with the formal syntax of programming, Hopkins creates a poetic economy in which Perl controls and shapes the poet’s emotions. When Hopkins writes “do*not*die (like this),” the instruction goes both to her lover and to the Perl interpreter, telling neither to “exit” the poem’s “appeal.”

In addition to her poetic writings, Sharon Hopkins has written a much-circulated essay on Perl poetry. Tracing the history of computer-generated writing to pangram generators and the Oulipo poets’ experiments, she notes that Perl poetry is the first effort to “develop human-readable creative writings in an existing computer language . . . that not only [have] meaning in [themselves] but can also be successfully executed by a computer.” Noting that “the great advantage formal poetry has over free verse is the balance provided between familiarity and strangeness, stasis and innovation,” she remarks that part of the “surprise” of Perl poetry, at least for the programmer,

is seeing language elements employed for poetic ends (1). With its 250-word vocabulary, Perl provides the enterprising poet with plenty of scope for innovation. Subsequent Perl efforts have involved adapting poetic forms such as sonnets and odes, reworking the poems of canonical poets such as William Butler Yeats, and, more generally, writing free verse Perl code that addresses the same sorts of themes mainstream poetry would: love, work, longing, and life.

Perl poems are what their creator Larry Wall meant for Perl itself to be when he wrote it: pearls of perfectly-formed syntax that arise from an ideal compression of language into code, poem into program, art into commanding expression. Perl poetry literalizes the aesthetic urge of Perl programming: it symbolizes, within Perl programming culture, the potential beauty of all Perl code, both its elegance and flexibility of expression and the clarity of its interpretation and execution by the interpreter. Perl poetry translates the aesthetics of code in such a way as to stress that all well-written Perl programs are already poetry. For this reason, it should not surprise us that so many Perl poems translate well-known canonical poems into working code. For example, Yeats's poem "The Coming of Wisdom with Time" is the subject of a Perl poem by Wayne Meyers. Here is the poem as Yeats wrote it:

Though leaves are many, the root is one;
Through all the lying days of my youth
I swayed my leaves and flowers in the sun;
Now I may wither into the truth

And here is the poem Perled, translated into operative code:

```
while ($leaves > 1) {  
  $root = 1;  
}  
foreach($lyingdays{'myyouth'}) {  
  sway($leaves, $flowers);  
}  
while ($i > $truth) {  
  $i--;  
}  
sub sway {  
  my ($leaves, $flowers) = @_  
  die unless $^O =~ /sun/i;  
}
```

When run through a Perl compiler, Meyers's "programmatic" interpretation of Yeats performs the movement of "The Coming of Wisdom with Time" for the Perl interpreter. Picking up on Yeats's use of the verb "sway," Meyers defines a "sway()" subroutine to ensure that the poem will "die()" if the operating system name variable shows the poem "in the sun"—i.e., running on a Sun Microsystems operating system. Passing Yeats through the Perl interpreter yields a formally new kind of poem, a work of art that literally works. As such, the work of Perl poetry may be said to be that of marking up a new canonical language, one whose deeply functional aesthetics emerge from and merge with older established modes of composition.

If the Perl poet's willingness to break up other poems in order to rewrite them seems on some level to be disrespectful, frivolous, or even blasphemous, we should bear in mind that all poetry relies upon the manipulation of syntax to separate itself from other written forms. Yeats himself made this point when he converted Walter Pater's sentence-long description of the Mona Lisa into a piece of free verse, thus:

She is older than the rocks among which she sits;
Like the Vampire,
She has been dead many times,
And learned the secrets of the grave;
And has been a diver in deep seas,
And keeps their fallen day about her;
And trafficked for strange webs with Eastern merchants;
And, as Leda,
Was the mother of Helen of Troy,
And, as St. Anne,
Was the mother of Mary;
And all this has been to her but as the sound of liars and flutes,
And lives
Only in the delicacy
With which it has molded the changing lineaments,
And tinged the eyelids and the hands.

Yeats's point is that Pater's prose *is* poetry—a point he makes, paradoxically, by reformatting it *as* poetry. (The move is paradoxical because if the prose really "is" poetry, it would not need to be reframed "as" poetry.) Perl poetry can be read as an extension of this logic: by marking up "real" poetry (some of it Yeats's), Perl poets assert the poetic nature of code by making the Perl programming language the framing, formal syntax of individual poems. In so doing, Perl poets

both make reformatting into the form and content of poetry, and reformat poetic language itself by making programming language into the language of a new kind of poetry, into that which “has molded the changing lineaments” of modern verse, “ting[ing] the eyelids and the hands.” Yeats may look at first to be the victim of the Perl poet’s invasive reworking. In reality, he is the architect of the Perl poet’s codified gesture.

As the Yeats example indicates, Perl poets are as deeply embedded in literary history as they are in Perl itself. Indeed, the Perl Poetry project archived on the Web site CPAST (Comprehensive Perl Arcana Social Tapestry) opens with this quote from T. S. Eliot’s “The Dry Salvages”:

Here the impossible union
Of spheres of existence is actual,
Here the past and future
Are conquered, and reconciled,
Where action were otherwise movement
Of that which is only moved
And has in it no source of movement—
Driven by daemonic, chthonic
Powers. And right action is freedom
From past and future also.
For most of us, this is the aim
Never here to be realised;
Who are only undefeated
Because we have gone on trying;
We, content at the last
If our temporal reversion nourish
(Not too far from the yew-tree)
The life of significant soil.

The lines announce the Perl poem as the making “actual” of that “impossible union of spheres of existence,” the marriage of poetry and programming. And in so doing they position Eliot as the man who best defines the essence of Perl poetry, who captures most compactly the liberatory energies underlying the genre’s attempt to remake not simply the disparate genres that compose it, but in so doing to remake the idea of genre itself. If Yeats may be seen as the architect of Perl’s creative markup, Eliot may be read as its enabling muse.

Together, Meyers’s transformation of Yeats’s verse into a programming “Perl” and CPAST’s inspirational Eliot epigraph point to a complex and textured relationship between literary

history—specifically the modernist poetic canon—and the history of computer programming. That a programming culture centered on close reading source code should begin producing poetry that is itself a dense amalgam of literary and computing languages raises a number of questions about the notion of genre, the location of literature, and the parameters of art. How, most basically, are we to understand the hybrid artifact that is the Perl poem? What kind of reader, and what sort of reading practice, are required to think about Perl poetry? What happens to the concept of the programming language when it doubles as poetic language? What happens to the concept of the poem when its content consists of code? No literary critic has written about Perl poetry; the genre's critics have thus far been the Perl programmers who are its practitioners. Nonetheless, in establishing code as both a practical and an aesthetic object, Perl poetry demands that the serious poetry critic come to terms with computing software in the same way students of operating systems have begun to come to terms with reading and writing practices more commonly associated with literary production. How might Perl poetry shape or reshape our understanding of contemporary experimental poetry? We won't know until literary critics begin to grapple thoroughly with the genre's uniquely functional, thoroughly interdisciplinary aesthetics. However, poetry criticism will not be able to grasp the Perl poem's aesthetic until poetry critics begin to take code seriously as a literary language. A fully cognizant poetry criticism—one capable of comprehending Perl poetry and placing it within the context of postmodern poetic experiment—will necessarily also be a technologically proficient criticism of the structure, form, and syntax of programming language.

There are many kinds of experimental poetry on the Internet, and there have been a number of attempts to use the immediacy, the mobility, and the interactive potential of the electronic environment to stretch poetry into new shapes. Most of them tend, however, to preserve a simplistic and infinitely problematic distinction between the very categories the concept of the electronic poem would seem to want to erase: poem and code, visible art and invisible mechanism, aesthetic surface and functional hardware. Consider, for example, Kenneth Goldsmith's poem, "Soliloquy." Structured around the idea that an electronic poem acquires its beauty from the continuous interplay of the oppositions named above, "Soliloquy" is an interactive poem that gets its syntax from the reader's manipulation of the cursor within a Web browser: beginning as a flat,

empty surface of white screen, the poem appears in fragments and bits as the reader moves the cursor. The effect is one of poetic creation by peeling: each movement of the cursor “peels” back a bit of the blank screen to reveal words or phrases. The poem is as long or short as the reader makes it, but no matter what the length, or pattern of peeling, the point of the poem is always that poems involve revelations, and that in an electronic environment, the revelation of meaning is something that can be programmed to occur.

Even so, the poem limits the extent of the revelations it makes possible. Mark Poster describes the genre of “cybertext” to which Goldsmith’s poem belongs: “Cybertexts . . . highlight the dynamic production of text, turning this production into a spectacle. Experiencing the text means watching words and meaning emerge and evolve on the screen, animated by the invisible code of a computer program” (9). Poster conceptualizes the “cybertext” as separate from, although animated by, “invisible code” that underpins it; he conceives of this layered text in turn as a kind of poem. As such, his metaphoric treatment of cybertexts as poems animated by invisible code perfectly recapitulates the logic of Goldsmith’s poem, whose soliloquy may be said, finally, to announce the solitude of the poem itself.

Perl poetry refuses such inevitably confused distinctions in favor of a thorough integration of the things other Web poetry tries to keep separate. In this, Perl poetry has more in common with the idea of “codework” developed by Alan Sondheim, Florian Cramer, *et. al.* in a recent series of essays published by *American Book Review*.¹⁹ Drawing critical energy from “cybercultural” and “cyberpunk” discourse, Sondheim defines “codework” as “a literary avant-garde concerned with the intermingling of human and machine” (1). McKenzie Wark comes closest to the spirit of Perl poetry when he comments that “codework . . . is a form of electronic literary work in which the protocols and structural aspects of the supporting technology, from which/to which the work is applied, are explored and exposed within the body of the text.” However, many of the “codework” experiments described in these essays rely not on programming languages, but on hypertext manipulation or sometimes the generation of pseudo-code—for instance, Beatrice Beaubien cites

¹⁹Contributors to the series include Alan Sondheim, McKenzie Wark, Talan Memmott, Beatrice Beaubien, Belinda Barnet, and Florian Cramer.

the “codework” poet Mez “defining herself” on a mailing list as follows: “if fle.sh!=sub.stance ov n animal body// sur.face of the body, s-pecially>with respect 2 flesh col.awe; [pinkish white with a tinge of yellow; > pinkish cream]” (3). Neither hypertextual experiment nor clever typographic rearrangement should be confused with the syntax of actual programming languages.

In Perl poetry, the code does not underpin the poetry: the poetry is the code, and the code is the poetry. The visible art is the visible mechanism, and the aesthetic surface is completely coextensive with both the human reader and the Perl interpreter that reads it. Perl poetry thus amounts to an aesthetic and programmatic disruption of the idea that art and programming should exist as separate entities, and that computing hardware should merely provide the material underpinning for poetic possibility. Far from poetry “animated by the invisible code of the computer program,” Perl poetry is animated by the visible code of computer programming languages.

If we stop seeing literary aesthetics and programming aesthetics as two separate entities that have been held apart by institutional and technological barriers, and begin to see them as interpenetrating and complementary modes of thought—something Perl poetry demands that we do—then we have to see their intertwining as having a history; we have to ask how literary and programming aesthetics have come together historically, and we have to identify the historical pressures and opportunities that have worked together over time to create such simultaneously analogous and separate realms. In the next chapter I will begin to do some of that necessary work by turning to one of the last century’s greatest literary aesthetes and most programmatically aesthetic author, James Joyce. As we will see, the authorial practices at work in Joyce’s composition of *Finnegans Wake* are also the authorial practices that animate programming aesthetics.

Chapter 3

“Harmonic Condenser Enginium”: Object-Oriented Joyce

Chapter One of this study argued that increasing degrees of abstraction from computing hardware have defined computer programming practices for the last half-century, showing how the effects of those abstractions have shaped programmers' professional and cultural identity, programming politics, and programming aesthetics. Concerned with the specifically aesthetic implications of code's abstraction into language, Chapter Two examined how some computer programmers have grounded their coding practices so deeply in literary aesthetics that they can talk about code as "beautiful," "elegant," "expressive," and "poetic"; write and read computer programs as literary texts; and even, in the case of Perl poetry, produce hybrid artifacts that are at once poems and programs. My concern thus far has been to show how identifiably literary sensibilities have influenced programming theory and culture. This chapter develops and deepens that aesthetic concern by reversing its focus: I argue in this chapter that programming theory, as a body of knowledge that thinks deeply about the semantics and organization of textual structures, can contribute to the project of literary study. Furthermore, in outlining a comparative genealogy of Joycean literary experiment and programming theory I argue that modern programming practice and Joycean semantics not only share common philosophies but common origins.

My first goal in this chapter will be to illuminate formal commonalities between Joyce's late aesthetic and authorial ideals, as exemplified in *Finnegans Wake*, and the theoretical principles that underpin the modern "object-oriented" paradigm for software development. In analyzing the complex structural patternings at play in Joyce's late work, the chapter situates itself alongside the criticism of Joyce scholars such as Clive Hart, A. Walton Litz, David Hayman, John Bishop, Vicki Mahaffey, and Jean-Michel Rabaté; in showing how Joyce's later work shares with object-oriented programming theory a set of similarly derived ideas about managing and structuring complex systems, it contributes to the growing body of work on Joyce and complexity theory.¹ My

¹I use "complexity theory" here to signify work that studies the emergence of order from dynamic systems of interacting elements. A classic essay on the design of complex systems is Simon's 1962 essay *The Architecture of Complexity* (republished in his 1982 book *The Sciences of the Artificial*). Courtois's 1985 essay "On Time and Space Decomposition of Complex Structures" applies Simon's principles to software architecture, while Klaus Mainzer's 1994 *Thinking in Complexity* traces the role of complexity science through numerous disciplines, including computer science, economics, and neuroscience. Mark C. Taylor's 2001 study *The Moment of Complexity* outlines the philosophical and aesthetic ramifications of complexity theory. In his 1997 book *Joyce*,

second goal will be to historicize these commonalities by tracing a genealogy of object-oriented design. Contending that both Joyce and pioneering object-oriented programmers grounded their approaches to complexity in modernist theories about human cognition, particularly in theories about how children think, learn, and play, I propose that their respective approaches to structure grow directly out of early twentieth-century ideas about how the mind organizes an increasingly complex and multivalent world.

The chapter is divided into four main sections. The first section discusses the authorial metaphors that Joyce applied to his late work and traces the critical heritage of those metaphors through to present-day Joyce scholarship. Relying heavily on engineering imagery to describe *Finnegans Wake's* ongoing construction, Joyce gradually stopped thinking of himself as the book's "author" and began conceiving of himself as a "master builder," or, as he wrote in a letter to Harriet Shaw Weaver, as "the greatest engineer" the world had yet seen. Joyce's own engineering imagery subsequently wound its way into the critical literature to describe the evolving formal complexities of Joyce's textual strategies, particularly during the latter part of his writing career. As critics have repeated and recycled Joyce's engineering imagery, so, too, have they modernized it; as I will show, writers such as Daniel Dennett, Jacques Derrida, and Donald Theall have reincarnated Joyce's mechanistic metaphors for the computing era by explicitly describing him as a software engineer and calling his texts "Joycean software" (Dennett) or even "joyceware" (Derrida). In this section I suggest that we should consider these characterizations as more than mere throwaway analogies. Seeing them instead as logically connected to Joyce's own authorial conceptions, I suggest that a serious reconsideration of *Finnegans Wake's* intricate structural patterning can benefit greatly from the insights of computer programming theory.

The chapter's second section introduces the formal conceptual principles that underlie the object-oriented programming paradigm. Situating the evolution of object-oriented design principles within the context of the "software crisis," an epoch in computer programming history that began in the 1960s and continues today, I describe how and why some programmers embraced

Chaos, and Complexity, Thomas Jackson Rice uses the principles of complexity science to read *Ulysses* and *Finnegans Wake*.

this development paradigm as a “silver bullet” solution to the problem of managing computer programs’ exponentially increasing complexity. Confronted with ever-larger software projects and frustrated with the organizational limitations of a “procedural programming” style largely governed by linear conceptions of textuality, programmers and language designers from the 1970s onward experimented with new, non-linear approaches to coding. Ultimately they gravitated toward an “object-oriented” methodology ordered around the principles of abstraction, inheritance, polymorphism, and encapsulation (I explain these principles in detail below). Object-oriented design enabled programmers to manage increasingly complex software systems by making them modular, by breaking code down into mobile, self-enclosed units known as “objects.” Furthermore, this new methodology turned the work of computer programming into a distinctly ontological and mimetic enterprise: from its earliest uses in interactive graphics and simulation software, the object-oriented paradigm gave programmers the potential to define *computational* objects that simulated the behavior of *real-world* counterparts. Charged with producing a modular mimesis of an increasingly complex and polyvalent reality, the object-oriented programmer’s work begins to bear a remarkable similarity to that of the modernist artist.

The third section of the chapter suggests how the principles of object-oriented programming can illuminate the formal endeavor of Joyce’s later writings. During Joyce’s lifetime, emergent ideas in the physical and human sciences—particularly in physics, mathematics, and psychology—exposed the extraordinary complexities underpinning physical reality, the nature of human consciousness, and the relationship between the two. Although the critical literature frequently characterizes Joyce’s later texts as self-enclosed and self-referential, I understand Joyce’s experiments with revolutionary new ways of organizing, compressing, and embedding multiple layers and kinds of meaning within his writing to indicate not that Joyce abandoned the realist project, but that he committed his texts to representing both physical reality and the human mind in all their newly-discovered physical and mental complexities. The argument of this section, then, is that Joyce’s and object-oriented programming’s revolutions in textual structure both originate in the need to model systems of ever-increasing complexity, and that the semantic paradigms that result from these revolutions share striking formal commonalities. In

turn, I show how an understanding of object-oriented programming theory can illuminate the formal architectural complexity of Joyce's late texts. Paying particular attention to two of Joyce's late stylistic innovations—his adoption of the “portmanteau word” and his use of what Roland McHugh calls “personality condensation” in *Finnegans Wake*—I suggest that object-oriented theory is an invaluable aid to understanding Joyce's technique of making meaning modularly and dynamically while retaining a high degree of formal coherence.

While the third section of the chapter proceeds in a strategically ahistorical, strictly conceptual manner, the fourth and final section historicizes and contextualizes my application of object-oriented methodology to Joyce's later writing by drawing out the genealogical debt both Joyce and object-oriented programmers owe to modernist theories of cognition and developmental psychology. Joyce, whose interest in developmental psychology was fuelled by increasing concern over his daughter's schizophrenia, filled his literary writings with meditations on children's thought processes and with philosophical speculations on how children acquire knowledge. A generation later, computing science researcher and experimental educator Alan Kay strove to overcome hierarchical, authoritarian models of knowledge transmission in ways that are deeply reminiscent of the Joycean project. Kay's response was to design Smalltalk, the first fully-fledged object-oriented programming language, as a tool for expressive, creative learning. As an interactive, responsive pedagogical tool, Smalltalk not only thinks about children's minds in ways that are deeply reminiscent of the educational paradigms elucidated in Joyce's work, but also extends a cognitive tradition in which Joyce himself was deeply invested. In bringing child psychology to bear upon programming language design, Kay was deeply influenced by the pioneering work of artificial intelligence theorist and cognitive psychologist Seymour Papert, with whom he studied in the 1960s. In turn, Papert, a former student of Swiss child psychologist Jean Piaget, derived much of his programming philosophy from modernist concepts of human cognition. Showing how the object-oriented paradigm that rose to prominence in software development circles during the 1980s actually has its roots in modernist debates about complexity, psychology, pedagogy, and knowledge—debates in which Joyce himself was a committed participant—this section will show how Joyce's late texts and object-oriented design philosophy share common

conceptions about how the mind develops and orders thought.

Though the chapter initially proceeds analogically, showing the parallels between Joyce's texts and programming theory at the level of conceptualization and at the level of execution, it aims finally to operate genealogically, to explain why and how it is that Joyce's most innovative textual ideas, what many literary critics see as his most idiosyncratic, least accessible, even least literary quirks, reappeared half a century later as the defining features of state-of-the-art computer programming. Ultimately, the question animating this chapter is this: is it possible that late twentieth-century programming theorists realize—however unconsciously—the authorial ideal with which Joyce flirted during the later years of his career? In other words, is it possible to see object-oriented programming as furthering and continuing some of the aesthetic ideals of late modernism?

3.1 Engineering Joyce

Joyce's vision of himself as a writer expressed itself most forcefully in his passionate, lifelong desire for a language that could transcend the parochial and conceptual constraints of ordinary language: "I'd like a language which is above all languages, a language to which all will do service. I cannot express myself in English without enclosing myself in a tradition," he said (Ellmann 397). Although he called English "the most wonderful language in the world," he was nonetheless sorely afflicted by its inadequacy (Ellmann 382). When asked, "aren't there enough words for you in English?", Joyce answered "yes" but added, "there are enough, but they aren't the right ones" (Ellmann 397). Not surprisingly, Joyce thrilled to the suggestion that *Ulysses* was actually a kind of code used to aid the Germans during the war (Ellmann 509–10).² Joyce sought to make words mean in ways they had never meant before: "I've put in so many enigmas and puzzles that it will keep the professors busy for centuries arguing over what I meant, and that's

²Though not among those who thought Joyce was a spy, Katherine Mansfield nonetheless fully participated in the feeling that *Ulysses* was more like code than communication: "It's absolutely impossible that other people should understand *Ulysses* as Joyce understands it. It's almost revolting to hear him discuss its difficulties. It contains *code words* that must be picked up in each paragraph and so on" (Ellmann 532).

the only way of insuring one's immortality," he crowed (Ellmann 521). Looked at in light of his desire for an alternative to the traditions and limitations of predefined English, Joyce's legendary "inaccessibility" is perhaps more properly understood as the result of his desire to formulate "a language to which all will do service." Such a language must be superficially inaccessible in order to move readers to "serve" it by reading it more closely, more inquisitively, and with greater attention to nuance than they otherwise would. Examined in this way, Joyce's inaccessibility becomes a means rather than an end. The end is the "language beyond language"—the language that is not fully written, or even fully known, by the author, but is instead provisionally organized by him into a "code" consisting of "puzzles" and "enigmas," and then put back out into the world for further, even endless, emendation, interpretation, vision, and revision.

Joyce's new language was a strategic assemblage of the old. As *Ulysses* progressed, the familiar, if deftly handled, stream of consciousness approach morphed into an exploration of how words themselves stream, and how that streaming—over time, across space, within and among minds—may become available to one's consciousness as an object of wonder, interest, appreciation, experimentation, frustration, silliness, and so on. As many critics have noted, Joyce came increasingly to think of writing as a choral event, something far more about associations, sounds, and layered repetitions than about plot or character *per se*. He became obsessed with allowing words to take on lives of their own: "A catchword is enough to set me off" he remarked (Ellmann 496). As Joyce explained to Budgen, "A man might eat kidneys in one chapter, suffer from a kidney disease in another, and one of his friends could be kicked in the kidney in another chapter" (436). The movement in Joyce's work is one of increasing abstraction, of language as systematic, beautiful design. In fact, Joyce liked to think of *Ulysses* as a prose version of his beloved *Book of Kells*: "you can compare much of my work to the intricate illuminations. I would like it to be possible to pick up any page of my book and know at once what book it is" (Ellmann 545).

One of the interesting effects of Joyce's transformation is that as he ceased to write things that were recognizable as "literature," he lost the ability to read novels, poems, and plays. These in turn became increasingly for him just another source of the "bits" that were becoming his

primary material. By the time Joyce was writing the *Wake*, he had his reader's block down to a science. Joyce's use of Twain's *Huckleberry Finn* in the *Wake* is a classic example: uninterested in actually reading the novel (or in having it read to him, when his eyes were too painful to read), he sent a copy off to David Fleischman with instructions to mark the book up according to his needs (Ellmann 699). Treating "literature" as so much junk to be sifted through, as a trash heap whose treasures lie not in the whole but in the reusability of its miscellaneous parts, Joyce signals the advent of a new literary language by wilfully and, some would argue, disrespectfully commandeering the odd parts of existing literary wholes. Whether *Huckleberry Finn* is a good, bad, or indifferent novel was utterly immaterial to Joyce; what mattered was what Twain's novel could do for the *Wake*, a book whose first word is "riverrun" and whose central symbol is the river. What Joyce does with *Huckleberry Finn* is akin to what Huck Finn does with the "p'simmons" he and Jim "borrow" as they float down the Mississippi. Because Huck wants and needs the p'simmons, he feels free to help himself to them, even though they don't belong to him. Likewise, Joyce plucks the aura and style of Twain's river symbolism whole from *Huck Finn* and drops it into his own prose wherever and whenever he feels like doing so. But because river references from *Huckleberry Finn* are doubly perfect for this book (Finnegan is Finn Again), it is as if Joyce is simply incorporating into his work a text that was somehow already his. Just as Joyce's head is full of "pebbles and rubbish," so literary history is full of polymorphic scraps just waiting for him to piece them together into new combinations.

Joyce's writing technique shifts to accommodate this shift in material, and one can watch him, during the course of *Ulysses*, stop writing as an "author" and begin writing as a sort of project manager. In a letter to Harriet Shaw Weaver, he wrote "I have not read a work of literature for several years. My head is full of pebbles and rubbish and broken matches and bits of glass picked up 'most everywhere" (Ellmann 512). Everyone Joyce knew became—whether they liked it or not—a collaborator in his project. "Since the material of *Ulysses* was all human life, every man [Joyce] met was an authority, and Joyce carried dozens of small slips of paper in his wallet and loose in his pockets to make small notes," Ellmann notes (412). Jolas recalls speaking with Joyce in a cafe: "Really, it is not I who am writing this crazy book," [Joyce] said in his whimsical way

one evening. ‘It is you, and you, and you, and that man over there, and that girl at the next table’” (13). “The sculptor August Suter was rather irritated to see how Joyce seemed to stage-manage conversations as if to use his friends as subjects for experimentation” (Ellmann 438–39). The writing itself proceeded along similarly opportunistic lines: “His method was to write a series of phrases down, then, as the episode took form, to cross off each one in a different colored pencil to indicate where it might go. Surprisingly little was omitted, but no one looking at the notesheets could have predicted how the fragments would coalesce,” Ellmann explains (416). The Joyce who wrote *Ulysses* was a Joyce intent on breaking free of the linear, character-centered constraints of narrative, and determined to break into a style that would center more on fusion than plot, more on coalescing fragments into wholes than on chronological coverage of evolving lives. This was a Joyce who could welcome editorial accidents as happy emendations: when Joyce’s friend McAlmon garbled the order of Molly’s soliloquy, Joyce liked the errors so much that he incorporated them into the book (Ellmann 514). This was also a Joyce who was completely unfazed by the difficulty his complex systems of internal references posed for translators (“There is nothing that cannot be translated,” he once commented).

Enormous, complex systems held an intrinsic appeal for Joyce. Recalling how Joyce became “deeply absorbed by the colossal conception” of Yeats’s *A Vision*, Eugene Jolas remembers that Joyce’s only regret about Yeats’s great system of abstractions was that “Yeats did not put all this into a creative work” (15). In turn, Jolas recalls Joyce describing the structural complexities of *Finnegans Wake*:

“There really is no coincidence in this book,” [Joyce] said during one of our walks. “I might easily have written this story in the traditional manner . . . Every novelist knows the recipe . . . It is not very difficult to follow a simple, chronological scheme which the critics will understand . . . But I, after all, am trying to tell the story of this Chapelized family in a new way . . . Time and the river and the mountain are the real heroes of my book . . . Yet the elements are exactly what every novelist might use: man and woman, birth, childhood, night, sleep, marriage, prayer, death . . . There is nothing paradoxical about this . . . Only I am trying to build many planes of narrative with a single esthetic purpose . . . Did you ever read Lawrence Sterne . . . ?” (11–12)

Joyce provides additional explanation for his radical new working method in a letter to Max Eastman:

In writing of the night, I really could not, I felt I could not, use words in their ordinary connections. Used that way they do not express how things are in the night, in the different stages—conscious, then semi-conscious, then unconscious. I found that it could not be done with words in their ordinary relations and connections. (Ellmann 546)

Taking words out of their “ordinary relations” and dispensing with “a simple, chronological scheme,” Joyce brings complex abstractions together with immense literary creativity. Synthesizing and consummating his earlier experiments along those lines, *Finnegans Wake* realizes Joyce’s lifelong fascination with structuring complex realities by weaving language into intricate patterns. Echoing his comment to Jolas that he was “trying to build many planes of narrative with a single esthetic purpose,” Joyce called his new technique “working in layers” (Ellmann 546).

Joyce naturally expected his readers to be surprised and puzzled by his new literary technique, but also expected that readers would discover the book’s intricacies as they came to understand his working method. Joyce was notoriously offended when this did not happen. When Harriet Shaw Weaver wrote to him in 1927 that “I am made in such a way that I do not care much for the output from your Wholesale Safety Pun Factory nor for the darkneses and unintelligibilities of your deliberately tangled language system” (Ellmann 590), Joyce grew distraught and felt obligated to defend not only his work but his new working method. Joyce complained to William Bird about the confusion and disappointment *Work in Progress* generated in its earliest readers:

I confess I can’t understand some of my critics, like Pound and Miss Weaver, for instance. They say it’s *obscure*. They compare it, of course, with *Ulysses*. But the action of *Ulysses* was chiefly in the daytime and the action of my new work takes place at night. It’s natural things should not be so clear at night, isn’t it now? (Ellmann 590)

On one level, Joyce’s remark to Bird feigns incomprehension: no one was more aware than Joyce himself of the extraordinary hermeneutic challenge his new work posed to its readers. However, Joyce also persistently believed that if only his readers could grasp the system underpinning the book’s “darkness and unintelligibilities,” his efforts in writing it would be vindicated. Another telling anecdote from 1931 shows Joyce’s conviction that his work would some day win him recognition and approval:

[Joyce] read from *Work in Progress* to a small group which included [Mary] Colum, and at the end asked her, “What did you think of it?” She replied with her accustomed

forthrightness, “Joyce, I think it is outside literature.” He did not make any comment then, but later he took Padraic Colum aside to remark, “Your wife said that what I read was outside literature. Tell her it may be outside literature now, but its future is inside literature.” (Ellmann 635)

Joyce’s odyssey with *Finnegans Wake* was twofold: first, struggling to finish the book itself amid the inhospitable circumstances of ill health, diminishing eyesight, uncertain financial conditions, and a mentally ill daughter; second, struggling to ensure the book’s reception as “literature,” despite its radical avant-garde experimentalism.

While writing the *Wake*, Joyce established not only a set of governing conceptual frameworks and structural devices, as he had with *Ulysses*, but a new set of governing metaphors to describe his compositional process. One of his favorites was that of the mechanic or engineer. Actively seeking out engineering metaphors for his authorial practices, Joyce famously wrote to Harriet Shaw Weaver that:

I am glad you liked my punctuality as an engine driver. I have taken this up because I am really one of the greatest engineers, if not the greatest, in the world besides being a musicmaker, philosopher and heaps of other things. All the engines I know are wrong. Simplicity. I am making an engine with only one wheel. No spokes of course. The wheel is a perfect square. (qtd. in Rabaté 113)

One of Joyce’s favorite engineering analogies was that of tunneling. When sculptor August Suter inquired in 1923 about the nature of Joyce’s new work, Joyce told him that “[i]t is like a mountain that I tunnel into from every direction, but I don’t know what I will find” (Ellmann 578). He used the analogy frequently in letters to Harriet Shaw Weaver, telling her “I feel like an engineer boring into a mountain from two sides. If my calculations are correct, we shall meet in the middle” (Ellmann 580); “I want to get as many sketches done or get as many boring parties at work as possible” (*Letters* 205); “I think that at last I have solved one—the first—of the problems presented by my book. In other words one of the partitions between two of the tunnelling parties seems to have given way” (*Letters* 222); and “[I am] pulling down more earthwork. The gangs are now hammering on all sides. It is a bewildering business” (*Letters* 222). On occasion, Joyce even adopts the guise of aircraft engineer: “I am also trying to conclude section I of Part II but such an amount of reading seems to be necessary before my old flying machine grumbles up into the

air” (Ellmann 564). Whether designing a perfect one-wheeled engine, boring into a mountain, supervising gangs of workers, or launching his grumbling old flying machine into flight, Joyce repeatedly figures his authorship within the framework of engineering design and execution.

It is not surprising to see Joyce deploy such engineering imagery to explain his aesthetic goals in *Finnegans Wake*: many of Joyce’s contemporaries were also very taken either with the idea of art as a form of engineering or with the idea of texts themselves as mechanical forms. In the experimental sculpture of Marcel Duchamp, the art photography of Alfred Stieglitz, the experimental symbolist poetry of Mallarmé, the geometric creations of the cubists, and the prophetic technophilia of the Italian Futurists, we can see the sheer range of modernist imaginative engagement with mechanism. Perhaps most poignantly, Joyce’s friend Paul Valéry calls the poem “a kind of machine for producing the poetic state of mind by means of words” (qtd. in Sypher 27). Valéry’s notion of the poem as a means of engineering a state of mind, and of words as the aesthetic engineer’s building blocks, speaks powerfully to Joyce’s own conception of language as the mechanism of modernist thought experiment. In keeping with this conception, Joyce carefully and explicitly built his engineering imagery into the text of *Finnegans Wake* itself. Its opening pages invoke the engineering marvel of the skyscraper, citing among others the Woolworth Building and the Eiffel Tower; they chart an evolutionary architectural progression from the “rushlit toofarback” of the ancestral Bygmester Finnegan through “a waalworth of a skyerscape of most eyeful hoyth entowerly, erigenating from next to nothing” (4). Finnegan, Joyce’s “master builder” (or “mysterbolder” [309.13]) in turn evolves into a vast electrical engine, “eelectrically filtered for allirish earths and ohmes” (309.24–310.1):

This harmonic condenser enginium (the Mole) they caused to be worked from a magazine battery (called the Mimmim Bimbim patent number 1132, Thorpetersen and Synds, Jomsborg, Selverbergen) which was tuned up by twintriadic singulvalvulous pipelines (lackslapping along as if their lifing deepunded on it) with a howdrocephalous enlargement, a gain control of circumcentric megacycles, ranging from the antidulibnium onto the serostaatarean. (310.1–8)

Broadcasting in “circumcentric megacycles” from antidulibnium (Dublin) across the entire serostaatarean (Free State of Ireland), the newly supercharged Finnegan/HCE becomes the

electro-mechanical successor to Roderick O’Conor, the “last pre-electric King of Ireland”; his “harmonic condenser engine” produces the text of *Finnegans Wake* itself.

Most importantly, however, *Finnegans Wake*’s running meta-commentary on its own composition frequently employs mechanical, electrical, geometrical, and mathematical metaphors to describe its complex, non-linear, system of assemblages. For example, the following passage describes the work of textual production as a perpetual motion machine made from Victorian cycles:

Our wholemole millwheeling vicociclotometer, a tetradomational gazebocon (the “Mama Lujah” known to every schoolboy scandaller, be he Matty, Marky, Lukey or John-a-Donk), autokinatonetically preprovided with a clappercoupling smeltingworks exprogressive process, (for the farmer, his son, and their homely codes, known as eggburst, eggblend, eggburial and hatch-as-hatch can) receives through a portal vein the dialytically separated elements of precedent decomposition for the verypetpurpose of subsequent recombination so that the heroticisms, catastrophes and eccentricities transmitted by the ancient legacy of the past, type by tope, letter from litter, word at ward, with sendence of sundance, since the days of Plooney and Collumcellas when Giacinta, Pervenche and Margaret swayed over the all-too-ghoulish and illyrical and innumantic in our mutter nation, all, anastomosically assimilated and preteridentified paraidiotically, in fact, the sameold gamebold adomic structure of our Finnius the old One, as highly charged with electrons as hophazards can effective it, may be there for you, Cockalooralooraloomenos, when cup, platter and pot come piping hot, as sure as herself pits hen to paper and there’s scribings scrawled on eggs. (614.27–615.10)

The “autokinatonetic” *Finnegans Wake* is a “wholemole millwheeling vicociclotometer.” an enormous processing and transmitting machine that receives “dialytically separated elements of precedent decomposition for the verypetpurpose of subsequent recombination.” In other words, it is a machine that decomposes and recombines its constituent elements: just like the *Wake* itself. As these examples show, the idea of discussing Joyce’s semantic architecture by way of comparison to other creative forms is as old as Joyce’s work itself.

So particularly compelling is the engineering imagery that Joyce built around and into *Finnegans Wake* that critics have subsequently adopted it wholesale, importing it into their own analyses of Joyce’s work. Most notably, A. Walton Litz, in his seminal *Art of James Joyce*, stresses the machinic quality of Joyce’s prose. Referring to Joyce’s “growing interest in formal—almost mechanical—designs” (37), he notes that “Joyce had a desperate and rather untidy passion for

order of any kind. All sorts of mechanical systems are used on the note-sheets to organize the diverse elements” (39). Litz subsequently searches for a more precise analogy, alternately referencing Valery Larbaud’s statement that *Finnegans Wake* is “a genuine example of the art of mosaic” and invoking the “critical commonplace” that Joyce’s writing techniques are analogous to those of musical composition. Although Litz elaborates extensively on the musical analogy, going so far as to declare that *Finnegans Wake* is “not ‘like’ music, it is a kind of music” (71), it is important to remember that Litz arrives at this claim by way of an unshakable conviction that Joyce’s prose is a highly engineered structure built around a “mechanical frame” (123).

Not only have critics translated Joyce’s engineering imagery into a critical lexicon for understanding *Finnegans Wake*, they have also allowed that imagery to evolve over time. Echoing Litz, Donald Theall elaborates on the mechanical quality of Joyce’s primary unit of meaning, the “bit”: “The words of *Ulysses* and the *Wake* are poetic machines that are assemblages of bits—fragments, clichés, typifications, words, syllables, letters, and etymological roots” (*James Joyce’s Techno-Poetics* xviii). Theall notes, too, that “building, burrowing, constructing, surveying, planning—‘machinic’ activities appropriate for an engineer—are just as primary components of the *Wake* as language, legend, and myth” (11). As if metaphors participated in technological progress, engineering images in Joyce criticism have morphed over the years into *computing* images. For instance, Harry Burrell argues that Joyce “used [*Finnegans Wake*] as a storage disc for all the bytes of information he accumulated over a lifetime” (6). Extending the computing analogy to the reader, Burrell mixes metaphors from computing and Artificial Intelligence theory to argue that the complex code of the *Wake* cannot be comprehended without a comparable recoding of the brain: “the difficulty you experience in attempting to understand *Finnegans Wake* arises from the conventional programming of your brain. It is necessary to reprogram it with Joycean software” (3). Such invocations of Joyce have even found their way outside literary criticism: neurophilosopher Daniel Dennett, who invokes the computer as a model of consciousness, refers to his concept of mind as “a virtual machine—the Joycean machine” (220).

Poststructuralist critics have absorbed a modernist, even specifically Joycean, notion of the machine underpinning the very possibility of writing. In *Allegories of Reading*, for instance, Paul

de Man asserts that the “machine is like the grammar of the text when it is isolated from its rhetoric, the merely formal element without which no text can be generated” (294).³ Jacques Derrida, too, has described his “grammatology” as a process of uncovering the machine within the text: “The originary and pre- or meta-phonetic writing that I am attempting to conceive here leads to nothing less than an ‘overtaking’ of speech by the machine” he writes in *Of Grammatology* (79). Therefore, it is not surprising that an extended account of Joyce as a programmer appears in Jacques Derrida’s 1984 essay “Two Words for Joyce.” In this piece, Derrida develops an extended analogy between Joyce’s prose and computer programming, calling Joyce “a sadistic demiurge” and describing him as a sort of maniacal superprogrammer. For Derrida, Joyce sets up

a hypermnesiac machine, there in advance, decades in advance, to compute you, control you, forbid you the slightest inaugural syllable because you can say nothing that is not programmed on this 1000th generational computer—*Ulysses*, *Finnegans Wake*—beside which the current technology of our computers and our micro-computerified archives and our translating machines remains a *bricolage* of a prehistoric child’s toys. And above all its mechanisms are of a slowness incommensurable with the quasi-infinite speed of the movements on Joyce’s cables. How could you calculate the speed with which a mark, a marked piece of information, is placed in contact with another in the same word or from one end of the book to the other? For example, at what speed is the Babelian theme or the word ‘Babel’, in each of their components (how could you count them?), co-ordinated with *all* the phonemes, semes, mythemes, etc. of *Finnegans Wake*? Counting these connections, calculating the speed of these communications, would be impossible, at least *de facto*, so long as we have not constructed the machine capable of integrating all the variables, all the quantitative or qualitative factors. This won’t happen tomorrow, and in any case this machine would only be the double or the simulation of the event ‘Joyce’, the name of Joyce, the signed work, the Joyce software today, joyceware. (147–48)

Derrida’s idea here is that *Ulysses* and *Finnegans Wake* deliberately close off the possibility of newness or invention. Because Joyce’s projects are so encyclopedic, so all-consuming, so totalizing, the “sadistic” author has always already pre-empted any possibility of linguistic or interpretive originality. Although *Finnegans Wake* has historically stood as one of the ur-texts of deconstruction, and Joyce’s linguistic experiments have enabled much of deconstruction’s critical energy, Derrida seems to find at the extreme limits of Joyce’s deconstructive textuality

³Geoffrey Bennington’s essay “Aberrations: De Man (and) the Machine” deals at length with metaphors of machinery in de Man’s criticism, particularly in de Man’s essay on Pascal.

a tendency toward the kinds of systematization and totalization that constitute the antithesis of the deconstructive ideal. One might say that for Derrida, “joyceware” represents an all-pervasive system that controls those unknowingly entrapped within it.

The image of a futuristic, almost impossibly fast “1000th generation computer,” beside which 1984’s computing technology is “a prehistoric child’s toy,” anchors this picture of Joyce’s pre-emptive prose. In Derrida’s analogy, computing technology functions as that which is at once entirely determined, and entirely determining; as a computer, Joyce’s texts are closed systems whose sheer computational power automatically usurps future utterances. The “sadistic” author becomes the impossibly dictatorial programmer, and the programmatic text becomes the all-embracing machine. In order to make his analogy, Derrida relies heavily on the bivalence of the word “program,” using it as a synonym for “control” and “appropriate” at the same time that he uses it to refer to the software (“joyceware”) that runs on his hypothetical hardware. For Derrida, the connotations of the word “program” are all negative; every layer of meaning it acquires in his textured usage simply consecrates the picture of Joyce’s prose, and the computer that is compared to it, as a hostile machine.

Two things are noteworthy about Derrida’s application of computing metaphors to Joyce’s prose. First we see how Derrida imports the engineering imagery that for Joyce was profoundly productive and positive, then warps that imagery, by way of the computer, to turn Joyce into an authoritarian sadist. Secondly, Derrida’s analogy rests less on any substantive knowledge about programming or computing history than an assumption that programming is necessarily a dictatorial, authoritarian act, one designed to limit, contain, and close off the play of meaning. The assumption is a false one, and as such the analogy is, too. An informed analogy between Joyce and programming theory yields a very different reading of the “programmatic” nature of Joyce’s prose, suggesting less that it is a means of controlling the play of meaning than of organizing words so as to maximize their semiotic potential. That analogy, which I will develop here, takes as its starting point the fundamental similarity between Joyce’s writing methods and the methodology of object-oriented programming.

3.2 Object-Oriented Programming

Before developing further the relationship between Joyce and the principles of object-oriented programming, it is necessary to cover some of the history of object-oriented programming and to explain how it gained popularity within programming circles as a methodological solution to the growing complexity of computer programs. This will involve describing the software crisis that emerged in the programming industry during the 1960s and 1970s, a crisis that was intimately related to the linear “procedural” or “functional” programming model’s inability to accommodate large structures of code without programs becoming untenably convoluted and labyrinthine. This section then explains the conceptual principles upon which object-oriented design rests, laying the groundwork for a reading of Joyce within those same principles.

“Of all the monsters who fill the nightmares of our folklore,” writes Fred Brooks in *The Mythical Man-Month*, “none terrify more than werewolves, because they transform unexpectedly from the familiar into horrors. For these, we seek bullets of silver that can magically lay them to rest.” In this classic 1974 work of software management theory, Brooks’s characteristically vivid werewolf metaphor describes the software development project that can morph all too readily into “a monster of missed schedules, blown budgets, and flawed products” (180–81). A hardened battler of such werewolves, Brooks honed his celebrated software management theories while working on one of computer programming history’s most celebrated disaster stories: IBM’s OS/360 software project. Between 1963 and 1966, IBM spent half a billion dollars and some 5,000 man-years completing the operating system component of the System/360 unified computing architecture described in Chapter One.⁴ Although IBM poured an unprecedented amount of money and human effort into the project, the software remained chronically behind schedule and the resulting product was mediocre in comparison to the accomplished System/360 hardware architecture. Altogether, OS/360 became an organizational, technical, financial, and public-relations embarrassment to IBM, which found its seemingly unassailable technical expertise humbled by the operating system’s

⁴Bob Evans, former vice-president of IBM’s Data Systems Division, which developed OS/360, gives his account of the development project and its many problems in “System/360: A Retrospective View.”

complexities. Users found the labyrinthine software unreliable and error-ridden; programmers charged with maintaining OS/360 found it difficult to repair outstanding bugs without generating new ones.

As Brooks takes pains to point out, the problem was not IBM or the OS/360 concept itself so much as it was the prevailing managerial and methodological conditions under which OS/360 was written. As such, the OS/360 project has less to tell us about incompetence at IBM than it does about the crisis that came to permeate the field of computer programming in the mid-1960s, a crisis that came about, ironically, because of phenomenally successful advances in other sectors of the computing industry. Rapid advances in core technologies during the 1960s compounded to generate exponential advances in computing hardware's power and speed; in the five years between 1960 and 1965 alone, computer memory and computer processing speed both increased tenfold, resulting in a hundredfold increase in overall computational performance (Cambell-Kelly and Aspray 196). These tremendous improvements in processing and storage capability meant that computers could run much more complex, elaborate, and sophisticated software programs than ever before; in turn, hardware manufacturers realized that the availability of such programs would greatly enhance the utility of their computers and so encourage more firms to switch from manual to electronic data processing.

Enticing as this logic was, these complex and elaborate programs were not readily forthcoming—as IBM so painfully discovered. Two factors combined to create this software vacuum. First, programmers' productivity had not increased at the same exponential rate as had computational capability; furthermore, as Fred Brooks famously demonstrated in *The Mythical Man Month*, trying to compensate for individual productivity limitations by adding more programmers to a project (the “brute force” approach favored by IBM in the 1960s) both slowed projects down and yielded markedly inferior final products. Second, the dominant programming paradigms of the mid-1960s were themselves limiting: designed to handle software projects containing 10,000 lines of code or less, they did not readily “scale up” to meet the needs of larger systems (Cambell-Kelly and Aspray 196). At a million lines of code, OS/360's unparalleled size and complexity was literally unmanageable, and the project's failure highlighted the organizational,

technical, and methodological deficits that had congealed to create the “software crisis,” a term that began to appear with distressing regularity in computing journals and managerial literature of the period.⁵

Responding to the reality of this crisis, a 1968 NATO conference purported to lay a new theoretical groundwork for managing large programming projects. Coining the now-ubiquitous term “software engineering” as the conference’s title, the organizers implied that programmers and corporations could resolve the software crisis by importing established engineering practices into the software development process. The conference announcement made no attempt to conceal the overarching rationale behind the name change, noting that “[t]he phrase ‘software engineering’ was deliberately chosen as being provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines that are traditional in the established branches of engineering” (qtd. in Cambell-Kelly and Aspray 201). The semantic shift is telling: “computer programming,” dogged with notions of failure, expense, flawed products, and public-relations disasters would become “software engineering”—theoretical, conceptually sound, and economically reassuring. Despite the shift in rhetoric, however, this attempt to “engineer” a managerial and/or methodological solution largely failed to resolve the software crisis. Programmers’ productivity has certainly improved since the 1960s, thanks to important advances in programming languages, diagnostic technologies, symbolic debuggers, and so on; and yet hardware capability has relentlessly outstripped these improvements in the programming process. Furthermore, the conceptual and managerial difficulties that began to plague the programming enterprise during the 1960s have still not been overcome. In a contentious 1986 paper, “No Silver Bullet—Essence and Accident in Software Engineering,” Fred Brooks gave his pessimistic outlook for the future of programming: “Not only are there no silver bullets now in view, the very nature of software makes it unlikely that there will be any—no inventions that will do for software productivity, reliability, and simplicity what electronics, transistors, and large-scale integration did for computer hardware. We cannot expect ever to see twofold gains every two years” (181). For

⁵In *From “Black Art” to Industrial Discipline: The Software Crisis and the Management of Programmers*, Nathan Ensmenger documents how the software crisis was conceptualized as a problem of programmer management. In particular, see 93–106 and 249–256.

Brooks, who understands computer programming as an art, something “creat[ed] by exertion of the imagination,” no silver bullet can magically transform the process into a mechanical *act* that would allow programmers’ productivity to keep pace with hardware advances.

Other programming theorists have challenged Brooks’s pessimism. In a 1990 response to Brooks, defiantly titled “There Is a Silver Bullet,” Brad Cox argued that a software development revolution only required a cultural shift away from the idiosyncratic craft-based approach employed in many development contexts. Noting that “computer software . . . is becoming the limiting strategic resource of the Age of Information much as petroleum is a strategic resource today,” Cox contended that the software crisis was ultimately a cultural problem rather than a technological one, and that “[t]he silver bullet is a cultural change rather than a technological change. It is a paradigm shift; a software industrial revolution that will change the software universe as the industrial revolution changed manufacturing” (10). What is needed, in Cox’s view, is a new *paradigm*, a new way of writing software that revolutionizes programming. Not to rethink entrenched ideas about what programming is and how it works is, for Cox, to fall into the same trap astronomers once fell into when their thought outgrew their models:

The Aristotelian cosmological model as extended by Ptolemy was once as entrenched and ‘obvious’ as today’s process-centered model of software development. Given any particular discrepancy, astronomers were invariably able to eliminate it by making some adjustment in Ptolemy’s system of epicycles, just as programmers can usually overcome specific difficulties within the software development paradigm of today.

But the astronomers could never quite make Ptolemy’s system conform to the best observations of planetary position and precession of the equinoxes. As increasingly precise observations poured in, it became apparent that astronomy’s complexity was increasing more rapidly than its accuracy and that a discrepancy corrected in one place was likely to show up in another. (215)

Cox’s parallel between tinkering with the Ptolemaic model of the universe and manipulating a woefully inadequate software development paradigm clearly announces what he believes programming needs: a conceptual reorientation analogous in scale to the Copernican revolution. The technology to enable that revolutionary reorientation, Cox assures us, already exists. Known as “object-oriented programming,” it is the elusive silver bullet solution to the software crisis. What was it about object-oriented technology that led Cox to see it as a paradigm shift in the way

we think about design methodology? A brief explanatory foray into the history and theory of object-oriented design illuminates the impulse at the heart of Cox's vision.

It is a mistake, albeit a common one, to regard object-oriented programming as an innovation of the 1980s and 1990s. Although object-oriented programming did not truly come into vogue until those decades—we see Cox arguing for its widespread adoption in 1990—some of its underlying principles may be found in the early Artificial Intelligence work of the 1940s and early 1950s (Berard 3). These ideas began to solidify into new programming approaches, somewhat ironically, at the very moment that the software crisis was first beginning to show itself. As programmers began to flounder under the weight of their impossibly complicated, exponentially lengthening projects, an MIT Ph.D. student named Ivan Sutherland was experimenting with a modular approach to software design. The result of Sutherland's research was the first interactive computer graphics program, SketchPad, introduced in 1962. Although the program was innovative in many respects—allowing its users to create images on-screen with a lightpen, for instance—its major design innovation from a programming standpoint was the way it employed the concept of *master drawings* and *instance drawings*. Once a programmer established constraints in the master drawing, SketchPad could define a multiplicity of instance drawings that followed these constraints—in other words, the instance drawings would inherit predefined attributes from the master drawing. Later programming theorists would recognize Sutherland's deceptively simple concept of defining new “instances” from preexisting, predefined “master classes” as a foundational breakthrough in the field of object-oriented design.

Experiments similar to Sutherland's took place elsewhere. At the Norwegian Computing Center between 1961 and 1967, Kristen Nygaard and Ole-Johan Dahl developed a discrete event simulation language named Simula I and a general programming language called Simula 67. Implementing many of the principles that would come to define modern object-oriented programming, Simula 67 would prove foundational for the creation of future object-oriented languages.⁶ A pivotal figure in advancing object-oriented technology was Alan Kay. While

⁶The computing industry has formally recognized Dahl and Nygaard's pioneering work in the field of object-oriented design: the Norwegian programmers received the Association for Computing Machinery's Turing Award in 2001, and the John von Neumann Medal in 2002.

studying at the University of Utah in the late 1960s, Kay became familiar with Sutherland's SketchPad program and began programming in Nygaard and Dahl's Simula. In the early 1970s, Kay, who coined the term "object-oriented," synthesized the ideas of Sutherland, Nygaard, and Dahl to create Smalltalk, the first fully-fledged object-oriented programming language.⁷

Although Sutherland, Nygaard, and Dahl heavily influenced Kay's work, biology supplied the flash of insight that led Kay fully to realize the object-oriented method. Looking for a simple, adaptable building block that would integrate his predecessors' insights, Kay, who had pursued a double undergraduate major in mathematics in molecular biology, hit upon the cell. Shasha and Lazere explain the significance of Kay's insight: "[t]he biological analogy suggested three principles to Kay. First, every cell 'instance' conforms to certain basic 'master' behaviors. Second, cells are autonomous and communicate with one another using chemical messages that leave one protective membrane and enter through another one. Third, cells can differentiate—the same cell can, depending on context, become a nose, eye, or toenail cell" (43). Crystallizing the essential structural potential of Sutherland's "master" and "instance" drawings, and the organizational principles of Nygaard and Dahl's Simula, computational objects modeled on cells exhibited traits that would later be named *polymorphism*, *inheritance*, *abstraction*, and *encapsulation*, each of which would become a core structural principle of object-oriented design theory.

These principles were very different from those that governed mainstream programming theory during the 1970s. The main precursor to object-oriented design, procedural (or "structured") programming, encouraged developers to build programs as sets of procedures or functions. Although this methodology, championed by computer scientists such as C.A.R. Hoare and Edsger Dijkstra, represented a major theoretical advance for computer science, it had significant limitations as a pragmatic programming methodology. Because data and functions existed in separate structures, and because data were often global, access to those data were often unregulated, and they could be modified in ways the programmer did not anticipate. Procedural programming,

⁷Kay describes the design of Smalltalk in his article "The Early History of Smalltalk." The language Smalltalk is still actively developed today: Contemporary implementations include Cincom Smalltalk, IBM's VisualAge Smalltalk, Dolphin Smalltalk, Smalltalk/X, and GNU Smalltalk.

seemingly elegant when used for short examples in computer science textbooks, quickly became messy when used for large development projects: as programs grew larger and more complex, bugs proliferated exponentially, and debugging became an arduous and time-consuming chore. By conceptualizing the object as the ontological starting point for a new programming methodology, object-oriented design theorists attempted to overcome many of the shortcomings and idiosyncrasies inherent in the procedural method. I mentioned above that the central attributes of object-oriented methodology are abstraction, encapsulation, inheritance, and polymorphism. I will now give a more detailed description of these principles, showing how they work together to shape the object-oriented design philosophy.

Rather than allowing data to exist globally in separate structures, object-oriented design methodology *encapsulates* data and functions (generally called *methods* in object-oriented terminology) together to form objects. Consider a simple object named `Sum` whose purpose is to add four numbers and report the total. The `Sum` object could contain four integers, `Int1`, `Int2`, `Int3` and `Int4`, a method for establishing the values of the integers, another method—perhaps a function called `Add()`—for summing the integers, and yet another method for retrieving the results. Because object-oriented design encapsulates data with its methods, the data cannot be modified by other unconnected functions. For instance, suppose we have another object called `DisplayTotal`, whose job it is to access the total calculated by `Sum` and display it on the screen. In order for `DisplayTotal` to access the data in `Sum`, the objects must be defined to allow that access; then `DisplayTotal` could send a message to `Sum` requesting the total of the summed integers and `Sum` could send the relevant data back to `DisplayTotal`. The principle of encapsulation thus allows the programmer to protect data from accidental modification—objects will only pass data to external functions when the programmer specifically defines those transfers.

Encapsulation allows an object to exist as an autonomous, self-contained entity. The independence of encapsulated objects in turn facilitates a high degree of *abstraction*. In Chapter One, I described how abstraction has worked historically to reduce programming complexity by hiding nonessential functions; object-oriented design further extends and advances this principle of abstraction. In the simple example above, the `DisplayTotal` object does not need to understand

how the `Sum` object calculated its output, and `Sum` does not know what `DisplayTotal` will do with the results. These principles can scale up to model complex real-world situations and transactions. If I order a book from Amazon.com, for instance, neither Amazon.com nor I need to know how or where the book was written, edited, manufactured, distributed and so on—such details are the responsibility of the book’s author and publisher. To complete the transaction, I need to pass only select, specific data to the online retailer (my name, address, and credit card information) and need to know only a limited amount of data about Amazon.com (that it is an online bookseller I can reasonably trust to deliver merchandise in return for payment). Neither Amazon.com nor I need to allow the other party access to global data about ourselves—Amazon.com does not need to know my marital status, my favorite foods, or whether I vote liberal or conservative; I don’t need to know details of Amazon.com’s upcoming promotional campaigns, earnings announcements, etc. In addition, neither Amazon.com nor I need to know the exact details of how the book will travel from the retailer’s warehouse to my home; we employ a third party, the postal service, to take responsibility for that particular set of decisions. Ultimately, neither Amazon.com nor the postal service needs to know why I want the book or how I will use it; indeed, the postal service does not even need to know what the package contains. The object-oriented paradigm thus models to a large degree the *information hiding* that occurs in the real world, where “objects” (authors, publishers, booksellers, postal workers, and readers) can interact and interrelate without needing to understand or even reveal all the animating methods and data that underpin those interactions.

The third core object-oriented trait is *inheritance*. When objects are created—or *instantiated* in object-oriented terminology—they may inherit many of their attributes from abstract classes, just as Sutherland’s instance drawings inherited attributes from master drawings. Instantiating two “car” objects `Car1` and `Car2`, a programmer may choose to inherit characteristics from an abstract `Car` class, then add unique data and methods to those objects. `Car1` and `Car2` may be different models, different colors, and so on, but they share essential commonalities: each car will have wheels, a steering mechanism, brakes, and so on. The ideal of inheritance, and the reason Cox and others promote object-oriented design as a “silver bullet” design solution, is that of code-reusability—rather than reinventing the wheel each time he wants to instantiate an object, a

programmer can simply draw on abstract classes and sub-classes for general templates that he then refines according to his specific needs. Given a large enough library of predefined classes and objects, a programmer, at least in theory, can assemble a large program from what are essentially reusable parts.

The final general core trait of object-oriented design, “polymorphism,” relates closely to inheritance in that polymorphism also aims to reduce the amount of code programmers must write from scratch. Polymorphism accomplishes this by allowing programmers easily to adapt inherited code to the requirements of specific environments, contexts, and commands. Kay’s analogy with the biological cell comes in useful here: all cells share the same DNA, but different genes are turned on in different cells to allow them to perform the many functions the body requires to sustain life—a skin cell, a red blood cell, and a brain cell may be understood as *polymorphous variants* of the same basic cellular object. As we will see, the same principles come into play with language; indeed, Edward Berard uses an epigraph from *Alice’s Adventures in Wonderland* to illustrate the principle of polymorphism of within object-oriented design:

“When I use a word,” Humpty Dumpty said, in a rather scornful tone, “it means just what I choose it to mean—neither more nor less.”

“The question is,” said Alice, “whether you can make words mean so many different things.”

As the procedurally-oriented Humpty Dumpty confronts the object-oriented Alice, we get a sense of the paradigm shifts that the new methodology necessitates.

The encapsulated, abstracted, polymorphous, inheriting object is thus designed to move code well beyond simple teleology and endow it with the power to break free of the constraints imposed by more linear programming techniques. The move toward modularity means that pre-written objects can be collected into software libraries—programmers can then draw on and adapt code for their purposes without having to start anew every time they need to accomplish an everyday task such as drawing an icon on a screen, or creating a drop-down menu. Thus advocates of object orientation argue that the object-oriented methodology is best suited for use in developing today’s increasingly complex systems.

What does all this have to do with Joyce? “Je suis au bout de l’anglais [I am at the end of English],” Joyce declared in a fit of frustration in 1922 (Ellman 546), just as he was shifting focus from *Ulysses* to *Finnegans Wake*. “What the language will look like when I have finished I don’t know,” he declared four years later, having established the *Wake*’s working method. “But having declared war I shall go on ‘jusqu’au bout’” (Ellmann 581). It will be the contention of the third section of this chapter that structurally speaking, the modular, omnidirectional structure of Joyce’s writing “jusqu’au bout d’anglais” has much in common with the structure of an object-oriented program. What object-oriented design essentially did was to move code beyond a simple narrative structure—beyond the assumption that a program, like a plot, contains a distinct beginning, middle, and end—and into a realm of expression far more complicated, far more powerful, and far more aesthetically sophisticated than the forms from which it was derived. So, too, with Joyce. Like the object-oriented programmer, Joyce’s technique consists of implicating the whole in the part, of making each scene, character, and sentence point to and even contain the entire work—in short, of building words, characters, and scenes into encapsulated, polymorphous units that inherit their attributes from predefined classes. In what follows, I turn to programming philosophy as a means of approaching *Finnegans Wake*’s structural problems. Reading the *Wake* as an elaborately woven piece of prose-code, I will demonstrate how object-oriented programming philosophy makes newly visible aspects of Joyce’s text that have not hitherto been productively synthesized or properly understood.

3.3 “Joyceware”: The Object-Oriented *Wake*

An attempt to read Joyce’s later fiction through the lens of programming theory does have precedent in the critical literature. In her 1989 book *Writing Joyce: A Semiotics of the Joyce System*, Lorraine Weir suggests that “the operations of John McCarthy’s system, ‘LISP,’ provide another way of configuring that fugal arborescence of textual paradigms [that is] a principal structuring mode in *Ulysses* and *Finnegans Wake*” (94). Recognizing the potential of computer language design theory to illuminate the complexities of Joyce’s texts, Weir selects LISP for its

“hierarchical (rather than linear) structure,” for its capacity to define “new functions in terms of other functions that are already defined,” and for the “nodal structure of a LISP tree” (95). Importing McCarthy’s LISP onto the scene of Joycean criticism allows Weir to identify and begin to define an overlap between programming theory and Joycean method; in particular, she notes how LISP is “bound by the multiplicative drive of its sets and subsets to reenact the operations of the system” and how, in LISP, “form *is* act in all senses, and act a recombinant sequence of recursive definitions” (96). Weir explained the specific connections between the nodal structures of LISP and those of *Finnegans Wake* as follows:

LISP memory . . . operates in terms of a sequencing of lists built as trees whose structures develop as programmed and according to required configuration, storing not only the final term or result of the sequence but also all of the connections among the nodes in the tree or list. The mnemonic hierarchy works by moving back from results to prerequisites or requirements and thus fixes a hierarchy of goals much as the “verbivocovisual” or eye/ear code does in *Finnegans Wake*. (95)

Weir’s experimental interpretive strategy never fully rises to the level of method; her treatment of the conjunction between programming language and Joycean language is too brief; moreover, her unexamined assumption that LISP’s hierarchical data structures may be mapped onto *Finnegans Wake* makes it impossible for her to account for the associative semantics of Joyce’s novel. Her analysis is thus truncated at precisely the point where its potential to open up Joyce scholarship is most intense: as I will go on to show, the modular structure of object-oriented programming provides a much more compatible and critically illuminating paradigm for reading Joyce’s late work. This section seeks to mine the untapped potential that Weir passed over by developing an extended analysis of how Joyce’s modular compositional practice operated according to the structural principles that are recognizable today as those of object-oriented programming.

The principle features of object-oriented theory (encapsulation, abstraction, inheritance, and polymorphism) combine to allow programmers an unprecedented amount of control over enormously complicated systems. They are techniques for managing complexity, ways of ordering layered, interconnected systems that, far from *simplifying* the whole, harness the whole’s complexity. Like an object-oriented program, Joyce’s work is at once extraordinarily complex and highly organized. Like an object-oriented programmer, Joyce was always concerned to

give the reader sets of conceptual tools that the reader could use to navigate his work. More to the point, Joyce develops a hermeneutic scheme whose organizational principles parallel those of object-oriented programming. Long before object-oriented programming languages were created, Joyce independently devised similar techniques as a compositional and methodological alternative to traditional narrative structure. Working in a highly modular manner whose dense conceptual nesting has yet to be fully apprehended by critics, Joyce used what would today be defined as “classes” and “objects” to structure the *Wake*, relying on strategies very like those of encapsulation, polymorphism, and inheritance to structure the flow of words as a flow of data. I use the term “data” deliberately here to describe the coded, informatic quality of Joyce’s language, which is often more interested in words as markers of patterns than as referents, and which allows this impulse toward pattern to become a guiding one, such that the “meaning” of the *Wake* lies not in the thematic or symbolic significance of individual words, but in the accumulation and repetition of word patterns.

Structural and genetic readings of late Joyce have unconsciously begun just such an “object oriented” reading of the *Wake*. Clive Hart’s definitive 1962 study, *Structure and Motif in Finnegans Wake*, is one of the earliest examples of this strand of Joyce criticism. In that work, Hart’s description of how the *Wake* makes meaning strongly evokes two of the defining principles of object-oriented design theory:

The primary energy which maintains the highly charged polarities of *Finnegans Wake* is generated by cycles of constantly varied repetition—‘The seim anew’, as Joyce puts it. The stronger the pre-established expectancy, the wider can be the variations played on a word or motif. A pun is effective only when its first term is vividly prepared for by the context. By using a vocabulary and style packed with well-worn units Joyce is able to play on what the psychologists call the reader’s ‘readiness.’ As with the basic style, so with the more specialized motifs: if Joyce builds them up from familiar phrases he is absolved from the need to establish familiarity with their shape in the early parts of the book (which would be out of keeping in a really cyclical work) and is immediately able to make the widest punning excursions while remaining sure of his readers’ powers of recognition. (31–32)

In Hart’s paradigm, which describes the dynamic interaction between reader and Joyce’s text,

words derive much of their local meaning from the book's wider structural patterning.⁸ In this, they operate less as signs than as *objects*; they inherit significance from past usages and related word plays, while at the same time morphing to suit their specific syntactical context. An object-oriented paraphrase of Hart's work would say that Joyce's portmanteau word-units display the object-properties of inheritance and polymorphism.

As Hart's work suggests, there is a striking convergence between critical characterizations of the *Wake*'s structure and object-oriented programming theory. For example, in his analysis of the *Wake*'s genetic evolution, David Hayman proposes that the series of preliminary sketches Joyce drafted in 1923 constitute the book's "nodal macrosystem" (9). Much of Joyce's work thereafter involved elaborating, deepening, and interlinking the primary nodes that he generated early on. Similarly, Shari Benstock notes that "there are stories within stories, each telling embedded in another frame of receding concentric rings," arguing that we need a "multiple 'frame of reference' allowing for a maximum number of correspondences between dreams and letters" (165). Joseph Valente adopts a different sort of metaphor for the cyclical process Benstock posits, referring to Joyce's technique of dropping words into different contexts and watching how they behave as "contextual polyphony" (59). These are all precise, if unselfconscious, descriptions of a textual system centered around the principles that govern object-oriented design.

Vicki Mahaffey's *States of Desire* describes an experimental reading approach that builds on this emergent understanding of the *Wake* as a built system of artifacts that both convey meaning associatively and acquire it dynamically as the context shifts, or as the "program" of *Finnegans Wake* runs. Drawing on the "machinic" textual analyses of Deleuze and Guattari, Mahaffey proposes a reading strategy that "sees words as temporary 'assemblages' capable of being taken apart and assembled differently, plugged in elsewhere, recontextualized; such assemblages are sites of multiple interconnections and potential recombination" (8). Like Hart, Hayman, Benstock, Valente, and Mahaffey are all describing Joyce's patterning as a system of dynamic objects that inherit their characteristics and rules from a wider library of languages, designs,

⁸Margot Norris, too, notes that "we find in *Finnegans Wake* that intellectual shift which locates meaning in relationships and structure rather than in content" (3).

formulae, histories, myths, games, and so on. As Jean-Michel Rabaté puts it, “The economy of *Finnegans Wake*, its amalgamative alchemy” depends on “a few important notions, such as those of codes, series, and performative language games . . . A single element is enough to trigger off a chain-reaction, and to magnetise the other elements of a latent group” (112–18).

Robert Scholes, Joseph Kestner, and others have invoked Jean Piaget’s structuralist theories to read Joyce’s work. In *Structuralism* (1971), Piaget proposed that “the notion of structure is comprised of three key ideas: the idea of wholeness, the idea of transformation, and the idea of self-regulation” (5). Adopting this Piagetian triad, both Kestner and Scholes derive an aesthetic that they believe describe Joyce’s writing (Scholes 185). Although their Piagetian structuralism focuses more on Joyce’s earlier fiction, Kestner’s concern with the “structured layer of the [Joycean] palimpsest” (32) shows how this structural model is perhaps most relevant to *Finnegans Wake*. In the final section of this chapter I discuss the genealogical relationships between Piagetian notions of structure and the design philosophy of object-oriented programming; for now I simply note that the Piagetian ideas of wholeness, transformation, and self-regulation bear strong formal similarities to the object-oriented notions of encapsulation and polymorphism.

Together, these critical formulations express a shared sense that Joyce’s late prose is founded upon a peculiarly dynamic infrastructure, one composed of mobile, highly defined objects that both move freely throughout the text and are bound by strict rules about how they interact with one another. At the same time, none of these formulations develops a definitively precise architectural vocabulary for characterizing the signifying activity that it describes. The commonalities between these descriptions and the model of meaning generated by object-oriented design theory suggests how much structural readings of Joycean prose have to gain from a conscious, thoroughgoing incorporation of object-oriented methodology into Joyce criticism. Object-oriented theory not only provides a conceptual framework for reading Joyce’s spatial approach to prose, but also supplies a conceptual framework for grouping some of the metaphors critics have used to describe Joyce’s compositional methods.

The sigla that simultaneously denote character and symbol, myth and archetype, are the most “object-oriented” aspect of Joyce’s compositional methodology. Joyce devised the sigla in

the early months of 1923 (the period when David Hayman says that he developed his “nodal macrosystem”). Initially Joyce’s shorthand for the *Wake*’s various characters, the sigla gradually became, in the words of Jean-Michel Rabaté, “the underground logic controlling the balance of individual chapters and books” (80). Joyce’s sigla are his “objects”; the principal building blocks of the *Wake*’s structural system, they orient and define his prose.

The sigla are elaborate condensations, a sort of structural shorthand that begins as a system for denoting character and evolves over time into a mechanism of compressing entire historical and epistemological systems. The sigla structure vast arrays of eccentric references into concentrated units; those units in turn contain the potential for an infinite opening outward of its accumulated allusive force. In turn, the sigla interact with one another as a means of mutual definition—Margot Norris notes that “[w]e find in the *Wake* not characters as such but ciphers, in formal relation to each other” (4). Deleuze and Guattari perfectly capture the compressed movement of sigla in their description of “assemblages” in *A Thousand Plateaux*: “[T]he material or machinic aspect of an assemblage relates not to the production of goods but rather to a precise state of intermingling of bodies in a society, including all the attractions and repulsions, sympathies and antipathies, alterations, amalgamations, penetrations and expansions that affect bodies of all kinds in their relation to one another” (90). If one were to substitute the word “sigla” for the phrase “bodies of all kinds,” one would have a precise description of how *Finnegans Wake* condenses a multitude of intersecting historical, mythic, and fictional patterns.

Joyce’s revolutionary condensations originate in a dissatisfaction with traditional characterization, a fact that is crucial for our understanding of how the *Wake*’s sigla operate as objects. That dissatisfaction culminated during the writing of *Ulysses*, when Joyce became frustrated with the static quality of his most completely “formed” character, Stephen Dedalus. When Ezra Pound suggested that Joyce could improve *Sirens* by relegating Bloom to the background and bringing Stephen forward, Joyce rejected the idea, commenting to Frank Budgen that “Stephen no longer interests me. He has a shape that can’t be changed” (Ellmann 459). A largely autobiographical character formed according to the conventions of nineteenth-century realism in Joyce’s earlier *Portrait of the Artist as a Young Man*, Stephen could not evolve along

with Joyce's changing ideas about narrative. As Joyce began to experiment more freely with the formal potential of prose, Stephen Dedalus ceased to fit into the narrative schema that originally were designed with him at the center. In his late work, Joyce sought to create characters whose shape *could* be changed according to a pre-defined, consistently applied, yet enormously flexible set of rules and constraints. The concept of the sigla—at once character and far more than character, possessing definite traits and yet infinitely mutable—answered Joyce's need.

A primary trait of sigla is what Roland McHugh has helpfully described as "personality condensation." For McHugh, the sigla are "personages," which are in turn "fluid composites, involving an unconfined blur of historical, mythical, and fictitious characters, as well as nonhuman elements." Moreover, these personages are also semiotic systems: "Joyce's technique of personality condensation is ultimately inseparable from his linguistic condensation. Coincidences of orthography and pronunciation are enforced with indifference to the ostensible logic of their past. That which is not coincidence is pared away, and the greater the similarity of two persons' names, the more usefully their personalities conjugate" (10). Through the creation of sigla, then, Joyce was able to create characters that were not bound by time, place, or even their own bodies; in the *Wake*, characters morph and mutate, moving and shifting as Joyce's narrative needs dictated. Instead of limiting his stylistic experiment as Stephen had, they became its centerpiece.

As the anchoring structural elements of the *Wake's* architectural system, the sigla contain within them the key to Joyce's vision of the impact he wanted the *Wake* to have on its readers. The overall effect managed by the sigla, McHugh notes, is not one of order or clarity, but of associative overload. The sigla are not intended to condense discrete ideas, or even defined patterns, so much as they are designed to manage Joyce's attempt to produce in the reader a state of mind McHugh labels "psychic saturation" in which "the mind fails to retain and reconcile a superfluity of levels" (10–11). The notion that producing a saturation-effect in the reader would resolve Joyce's difficulties with traditional notions of character may seem counterintuitive at first glance: a character is not a character if it does not have firm boundaries and a coherent presentation. Joyce was approaching the problem of characterization, and the related issue of narrative innovation, from the viewpoint of the night, however. He was adamant that *Finnegans Wake* was not a work

that proceeded according to the rules of everyday life, but that instead observed the strange internal logic of dreams. In this, Freud was his inspiration and his guide; Joyce's conversion of character to compressed sigla, and his use of personality condensation to release vast amounts of submerged textual energy, grows out of his reading of Freud's *Interpretation of Dreams*. There, Freud explains the logic of dreams in terms that Joyce adapts for his own expressive purposes in *Finnegans Wake*.

In writing *Finnegans Wake*, Joyce was particularly influenced by Freud's model of condensation.⁹ For Freud, "condensation" described the process by which far-ranging significance and multiple layers of meaning (what Freud calls "dream-thoughts") are compressed into individual elements that are recalled after waking as the dream (or, in Freud's terms, "dream-content"). In *The Interpretation of Dreams*, Freud focuses on two principles of condensation: personality condensation and linguistic condensation. These are at once distinct and interpenetrating; both are mutually constitutive. For Freud, in dreams "new unities are formed—in the shape of collective figures and composite structures—and intermediate common entities are constructed" (330). Freud gives an example from his own dreams:

On one occasion a medical colleague had sent me a paper he had written, in which the importance of a recent physiological discovery was, in my opinion, overestimated, and in which, above all, the subject was treated in too emotional a manner. The next night I dreamt a sentence which clearly referred to this paper: *'It's written in a positively norekdal style.'* The analysis of the word caused me some difficulty at first. There could be no doubt that it was a parody of the [German] superlatives *'kolossal'* and *'pyramidal'*; but its origin was not so easy to guess. At last I saw the monstrosity was composed of the two names *'Nora'* and *'Ekdal'*—characters in two well-known plays of Ibsen's (*A Doll's House* and *The Wild Duck*). Sometime before, I had read a newspaper article on Ibsen, by the same author, whose latest work I was criticizing in the dream. (331)

As this example elegantly shows, a dream's compressed words may also compress personality; likewise, personality compression may take place via linguistic condensation. Ibsen's characters merge to characterize the overwrought tone of Freud's colleague's argument; moreover, they merge both into a composite figure and into a portmanteau word—within the logic of the dream, Freud's

⁹For a genetic study relating Joyce's use of Freud to the process of sigla-formation, see Daniel Ferrer's "The Freudful Couchmare of Δ d: Joyce's Notes on Freud and the Composition of Chapter XVI of *Finnegans Wake*." For Joyce's reliance on Freud to integrate elements of the *Wake* see Margot Norris, especially 15–22.

objections to his colleague's argument are thus objectified as a single enormously telling adjective, one that in turn contains within it the combined force of Ibsen's own psychological drama.

In the *Wake*, Freudian principles of personality condensation and linguistic condensation form the structural rules for the sigla that govern the architecture of the whole. McHugh tacitly acknowledges Joyce's conceptual debt to Freud when he defines the sigla in a vocabulary that is openly—if not explicitly—drawn from *The Interpretation of Dreams*: “Joyce's technique of personality condensation is ultimately inseparable from his linguistic condensation.” “Personality condensation” and “linguistic condensation” are not, we have seen, McHugh's terms, but Freud's. Likewise, the notions that the one type of condensation begets and even necessitates the other, and that together they work to produce something akin to psychic overload, or “saturation,” belong to Freud. One might say that Freud's *Interpretation of Dreams* forms the structural unconscious of the sigla.

According to Freud, the remembered elements of dreams are the manifest signs of a meaning that is always by definition latent, even repressed: “When we reflect that only a small minority of all the dream-thoughts revealed are represented in the dream by one of their ideational elements, we might conclude that condensation is brought about by *omission*: that is, that the dream is not a faithful translation or a point-for-point projection of the dream-thoughts, but a highly incomplete and fragmentary version of them” (315). The formulation describes perfectly Joyce's compositional process, which formally registered sigla as a notational means of pointing toward a mass of unrecordable association. In his notebooks, the sigla might be said to mark the dream-content of the streaming, unending dream-thoughts that would ultimately form the text of the *Wake*; beginning with the compositional analogue of dream-content, Joyce moved backward—or inward—according to a recognizably Freudian hermeneutic. As Freud works backward from “ideational element” (315) to dream-thought, so Joyce works from sigla to the vast, inchoate series of associations that each crystallizes, embroidering, deepening, and complicating *Finnegans Wake* as a means of approximating the continuous turbid current of dream-thought.

Joyce thus translated Freud's portrait of the psychic process of dream-building into a theory of narrative structure. Thomas Jackson Rice calls Joyce's sigla “‘activity rules’ for [characters’]

interaction”; Donald Theall has noted Joyce’s interest in the mathematical definition of types and their relation to the sigla: “Joyce read Russell’s *Introduction to Mathematical Philosophy* with a particular interest in the chapter on the theory of types. . . . [H]e used the formality of mathematical logic in the construction of relationships between sigla. . . . He apparently sensed the problems of ‘strange loops’ and the infinite regression of classes, hence also the concept of ‘meta-’levels.” In his use of sigla, Joyce seemed to be looking for mathematical formulae that would allow him to define the attractions, repulsions, amalgamations, and metamorphoses of his characters” (*James Joyce’s Techno-Politics* 167). Combining the concepts of personality and linguistic condensation with his own mechanical and mathematical rules for building sigla, Joyce adapted Freud’s hermeneutics of the dream into a new technique for engineering a form of textuality that contained but did not precisely tell a story; that developed character through accretion and pattern rather than through description and event; and that thereby bypassed the linear model of development associated with the nineteenth-century realist novel in favor of a more fluid, undefined, ultimately more mimetic style. In this way, Joyce derives a central component of his “object-oriented” narrative technique from his reading of Freud’s understanding of dream-content as a condensation of much more amorphous, inaccessible, associative, and submerged dream-thoughts. If Joyce’s sigla are his objects; Freudian condensation is the means by which they achieve what object-oriented programming theory describes as “compression.”

Both Freudian theories of condensation and programming theories of objects share a notion of textual concentration: just as condensation is the operative feature of dreams, so “compression” is a primary trait of objects. The object-oriented programming theorist Richard Gabriel describes “compression” as “the characteristic of a piece of text that the meaning of any part of it is ‘larger’ than that piece has by itself. This is accomplished by the context being rich and each part of the text drawing on that context—each word draws part of its meaning from its surroundings” (5). Gabriel’s description of a compressed object could as easily describe Joyce’s sigla, which compress or condense meaning in precisely the same way. The profoundly literary quality of compression was not lost on Gabriel, who expressly noted that insofar as objects could be designed as compressions of larger, unreferenced libraries of code, those objects were also literary

artifacts, written according to time-honored aesthetic technique: “A familiar example from outside programming,” he notes, “is poetry, whose heavily layered meanings can seem dense because of the multiple images it generates and the way each new image or phrase draws from several of the others” (5). For Gabriel, the better the program’s compression, the more literary the program.

Gabriel emphasizes, however, that the cost of compression can be quite high; indeed, that the costs increase as the quality and extent of the compression do: the more compressed an author’s program is—the richer its referenced context—the less “habitable” it is, the less able other programmers are to insert themselves into the code and continue to write it. Compression is signature, a form of shorthand. The tension Gabriel traces between the quality of a program’s compression and “code habitability” is a defining tension within Joyce’s work. Indeed, one of the most bizarre episodes in Joyce’s career—Joyce’s near-decision to have another writer finish the *Wake*—may be explained in those terms: Joyce’s belief that a little-known novelist named James Stephens could complete the *Wake* is perhaps best understood as a sign of his faith in the structural “habitability” of his book’s design.

Joyce was very far from seeing himself as a singularly original, self-contained artist. He felt for example that his mind was written over by the writing he had read. And indeed it was. *A Portrait of the Artist as a Young Man* was very clearly a rewriting of Meredith’s *The Egoist*. *The Dead* drew heavily on the work of George Moore (Ellmann 250–51). And *Ulysses* combined a borrowed mythical structure with the recorded words and lives of just about everyone who had ever made an impression on its author. “They are all there [in *Ulysses*], the great talkers, they and the things they forgot,” he said (Ellmann 524). (All, that is, except his brother Stanislaus, whose insistent claims to certain ideas in the book met with the special contempt Joyce reserved especially for him [Ellmann 531].) Joyce made a virtue of necessity, however, and what began as worry about whether he was original quickly became pride at his ability to do more with ideas than their originators could. *The Dead* was a case in point: though Joyce had taken the idea for the ending from Moore, Moore could only regard what Joyce had done with awe, remarking without irony in a letter to Yeats that he wished he had written a work that was in fact a rewriting of his own work. In 1918, Joyce remarked to his friend Frank Budgen that he was more of a developer than an originator of ideas:

“Have you ever noticed, when you get an idea, how much *I* can make of it?” (Ellmann 439).

Joyce preferred to represent his work as a collaborative elaboration of ideas rather than the singular effort of one. Ellmann says of Joyce composing *Ulysses*, “He was never a creator *ex nihilo*; he recomposed what he remembered, and he remembered most of what he had seen or had heard other people remember” (364–65). Joyce’s refusal to own his originality as anything other than a type of sophisticated book-keeping is a telling gesture, one that points counterintuitively but unmistakably to a notion of authorship as something separate from an individual author, something only provisionally owned, something never authentic.

In 1927, frustrated with the frustration of his readers, suffering from serious health problems, and plagued by family disturbances, Joyce came quite close to bequeathing the unfinished *Finnegans Wake* to James Stephens, who he imagined could continue the project once he understood the principle (or code) of its creation. Joyce explained his rationale in a letter to Harriet Shaw Weaver:

As regards that book itself and its future completion I have asked Miss Beach to get into closer relations with James Stephens. I started reading one of his last books yesterday *Deirdre*. I thought he wrote *The Return of the Hero*, which I liked. His *Charwoman's Daughter* is now out in French. He is a poet and Dublin born. Of course he would never take a fraction of the time or pains I take but so much the better for him and me and possibly for the book itself. If he consented to maintain three or four points which I consider essential and I showed him the threads he could finish the design. JJ and S (the colloquial Irish for John Jameson and Son's Dublin whiskey) would be a nice lettering under the title. It would be a great load off my mind. I shall think this over first and wait until the opposition becomes more general and pointed. (Ellmann 591–92)

Joyce liked the coincidence that the name “James Stephens” combined his own name with that of Stephen Daedalus, and delighted in the discovery that Stephens was born in Dublin at 6 A.M. on February 2, 1882—just as Joyce himself was. At the time, Joyce was fixated on the possibility of handing his project over to someone else. He turned the idea over in his mind for over seven months and even discussed it with Stephens (Ellmann 592–93).

Finnegans Wake has come to be wholly identified with Joyce, so much so that it is seen as something no one else could have imagined, let alone produced; so much so that it seems virtually impossible to comprehend how Joyce could envision delegating the *Wake* to someone else. How

could Joyce see himself as inessential to such an idiosyncratic, complex, and even inaccessible project? How could the author of one of the most “original” works of literature ever imagined even consider passing it on to someone else to finish? What must Joyce’s idea of authorship have been, and what must have been his understanding of what the *Wake* was, that he could seriously envision leaving his masterpiece to be finished by another, far lesser writer?

Richard Ellmann calls Joyce’s idea “one of the strangest ideas in literary history” (591), but the notion of handing original work off to others to complete is not a strange idea at all in *programming* history. As Eric Raymond points out, in open source programming culture is not only common for programmers to “pass the baton” when they can no longer maintain a project, but it is their duty to do so. Such transfers are, in turn, possible because software is written according to a set series of coding rules, and each new creation takes shape as a singular design within those rules. So too with Joyce, who was so convinced of the structural precision of *Finnegans Wake*—which he called “mathematical” (Ellmann 614)—that he believed handing his work over to Stephens was a simple matter of “showing him the threads” so he “could finish the design.”

What is inexplicable by the standards of literary history—Joyce’s strangely detached relation to the product of his creative genius—becomes perfectly understandable, even sensible, when viewed as part of another history, that of object-oriented design. As we have seen, Joyce’s attitudes toward writing—toward creating original text, toward using the work of others, toward compiling discrete units into elaborately designed wholes, and toward designing those wholes in regular, orderly, reproducible ways—have much in common with the attitudes of object oriented programmers; indeed, computing history has much to tell us about those aspects of Joyce’s creative process that have seemed, to literary historians, to make least sense. When Joyce suggested bequeathing the *Wake* to James Stephens in 1927, we can see more than just frustration with the work, with his family, with his eyes, or with his readers. This moment in Joyce’s career represents an evolving understanding *Finnegans Wake* as a future-oriented project whose eternal construction could continue dynamically if its readers, and potential authors, could only grasp the threads. When seen as the act not of one who sees his work as the unique expression of his singular artistry but of one who sees his work as part of a larger, collective design, Joyce’s impulse to hand the *Wake* over the

James Stephens no longer seems one of the strangest ideas in literary history. That decision, rather, is the product of a complex and evolving understanding of what it meant to be an author, a writer, and an artist.

Did Joyce want *Finnegans Wake* to be a “habitable” text? Yes and no. Joyce proclaimed himself dismayed when people pronounced *Finnegans Wake* unreadable—yet his boast that it would take academics hundreds of years to unravel the text belies his claim to feel distress at puzzling even the most sophisticated readers. The paradox of Joyce’s simultaneous desire to be understood and his contempt for readers’ failed efforts to understand is easily resolved, however. Joyce’s contempt for those readerly “failures” was primarily a contempt for the simplistic notions of mastery such readers inevitably brought with them to the reading of the *Wake*. The *Wake* is thus not only uninhabitable, but deliberately so, to an educated adult audience bent on locating the “key” or the “truth” of the text (academics in particular tend to get caught in the trap of trying to master Joyce’s encyclopedic metaclasses, an enterprise that only creates an experience of frustration and insufficiency, in part because one cannot absorb all that Joyce did, in part because doing deep history on the *Wake*’s references is not the way to get at how or why Joyce was using them). By contrast, however, the *Wake* was designed to be entirely habitable to readers who were not obsessed with controlling their reading experience or definitively isolating Joyce’s meaning. More specifically, the *Wake* was envisioned as a text that would ideally be read from the playful, open-minded vantage point of a child.

In the next section, I will explore Joyce’s fascination with the child as a model reader. For Joyce, children possess a linguistic fluidness that the fixated adult reader does not. They have the ability to laugh at patterns or puns that they not necessarily grasp, and they are willing to keep going when they are not in full command. These qualities of imaginative acceptance, at once creative and equable in the face of confusion, were, for Joyce, the signally redeeming features of a childlike approach to reading. The concluding section of this chapter develops this idea, charting the origins of the object-oriented programming paradigm in modernist theories of child psychology in order to show how closely tied Joyce’s stylistic innovations were to an equally innovative concept of how human beings best learn, comprehend, and imagine. My aim will be to demonstrate

both the deep conceptual kinship Joyce's prose shares with complex programming languages, as well as the benefits to be had from importing the logic of those languages into Joyce criticism.

3.4 A Modernist Genealogy of Object-Oriented Design

Joyce's early work is permeated with contempt for formal education. In its intellectual pretensions, its systemic inefficacy, and its almost certain failure, Joyce saw a self-defeating system that did more to impair the development of the mind than to encourage it. For Joyce, the problem was not so much that the educational system did not teach, but that it taught children how *not* to think, learn, and question. The habits of mind inculcated by schools, Joyce believed, were anathema to the independence he valued so deeply; in schooling children to memorize received ideas, to regurgitate rote formulae and facts, and to accept passively what they are told rather than to quest actively for truth, teachers actively worked to beat (sometimes literally) individuality and intellect out of children. For this reason, Joyce associated formal education with violence and prejudice, noting repeatedly in his work that the worst traits of human nature—those things that education was supposed to temper or even eliminate—were actually enshrined within it. In *A Portrait of the Artist as a Young Man*, Joyce conveys these ideas by representing Jesuit teachers as ultimately more interested in terrorizing children than educating them; Stephen's alienation from the rigidly hierarchical narrowness of his teachers accounts in large part for his decision not to join their order and instead to strike out on his own, to "create," in Joyce's epiphanic phrasing, "the uncreated conscience of [his] race." In *Ulysses*, Joyce returns to the theme of education, deepening what was in *Portrait* a largely local critique of how the Jesuits failed one boy into a thoroughgoing condemnation of the very idea of formal education.

Taken as a whole, the "Nestor" chapter of *Ulysses* exemplifies—even crystallizes—Joyce's critique of formal education. The story of Stephen's failed day as a teacher is also the story of what is wrong with the project of formal schooling; indeed, the chapter may be read as a damning indictment of what happens to the minds of both teacher and student when they are compelled to perform their respective roles in the classroom. For the Joyce who writes *Ulysses*, the project of

the educational system is an impossible one in which rote repetition replaces the vitality of genuine intellectual discovery. In Stephen's classroom, students either demonstrate "comprehension" by responding correctly to the queries in a pedagogical catechism, or they admit ignorance ("The boy's blank face asked the blank window" [28]). Stephen, for his part, "conveys" information by drilling his students. Right and wrong substitute for give and take; rehearsal and recitation stand in for thought.

The one break in this stylized, sterilized procedure is Armstrong's joke—when asked if he knows anything about Pyrrhus, Armstrong responds, "Pyrrhus, sir? Pyrrhus, a pier" (29). The joke draws "mirthless high malicious laughter," a "silly glee" that at once appeals to Stephen's own propensity for word play (he goes on to describe a pier as a "disappointed bridge") and makes him fear that he is losing control of his class ("In a moment they will laugh more loudly, aware of my lack of rule and of the fees their papas pay" [29]). There can be no true play in the classroom; the power imbalance between teacher and student makes joint exploration impossible while the class difference between the impoverished teacher and the wealthy boys makes every deviation from routine on the part of a student an automatic mockery of the instructor.

The irony of "Nestor" is that Stephen is now expected to enforce the very things that he ran from. Joyce captures Stephen's painful awareness of this irony in the scene with Sargent, who has, at the behest of Mr. Deasy, haplessly copied and re-copied out the correct solutions to math problems in lieu of learning how to solve them. "Futility," Stephen thinks. Stephen both hates Sargent for being a passive rube ("Ugly and futile: lean neck and tangled hair and a stain of ink, a snail's bed. . . . a squashed boneless snail" [33]), and sees himself in Sargent ("like him was I, these sloping shoulders, this gracelessness. My childhood bends beside me" [34]). He shows Sargent how to solve the problem, but he cannot show Sargent how not to be the tentative, credulous creature he has been conditioned to be ("Waiting always for a word of help his hand moved faithfully the unsteady symbols, a faint hue of shame flickering behind his dull skin" [34]). Stephen's own classroom is a condensation of the parochial values that he finds so stifling in *Portrait*. Even in the moment of helping a student understand a concept that had previously mystified him, he cannot impart either a sense of independence or of pride.

The problem, Joyce seems finally to be saying, is the concept of the classroom itself. Within its confining space, neither teacher nor student occupies a role that is truly conducive to inquiry or learning. Whether a teacher is “good” or “bad” is beside the point; the failure is built into the system itself. Joyce gestures toward this fact when Mr. Deasy tells Stephen that “I foresee . . . that you will not remain here very long at this work. You were not born to be a teacher, I think. Perhaps I am wrong.” Stephen, who disagrees with Mr. Deasy about everything else, does concur in this. “A learner rather,” he replies. For Mr. Deasy, Stephen’s unorthodox ideas about God, nation, money, and history announce that he is not suited to his job. For Stephen, it is the desire to have ideas that makes him unsuited for his job. “Teaching” and “learning” are opposed in his world view; one cannot be a teacher and a learner, too.

Joyce was not alone in this opinion of formal education: Stephen’s belief that teaching and learning cannot take place in the same context was also the belief of Jean Piaget, the Swiss psychologist whose progressive theories of child development set the tone for twentieth-century educational theory. Piaget began to articulate his theories of how children learn during the period when Joyce was writing the *Wake*; in such works as *Judgment and Reasoning in the Child* (1924), *The Child’s Conception of the World* (1929), *The Child’s Conception of Physical Causality* (1932), and *Origins of Intelligence in the Child* (1936), Piaget expressed his conviction that children are “builders of their own intellectual structures” (Papert 7).¹⁰ For Piaget, as for Joyce, the classroom setting is antithetical to learning; formal education is an oxymoron, more a means of truncating the dynamic, intensely personal experience of discovery than of enhancing it. Devoting himself completely to the study of child psychology, Piaget concluded that children learn best when they own their own learning process. As Piaget’s student Seymour Papert explained it, “Piagetian learning” is the same as “learning without being taught” (7).

Papert notes that this aspect of Piaget’s work is at once his most radical and least acknowledged insight. The notion that children learn best in the absence of teachers simply has no place within educational theory; to the extent that twentieth-century educationists took up Piaget’s work, they

¹⁰For a Piagetian reading of *Ulysses*, see Joseph Bentley’s “The Stylistics of Regression in *Ulysses*.”

focused on his ideas about developmental phases, using them to devise curricula and pedagogy even as their attempts in this direction went against the grain of Piaget's anti-curricular thought. Joyce, however, had no such constraints, and in his depiction of Stephen Dedalus's sense of the "futility" of school he not only damns the system of formal education but initiates a thoroughgoing exploration of alternatives to that system. As such, the "Nestor" episode both tells us a great deal about how Joyce regarded education, and allows us to gain particular insight into how Joyce understood his own narrative experiments. If *Ulysses* articulates a devastating critique of formal schooling, finally suggesting that one who wishes to learn may neither teach nor be taught, *Finnegans Wake* develops a narrative style that is itself a concrete manifestation of an alternative approach to learning.

In "Night Lessons," the chapter of *Finnegans Wake* that is most devoted to the question of education, Shem, Shaun, and Issy do their lessons alone in a room above a pub. Joyce models their interaction on the pages of the chapter themselves, which are laid out spatially so as to indicate the dialogic quality of their engagement with their textbooks. As Joyce put it in a letter to Frank Budgen, the chapter is "a reproduction of a schoolgirl's old classbook complete with marginalia by the twins, who change sides at half time, footnotes by the girl (who doesn't), a Euclid diagram, funny drawings, etc." (*Letters* 405). The lessons themselves form a column of text running down the center of the page; these stream into one another in the signature manner of Wakean prose. The lessons-column is surrounded by the marginal and footnotes commentary of Shem, Shaun, and Issy. The overall effect is one of movement and dynamic instability. The text of the lessons is neither static nor set; the lessons themselves morph from subject to subject (an analysis of a geometric diagram is not simply a lesson in Euclid, but also in human sexuality, and, by extension, in competing theories of the origin of life). Likewise, each child engages with the lessons differently, deriving different truths, questions, and observations from them.

In typically Wakean terms, the story of the children doing their lessons unfolds into a "genetic epistemology" (the term is Piaget's), an investigation into the origins of knowledge

and into the specific question of how children think, learn, and know.¹¹ In “Night Lessons” there is no authoritarian teacher or adult moderating and instructing; the children interact freely, independent and unsupervised. As such, “Night Lessons” forms Joyce’s analytical and creative sequel to “Nestor.” This chapter is the *Wake*’s answer to the problems posed by its precursor in *Ulysses*; in it Joyce moves beyond the highly formalized, arid environment of Stephen’s classroom toward a vision of the ideal learning environment. That environment is tumultuous, spontaneous, dynamic, playful, skeptical, and often scatological; crucially, too, it is unsupervised, self-directed, and wholly experimental. Shaun captures the revisionary character of his learning process in the margins by noting the kind of progression it represents: “FROM CENOGENETIC DICHOTOMY THROUGH DIAGONISTIC CONCILIANCE TO DYNASTIC CONTINUITY” (275). Drawing an explicit contrast with “Nestor,” “Night Lessons” contrasts its “dynastic continuity” with the “cenogenetic dichotomy” of traditional conceptions of mastery. In “Nestor,” Mr. Deasy proudly gloats about his editorial on foot and mouth disease, crowing that he has “put the matter into a nutshell” (40). In “Night Lessons,” Shem glosses a passage with the phrase “*Omnitudes in a knutshedell*” (276). In the one, Mr. Deasy aims to boil complex issues down so that “there can be no two opinions on the matter” (40). In the other, Shem sees that even compressed concepts contain worlds. In these opposing views of understanding lies the kernel of Joyce’s critique of education; through an allusion to Deasy’s narrow-minded nutshell, “Night Lessons” expresses the sheer largeness of aptitude: every child the chapter seems to say, contains “omnitudes.” Small things—children, words—contain everything.

“Nestor” aimed to critique a particular model of education. “Night Lessons” responds to that critique, not by articulating a more desirable, alternative philosophy of learning but rather by attempting to mimic the process of learning itself. Samuel Beckett famously said that the *Wake*’s meaning lay in its structure: “Here form *is* content, content *is* form. You complain that this stuff is not written in English. It is not written at all. It is not to be read—or rather it is not only to be read. It is to be be looked at and listened to. His writing is not *about* something; *it is that something*

¹¹In this respect, Grace Eckley’s conclusion that “the children’s study session is primarily a lesson in sex” (198) seems reductive and overdetermined.

itself' (14). One could specify Beckett's claim even further: the content that is the form, the form that is the content, the something that the text of *Finnegans Wake* is is Joyce's conception of how the mind makes sense of the world. "Night Lessons," placed at the center of *Finnegans Wake*, is the focal point of the whole, the point of convergence that unifies the entire design. It is theory and practice; form and content; model and mimesis. Joyce's concept of learning without being taught lies at the heart of *Finnegans Wake*, embodied in the style that is ultimately its own subject matter.

To say that form is content and content is form is a literary way of saying that the concrete has been merged with the abstract. Individual words in the *Wake* mean more than themselves because they concretize Joyce's structural pattern; they are design elements as well as signifiers, libraries as well as referents. As such, they perform a type of synthesis that embodies Joyce's vision of how thought takes shape. The *Wake*'s words, phrases, puns, and sentences add up to an elaborate materialization of Joyce's theory of mental process. That theory in turn both derives from an understanding of learning that parallels Piaget's and moves beyond what Piaget himself thought was possible. Piaget believed that children were natural epistemologists, capable of, in Papert's words, being the "active builders of their own intellectual structures" by "appropriat[ing] to their own use materials they find about them" (19). But he also believed there were strict limits on how children of different ages could reason. By age six, Piaget believed, children are capable of "concrete" thinking but cannot make the move from the concrete to the abstract. "Formal," or abstract, thought does not become available until late childhood—around age twelve, children move from the concrete to the formal phase of development, which in turn signals their passage into adulthood. Joyce's understanding of children is neither so rigid nor so strict; in "Night Lessons," the children combine the concrete and the formal, reasoning both ways simultaneously to think on several levels at once.

The concrete epistemology of the *Wake*'s children is most clearly visible in their approach to the geometry problem: "Problem ye ferst, construct ann aquilittoral dryankle Probe loom!" or "Concoct an equoangular trillitter" (286.19–22). The children solve the problem in a roundabout way, collaboratively, through trial and error. Playing with their compasses ("a daintical pair of accomplasses!" [295.27]), they draw two overlapping circles of the same size (a "twain of

doubling bicirculars, mating approxemetely in their suite poi and poi, dunloop into eath the ocher” (295.31–33), connecting the centers of the “doubleviewed seeds” (296.1) to form the base of the triangle and drawing lines from each center to the point of intersection to form the sides. But the solution is also a new problem in its own right: the drawing that satisfies the requirements of the geometry lesson raises the question of what the female genitals look like. To the associative eyes of Shem, Shaun, and Issy, the equilateral triangle resembles the pubic triangle of a woman. A mathematical problem thus becomes the “mythametical” (286.23) problem of the body: the drawing enables the children to render the abstract problem of Euclid concrete; the concrete representation of the Euclidean figure in turn facilitates a different, more grounded—*earthier*—set of imaginative flights. As an abstract picture of the female genitals, the drawing is a formal impetus to concrete speculations about sex. In their most specific instantiations, these speculations center on the children’s mother, whose sexual parts now seem to be a secret whose truth her children must know: “Outer serpumstances beiuig ekewilled, we carefully, if she pleats, lift by her seam hem and jabote at the spidsiest of her trickkikant (like thousands done before since fillies calpered. Ocone! Ocone!) the maidsapron of our A.L.P., fearfully! till its nether nadir is vortically where (allow me aright to two cute winkles) its naval’s napex will have to beandbe” (279.7–14). The movement here is one of compression: a “dryankle” is also the “naval’s napex,” parallax—an angle—is also a pair of legs (“parilegs” [284.2]). Joyce’s name for the logic embodied in the lesson captures the playful precision, as well as the scatological geometry, of its incessant synthetic movement: “joyclid” (302.12).

Joyce’s literary portrayal of the ease with which the children move from formal to concrete and back again finds its computational counterpart in the work of Seymour Papert, who has devoted his career to the problem of making complex concepts—particularly mathematical ones—available to children. A devoted student of Piaget, Papert departs from Piaget’s rigid conception of developmental phases; his belief is not that children cannot reason formally before a certain age, but rather that their environment deprives them of the opportunity to develop the concrete attachments that facilitate abstract thought: “I see the classroom as an artificial and inefficient learning environment that society has been forced to invent because its informal environments fail

in certain essential learning domains, such as writing or grammar or school math” (8). For Papert, as for Piaget, children are their own best teachers. Unlike Piaget, Papert believes that when given the opportunity, children can readily work from the concrete to the abstract, combining them in much the way Joyce depicts in “Night Lessons.”

Like Joyce, Papert sees a combination of mathematics and writing as the medium that is most likely to enable complex combinatorial thought in the young. Specifically, he imagines that computers can help children bridge conceptual domains: “My conjecture is that the computer can concretize (and personalize) the formal. Seen in this light, it is not just another powerful educational tool. It is unique in providing us with the means for addressing what Piaget and many others see as the obstacle which is overcome in the passage from child to adult thinking. I believe that it can allow us to shift the boundary separating concrete and formal. Knowledge that was accessible only through formal processes can now be approached concretely” (21). Papert’s thesis is that working with computers helps children develop the two kinds of thinking that Piaget most firmly associates with the formal stage of intellectual development: “combinatorial thinking, where one has to reason in terms of the set of all possible states of a system, and self-referential thinking about thinking itself” (21). As such, Papert contends, the computer can help revolutionize our understanding of how children think, of what learning is, and ultimately of what human beings can accomplish: “The Piaget of the stage theory is essentially conservative, almost reactionary, in emphasizing what children cannot do. I strive to uncover a more revolutionary Piaget, one whose epistemological ideas might expand known bounds of the human mind” (157).

Papert’s thesis is that teaching children to program computers allows them to teach themselves science, mathematics, and language skills all at once: “I do not wish to reduce mathematics to literature or literature to mathematics. But I do want to argue that their respective ways of thinking are not as separate as is usually supposed” (39). He has tested—and proven—this thesis hundreds of times over during the past several decades, working with children as they teach themselves to teach computers to perform specific tasks (usually to make geometric drawings). Programming, Papert believes, is at once profoundly scientific and deeply humanistic, a form of mathematical reasoning that is also a type of communication: “Programming a computer means nothing more

or less than communicating to it in a language that it and the human user can both ‘understand.’ And learning languages is one of the things children do best. Every normal child learns to talk. Why then should a child not learn to ‘talk’ to a computer?” (5–6). It’s not just that children *can* learn to program, but that they are better suited to it, in ways, than adults because they are in their years of language acquisition. The linguistic facility of the child thus becomes, in Papert’s view, a mandate for bringing children “into a more humanistic as well as a more humane relationship with mathematics” (39); in so doing, Papert argues, programming can make children into the long-term beneficiaries of “a less dissociated cultural epistemology.” While programming, children must think on several levels at once; the computer’s drawings make concrete the abstract operations the children teach it to perform.

Papert does not refer to Joyce in his writing, and there is no evidence that Joyce was a significant influence on him. And yet his work with children and computers performs the imaginative work of “Night Lessons.” As “Night Lessons” is ultimately a meditation on how children think, so programming, for Papert, ultimately compels children to meditate on how thought works. Because programming is an activity that melds formal concepts and concrete tasks, children who program must necessarily become philosophers of cognition. When children have to think about how to teach the computer how to think, they are “embark[ing] on an exploration of how they themselves think. The experience can be heady: thinking about thinking turns the child into an epistemologist, an experience not even shared by most adults” (19). Such a vision of creatively engaged children contrasts sharply with Joyce’s portrait of disengaged, unthinking schoolchildren in “Nestor.” As one boy jerkily recites “Lycidas” from memory, glancing periodically at his book to prompt his memory and clearly giving no thought at all to either Milton’s meter or his sense, Stephen meditates on absorption, concentration, and inspiration. His thoughts travel to his time in Paris, and as he presides over the rocky recitation, he recalls the library of Saint Genevieve, where he had read each night, surrounded by others equally rapt before their books: “Fed and feeding brains about me: under glowlamps, impaled, with faintly beating feelers” (30). For Stephen, the fed and feeding brain of the larval studious adult is closely connected to the regurgitating brain of the dutiful but thoughtless boy; that the one is an image

of the other becomes clear when Stephen thinks contemptuously of Sargent as a “boneless snail.” The despair of “Nestor” is that of the brain intelligent enough to know that it has never properly developed, but not capable of seeing its way to a realistic or satisfying alternative; it is the depression born of knowing that the life of the mind is sterile and self-serving, while at the same time not knowing any other way to live: “Thought is the thought of thought” thinks Stephen (30–31). The joy of the *Wake*’s “joyclid” lessons, by contrast, is that of thought that is fully integrated into being, that can live and play and breathe as part of the body it inhabits.

Papert conceived of a similarly integrated kind of learning, one that would engage the emotions as much as the mind (vii) and that would consequently contain within it the potential for cultural revolution: “Computers can be carriers of powerful ideas and of the seeds of cultural change . . . they can help people form new relationships with knowledge that cut across the traditional lines separating humanities from sciences and knowledge of the self from both of these,” he writes; the transformative potential of technology is about “using computers to challenge current beliefs about who can understand what and at what age. It is about using computers to question standard assumptions in developmental psychology and in the psychology of aptitudes and attitudes. It is about whether personal computers and the cultures in which they are used will continue to be the creatures of ‘engineers’ alone or whether we can construct intellectual environments in which people who today think of themselves as ‘humanists’ will feel part of, not alienated from, the process of constructing computational cultures” (4–5). And so Papert pursues via computers the same sort of redemptive vision embodied in the *Wake*’s joyclid prose: that of a world where children are not only free to think, but where adults have not forgotten how to think like children. The central lesson of “Night Lessons,” after all, is that the entire *Wake* is a “night lesson”; what we encounter among school children is a style of thought that is identical to the style of the entire work, and what we learn from Shem, Shaun, and Issy’s studies is that in order to read the *Wake* we must learn—or recover the memory—of how to think like we did when we were children.

Papert’s idea is that the computer can facilitate in children the kind of profoundly affective connection with a “transitional object” that enables self-directed learning to occur: “The computer is the Proteus of machines. Its essence is its universality, its power to simulate. Because it can

take on a thousand forms and can serve a thousand functions, it can appeal to a thousand tastes” (viii). His own transitional object was the mechanical gear, which so fascinated him as a child that he taught himself many of the principles of physics simply by studying them. Papert’s 1980 book *Mindstorms* chronicles his decade-long attempt to develop an all-purpose transitional object to facilitate the learning of all kinds of children, “to turn computers into instruments flexible enough so that many children can each create for themselves something like what the gears were for me” (viii). One might say that what Papert does with the computer, Joyce attempted to do with language. The style of *Finnegans Wake* is designed to be the reader’s “transitional object,” at once a representation of an ideal melding of formal and concrete thought and a catalyst to help the reader teach herself how to engage the type of synthetic thought process Joyce’s prose represents. Joyce’s style is not only form and content, as Beckett noted, but a formal concretization of thought that merges the formal with the concrete. The “sophistication” of the *Wake* is thus actually something of a misnomer; the ultimate aim of Joyce’s style is simplicity itself. “People have lived with children for a long time,” Papert observes. “The fact that we had to wait for Piaget to tell us how children think and *what we all forget about our thinking as children* is so remarkable that it suggests a Freudian model of ‘cognitive repression’” (41). Joyce understood this. *Finnegans Wake* marks his effort to stage a return of the child’s repressed cognition.

In this chapter, I have argued that the mechanical metaphors Joyce used to describe his compositional process while writing *Finnegans Wake* morphed during the 1980s into computing metaphors. Taking seriously the notion that Joyce’s methodological techniques can be described in computing terms, I have argued both that the central principles of object-oriented programming can serve to ground a modern structuralist approach to Joyce’s text and that the structure of *Finnegans Wake* is itself that of an object-oriented program. Tracing the continuities between Joyce’s late stylistic experiments and late twentieth-century advancements in programming theory, I have shown how both scenes of innovation take the creative potential of the child as their inspiration, devising techniques for accessing that creativity that combine the engineering impulse with the communicative power of linguistic expertise.

It is not that Joyce anticipates object-oriented programming, or even that object-oriented

programming fulfills the vision of Joyce, but rather that both Joyce and object-oriented programmers solved the problem of complexity in the same way for the same reasons: both Joyce and the creators of object-oriented programs objected to reductive approaches to complex systems, preferring instead to honor complexity by organizing it; both devised simple means of harnessing the energy complexity generates; both, too, used those means to amplify complexity's power. These creative structural solutions to text-based communication systems were in turn progressive—even radical—critiques of traditional epistemologies and ontologies. Joyce saw that nineteenth-century realist narrative could not accommodate the growing complexity of the modern world; hence the mechanical, object-oriented architecture of *Finnegans Wake*. Programmers saw that linear, or procedural, programming could neither produce not sustain the growing size and complexity of modern programs; hence their creation of a modular programming method centered on objects.

The continued conceptual convergence of these very separate strands of history is registered in both the language programmers use to describe the transformative power of the object and in the role children have played in refining the theory and practice of object-oriented programming. Programmer and theorist Bruce Eckel describes the modern computer in much the way Joyce thought of the *Wake's* prose, as a means of both enhancing the operation of the mind and a mechanism for extending the mind beyond itself: “[C]omputers are not so much machines as they are mind amplification tools (‘bicycles for the mind,’ as Steve Jobs is fond of saying) and a different kind of expressive medium. As a result, the tools are beginning to look less like machines and more like parts of our minds, and also like other expressive mediums such as writing, painting, sculpture, animation, and filmmaking. Object-oriented programming is part of this movement toward using the computer as an expressive medium” (22). As Joyce moved narrative toward program via the sigla, so programmers use the object to increase the expressive and aesthetic dimensions of code. And as Joyce envisioned children as the prime movers of the *Wake*, at once main characters and conceptual modelers, concrete epistemologists and stylistic bricoleurs, so programmers created object-oriented technology with children in mind: inspired by Papert's work with children and computers, Alan Kay wrote Smalltalk, the programming language designed for

children that was also the first object-oriented programming language.

Kay helped program the innovative FLEX computer at the University of Utah while a doctoral candidate there in the late 1960s, and in 1968 he spent time with Seymour Papert at MIT's Artificial Intelligence Laboratory. When he started work at Xerox PARC two years later, he designed a small computer for children called the "KiddiKomp" and began work on a children's programming language called Smalltalk. Kay's effort was as much to revolutionize education as to challenge the dominant modes of thinking about programming languages. Kay himself learned to read at the age of three, devoured books, and as a five-year-old child already embodied the voracious intelligence, anti-authoritarian mindset, and quest for new perspectives that would characterize his career in computing research. He recalls that "By the time I got to school, I had already read a couple of hundred books. I knew in first grade that they were lying to me because I had already been exposed to other points of view. School is basically about one point of view—the one the teacher has or the textbooks have. They don't like the idea of having different points of view so it was a battle. Of course, I would pipe up with my five-year-old voice" (qtd. in Shasha and Lazere 39–40). Eager to equip children with tools that would allow them to overcome the limitations of his own formal education, Kay followed Papert in designing a programming method that would allow children to teach themselves. Kay's early experiments with the object-oriented Smalltalk were strikingly successful—using it, a 12-year-old designed a sophisticated drawing program, and a 15-year-old created a program that would design circuits (Shasha and Lazere 48).

Kay's way of thinking about education—that children teach themselves better than they can be taught; that children are capable of complex, synthetic thought normally classified as "adult"; that children's capacity for experimentation and their relative freedom from inflexible mindsets makes them ideal learners and strikingly creative thinkers—would not appear at all unfamiliar to Joyce, who thinks about education throughout his fiction. Nor should we be surprised at the unusual confluence of Kay's and Joyce's ideas about education—both owe their progressive philosophies of learning, as well as their own innovative approach to writing, to the radically experimental psychiatric climate of early twentieth-century Europe, which studied the formative process of children's thought and which, in the figure of Piaget, first forwarded the notion that children are

their own best teachers. In Smalltalk, the object functions as the concrete building block that enables children to piece together elaborate abstract structures. As such, it is both a practical tool and an epistemological entity, an engineering aid and a philosophical engine. Kay describes it in the following mechanico-philosophical terms: “Smalltalk’s design—and existence—is due to the insight that everything we can describe can be represented by the recursive composition of a single kind of behavioral building block that hides its combination of state and process inside itself and can be dealt with only through the exchange of messages. Philosophically, Smalltalk’s objects have much in common with the monads of Leibniz and the notions of 20th century physics and biology. Its way of making objects is quite Platonic in that some of them act as idealizations of concepts—*Ideas*—from which *manifestations* can be created. That the Ideas are themselves manifestations (of the Idea-Idea) and that the Idea-Idea is a-kind-of-Manifestation-Idea—which is a-kind-of itself, so that the system is completely self-describing—would have been appreciated by Plato as an extremely practical joke” (512–13). It does not seem farfetched to imagine Joyce appreciating the object’s practical jokery as well; his own objects work precisely that same way, and his masterwork is likewise as endlessly recursive as the most finely honed object-oriented program, its chapters, paragraphs, sentences, phrases, words, and even parts of words each containing within it the genetic code of the whole.

Conclusion: The Poetics of *Technē*

No study of the interrelationships between the aesthetic, political, and cultural forces shaping the relationship between literature and code would be complete without taking into account Martin Heidegger's enormously influential writings on technology. While philosophers and critics as diverse as Adorno, Marcuse, Foucault, Deleuze, Benjamin, and Habermas have all made important contributions to modern debates about the nature, meaning, and impact of modern technology, none have been as influential as Martin Heidegger, whose sustained and far-ranging critiques have definitively shaped the modern philosophical relationship between technology and aesthetics. In his later writings, notably in "The Question Concerning Technology," Heidegger develops a set of theses about technology that—difficult, ambiguous, and self-contradictory as they often are—have provided the foundation for debates about the relationship between aesthetics and technology since the 1950s. Furthermore, critics and philosophers have often invoked Heidegger's seemingly anti-technological bias and apparent Romantic nostalgia for a pre-technological age to authorize their impulse to divide the literary from the scientific and the technological from the aesthetic.

This dissertation concludes by suggesting that Heidegger's philosophical authority has been improperly invoked in support of these divisions. I will argue that Heidegger himself was far from espousing either a strongly reactionary, anti-technological stance or an understanding of technology as something essentially separate from, and hostile to, art. Indeed, commentators who enthrone Heidegger as an exemplary ancestral technophobe, or who use Heidegger's writings to justify their own exemplary technophobia, visit a reductive hermeneutic violence on Heidegger's writing that ultimately fails to account for the complexity of his technological speculations. I would like to argue instead that Heidegger's work may be read as an endorsement of responsible and creative technological practice, rather than as a backward-looking nostalgia for a lost pre-industrial past. I will suggest that the "art of code," which inextricably combines the technological and the aesthetic, exemplifies a pragmatic approach to technology that Heidegger strove philosophically to imagine.

Heidegger's late writings attempt to diagnose a technological malaise that he saw as symptomatic of the modern human condition. In his 1954 essay "The Question Concerning Technology," Heidegger claims that modernity opens a philosophical chasm between the technological and aesthetic modalities of "revealing," and argues that the overwhelming

twentieth-century dominance of technology represents both mankind's ultimate forgetting of Being and the historical end of metaphysics. The apocalyptic tones of Heidegger's essay is undoubtedly a sign of the times: if the machine guns and barbed wire of the First World War put an end to the nineteenth-century's largely unquestioning faith in technological advances, the warplanes, tanks, and atom bombs of the Second World War created inevitable associations between technological "progress" and inevitable Armageddon. With millions dead and much of Europe destroyed, Heidegger's pessimistic speculations about the ethical consequences of heedless technological innovation take on an undeniable poignance. Even so, "The Question Concerning Technology" should also be understood as part of a more sweeping historical, philosophical, and social critique, one that originates in Heidegger's earlier analyses of modern metaphysics in *Being and Time*.

In order to argue for the particularly *modern* nature of the aesthetic/technological split that concerns him, Heidegger turns in this essay to a strategically romanticized picture of Ancient Greece, whose putatively utopian manufacturing practices represent the ideal from which modern technological processes have deviated so dramatically. For Heidegger, the Ancient Greek craftsman is the direct ancestor, at least in terms of sensibility, of the peasant he observes working in the Rhine Valley: both are perfectly integrated into their environments; both labor unalienated from authentic earthly realities. However, while drawing this parallel, Heidegger was painfully aware that the analogy was rapidly dissolving into anachronism: the Rhine Valley peasants, vulnerable to the sociopolitical changes ushered in by the twentieth-century technological revolution, were also the last vestiges of earthly authenticity.¹² Terry Eagleton comments on the socioeconomic status of the German peasantry by the second decade of the twentieth century:

Heidegger's sturdily independent small farmer was an untypical, increasingly marginalized phenomenon: peasants with paltry smallholdings often had little opportunity to eke out their meagre livings through work for big landowners, and imported farm labour seriously jeopardized the position of the German farm

¹²German industrialization, and the consequent loss of rural, agricultural lifestyles, engendered intense political and philosophical debate, particularly during the 1920s and 1930s. Heidegger followed these discussions avidly; see discussions in Hans Sluga's *Heidegger's Crisis: Philosophy and Politics in Nazi Germany*, Michael Zimmerman's *Heidegger's Confrontation with Modernity: Technology, Politics, and Art*, and Jeffrey Herf's *Reactionary Modernism: Technology, Culture, and Politics in Weimar and the Third Reich*.

workers. They migrated, often enough, to the cities, to swell the ranks of the industrial proletariat. Conditions there were frequently dire: long hours, low wages, unemployment and poor housing were the price the German working class paid for the industrial capitalist boom. (308)

On the contrary, Heidegger can safely envision Ancient Greek craftsmen existing in a pristine, preindustrial state, untouched by industrialism and able to experience technological and aesthetic pursuits as intimately conjoined, even, at times, as identical.

Heidegger bolsters the crumbling analogy between Greek craftsman and modern peasant with etymology, noting that the word “technology” derives from the Greek word *technē*, and that “*technē* is the name not only for the activities and skills of the craftsman, but also for the arts of the mind and the fine arts: *Technē* belongs to bringing-forth, to *poiēsis*; it is something poietic” (13). Given the ontological significance that Heidegger so often accords the Greek language, it is hardly an exaggeration to suggest that he saw the history of the word *technē* as a metonymic history of metaphysics itself. In recovering a lost Greek etymology, Heidegger gestures toward a lost history. In highlighting the precarious condition of the modern-day peasant, Heidegger attempts to tell us why that history mattered, and what its loss will signify.

Combining sociological and etymological arguments, Heidegger demonstrates the interaction between technological and poetic modalities of revealing by asking his reader to consider the process by which a Greek silversmith would have manufactured a sacrificial chalice. As Heidegger imagines it, the silversmith would first have considered the relationship between the material at hand (silver) and the ideal (*eidos*) of “chaliceness.” Then he would have acknowledged his pivotal role as “that from whence the sacrificial vessel’s bringing forth and resting-in-self take and retain their first departure” (8). Presenting the act of manufacture as a moment of productive mediation between nature, mankind, and the gods, Heidegger imagines that the Greek craftsman must have conceptualized his work as an act of “bringing-forth” in which *technē* and *poiēsis* necessarily existed as one. Contending that such an aesthetic relationship to technology was foundational to Ancient Greek thought and culture, Heidegger argues that the Greeks ultimately understood technology as a mode of revealing: “technology comes to presence in the realm where revealing and unconcealment take place, where *alētheia*, truth, happens” (13). As such, Heidegger envisions

Ancient Greek craftsmanship as a passive activity; the job of the craftsman is not to impose his will on the material but to “unconceal” aesthetic artifacts, to let them become what they most clearly are: “It is as revealing, and not as manufacturing, that *technē* is a bringing-forth” (13). In Heidegger’s formulation, the craftsman is both crucially present and curiously absent in the moment of manufacture: although he is the active agent by which the silver (or wood, or stone) takes shape, he is also merely a conduit through which the chalice (or ship, or house) reveals itself.

The contrast with mid-twentieth century Germany is profound. Although technology in our modern era still functions as a mode of revealing, the *nature* of that revealing has changed drastically: “The revealing that rules in modern technology is a challenging,¹³ which puts to nature the unreasonable demand that it supply energy that can be stored as such” (14). Rather than “unconcealing” the natural world, modern technology violently converts it into what Heidegger calls “standing-reserve,” or resources to be stored and used up: “The coal that has been hauled out in some mining district has not been supplied in order that it may simply be present somewhere or other. It is stockpiled; that is, it is on call, ready to deliver the sun’s warmth that is stored in it. The sun’s warmth is challenged forth for heat, which in turn is ordered to deliver steam whose pressure turns the wheels that keep a factory running” (15). Heidegger gives the term *Gestell* (Enframing) to the animating principle behind the desire to transform everything into an easily manipulable form. Rather than living in harmony with nature as the Ancient Greeks did, and letting natural materials “reveal themselves” as useful or beautiful objects through acts of craftsmanship, humans now attempt to impose their will upon the natural world and “reveal” everything as standing-reserve. For Heidegger, the culmination of Enframing is that “[man] himself will have to be taken as standing-reserve.” At its most extreme, then, Heidegger’s vision of modernity evokes something very like the technological nightmare of *The Matrix*, where humans have literally been converted into a source of energy, a giant battery that powers the very force that dominates them.

For Heidegger, modernity is characterized by an ontological tension between the *technē* and *poiēsis* he saw as harmoniously conjoined in the Greek worldview. No longer working

¹³Heidegger’s word is *Herausfordern*, which the translator notes has various possible interpretations in English, including *to challenge*, *to call forth*, *to summon to action*, *to demand*, or *to provoke*.

in harmony with *poiēsis* as part of a respectful *alētheuein*, *technē* has become the agent of domination and power. In fact, the very modality of revealing embodied by *poiēsis* is threatened by an all-consuming technological Enframing: “As a destining, [Enframing] banishes man into that kind of revealing which is an ordering. Where this ordering holds sway, it drives out every other possibility of revealing. Above all, Enframing conceals that revealing which, in the sense of *poiēsis*, lets what presences come forth into appearance” (27). As far as Heidegger is concerned, modern humankind has lost the Greek silversmith’s attentiveness to *poiēsis*. Instead, he warns, we have avidly embraced the *technē* that threatens to engulf us. It is here that we can see the nature of the paradox that drives Heidegger’s technological critique. On one hand, Heidegger associates *poiēsis* with passivity, letting-be, and giving; while Enframing signals domination, exploitation, and the will-to-power. On the other hand, *poiēsis* is Heidegger’s solution to the dangers of Enframing. Heidegger freely admits that the openness to *poiēsis* exemplified by the Ancient Greek silversmith has all but disappeared in our modern era. But he also believes that a sufficiently aware and attentive philosophy can allow *poiēsis* to presence-forth from within *technē* and temper its dangerous excesses. Unimpressed by what looks like weak reasoning and weaker wishful thinking, critics—many of them Marxists critics—have charged Heidegger with advocating little more than a retreat into passive aestheticism. Moreover, they have expressed concern whether—and how—such seemingly irresponsible philosophy may have been complicit with Heidegger’s endorsement of the Nazi movement during the 1930s. In *The Philosophical Discourse of Modernity*, for example, Jürgen Habermas proposes that the philosophical positions adduced in *Being and Time* foreshadow Heidegger’s involvement with National Socialism in the 1930s; in *Heidegger’s Confrontation with Modernity*, Michael Zimmerman wonders aloud “whether Heidegger’s thought is intrinsically fascist” (37); and in *La fiction du politique*, Philippe Lacoue-Labarthe declares that Heidegger’s support of Nazism was “absolutely coherent with his thought” (38). Terry Eagleton writes that Heidegger’s aesthetic “leaves the human subject rapt in paralytic reverence before a numinous presence” (313), and suggests that this “numinous presence” could as easily be Adolf Hitler as anything else: “Heidegger swings with a minimum of mediation between the nebulously ontological and the sinisterly specific” (310). This reading of *poiēsis* as passive, and the insistent

condemnation of that passivity as fascist, has meant that Heidegger has essentially been excluded from the bulk of late twentieth century thinking about art's relation to technology. But the eagerness to discount and discredit Heidegger has rested on a series of misreadings of his ideas.

Heideggerian *poiēsis* is not limited to what we conventionally recognize as the aesthetic realm. It does not signal a retreat from the world or from technology, nor does it entail either political or aesthetic passivity. Rather it is part of Heidegger's *critique* of a passive aestheticism that sees art as a privileged realm of avoidance and escape. In fact, Heidegger proposes that our contemporary notion of "the aesthetic" is itself a product of the very split between *poiēsis* and *technē* that his philosophy tries to reconcile. He even argues that aesthetic objects do not necessarily embody *poiēsis* in any significant way. Commenting on the place occupied by the aesthetic in contemporary life, Heidegger writes "When there is still room left in today's dwelling for the poetic, and time is still set aside, what comes to pass is at best a preoccupation with aestheticizing, whether in writing or on the air. Poetry is either rejected as a frivolous mooning or vaporizing into the unknown, and a flight into dreamland, or is counted as a part of literature. . . . In such a setting, poetry cannot appear otherwise than literature" (*Poetry, Language, Thought* 214). Emphasizing this point through a further contrast with Ancient Greece, he writes, "Once there was a time when the bringing-forth of the true into the beautiful was called *technē*. And the *poiēsis* of the fine arts also was called *technē*. . . The arts were not derived from the artistic. Art works were not enjoyed aesthetically. Art was not a sector of cultural activity" ("Question Concerning Technology," 34). Our modern designation of specifically "aesthetic" genres and zones would seem—at least for Heidegger—to be a form of false consciousness, to offer a deceptively simplistic reassurance about the vitality of *poiēsis*.

So it is that in "The Origin of the Work of Art," Heidegger distinguishes between *poesie*, or poetry in the narrow, literary sense, and *dichtung*, a broader concept that would allow *poiēsis* to expand beyond the narrow categorical and institutional confines of literature. Heidegger makes this distinction not in order to project a poetic "mentality" onto extra-poetic objects and thus to aestheticize away sociopolitical realities (as his Marxist detractors would have it); but rather to re-create the possibility for a presencing-forth of *poiēsis*, to make that presencing-forth a vital

element in modern existential experience, and to allow for a potential reunion of *poiēsis* and *technē* that would also re-establish an authentic relation to Being. Heidegger thus suggests that our modern malaise can only be resolved through *philosophical* resistance to the pervasive dominance of Enframing and to the forgetting of Being.

I have devoted considerable space to explicating Heidegger's position on technology because it is my contention that computer programmers, who refuse to recognize conventional disciplinary boundaries between poetry and technology, demonstrate a curiously Heideggerian appreciation for the "presencing forth" of *poiēsis* within *technē*. As I have demonstrated above, a careful reading of Heidegger shows that his writings emphatically resist the institutionalization and compartmentalization of the poetic, that indeed they point frequently to the fact that *poiēsis* and *technē* share common etymological and historical origins. As I have shown throughout this dissertation, the aesthetic culture of computer programming poses a similarly compelling challenge to narrow institutional definitions of the "literary."

That the literary culture of programming has gone almost entirely unnoticed by literary theorists should not, then, surprise us. During the half-century of that culture's genesis, Heideggerian critics typically espoused stances that first embedded *poiēsis* within a predefined set of institutionalized aesthetic artifacts, and then protected those aesthetic artifacts from alien technological intrusion. For instance, in *Poetry at Stake: Lyric Aesthetics and the Challenge of Technology*, Carrie Noland shows how twentieth-century poetry criticism first enshrined an antagonistic relationship between the lyric poem and technology, and then consistently deprecated technology for mechanizing and mediating the supposedly self-present lyric voice. Noland argues that poetry critics' entrenched hostility to technology draws much of its philosophical reinforcement from a recognizably Heideggerian ontology of poetic language (4). In *Heidegger's Confrontation with Modernity*, Michael Zimmerman succinctly summarizes this ontology when he argues that for Heidegger "the gift of language poses at once the greatest gift and the greatest danger for humanity, for the loss of a proper relation to language leads to spiritual and social catastrophe" (114). As we have seen, Heidegger often uses language as a kind of index to

philosophical well-being. He also undeniably valorizes poets—notably Rimbaud—who hold more “authentic” relationships to language. And yet Heidegger also argues against an inflexible division of language into “poetic” and “unpoetic” discourses.

Although Heidegger’s influence on modern concepts of both poetry and technology has been profound and far-reaching, his philosophy has generally enabled a separatist standoff between the two realms. Heideggerian poetic theorists have inevitably argued for poetic purity; Heideggerian technologists warn that technology will drain us of our humanity and creativity. This dissertation has resisted such assumptions about both the nature of the poetic and the nature of the technological, and yet it has tried to do so not by following deconstructive and post-deconstructive theoretical debates, but by following what I understand as the true spirit of Heideggerian thought. As Heidegger understood *poiēsis* and *technē* to be inseparable aspects of a larger creative whole, so I have understood computing history as a deep intertwining of the poetic and technological.

More broadly, this dissertation has aimed to show that aesthetics, understood in the traditional sense, is inherently political, in other words, how traditional, supposedly debunked, values—values having to do with sincere faith in artistic effort and a belief in beauty—may emerge in putatively “non-artistic” spheres as powerful political forces. The political struggle in which Richard Stallman, Eric Raymond, and others are engaged is fundamentally inseparable from an idea of code as an aesthetic object: there is a one-to-one correlation between the beauty of code and its political efficacy. My purpose in doing this study has thus been to illuminate and analyze an aspect of hacker culture that has not previously been studied: its insistence on the interpenetration of *technē* and *poiēsis*. Hacker culture has been explored historically by Stephen Levy, anthropologically by Eric Raymond, philosophically by Pekka Himanen, but it has never been studied aesthetically, as an artistic culture in and of itself. As the first attempt to develop a sustained, serious analysis of computing culture as a literary culture, *The Art of Code* seeks to lay the foundation for an aesthetics of technology that would not accept the Heideggerian split of *technē* from *poiēsis*.

A related goal of this dissertation has been to break through some of the categorical and conceptual gridlock that presently defines the humanites’ relation to beauty, to literary value, to politics, and, of course, to all non-literary “texts” such as computers. Ideals espoused in

programming cultures—of innovative understandings of authorship and creative approaches to beauty—are being lost in the field that ostensibly has a greater claim to them. The rise of politicized literary theory during the 1970s and the emergence of cultural studies in the 1980s coincides with the emergence of the new critical approach to code: as literary studies ceased to center on the close reading of discrete literary texts, computer science began to center increasingly on close reading code with an eye to appreciating the structural links between form and content. By contrast, hackers concentrate on the form and content of code. To the extent that hackers interest themselves in the “politics” of code (to the extent that they have found themselves to be, in Raymond’s word, “accidental revolutionaries”), they focus on making what has become known as “rebel code”: better, more robust, more efficient, more reliable code than the proprietary software with which it competes. In short, their way of politicizing their code is to strive for its aesthetic perfection. So it is that while literary criticism moves ever closer toward a dismantling of the aesthetic (as a ruse that obscures the material “reality” within and beneath a text), hackers realize that to make code useful politically they must never cease to make it more beautiful.

Works Cited

- Akera, Atsushi. "The Early Computers." Akera and Nebeker 63–75.
- Akera, Atsushi and Frederik Nebeker, eds. *From 0 to 1: An Authoritative History of Modern Computing*. Oxford: Oxford University Press, 2002.
- Armer, Paul. "SHARE—A Eulogy to Cooperative Effort." *Annals of the History of Computing* 2 (April 1980): 122–29.
- Arnaud, Noël. *Poèmes algol*. Verviers: Temps Mêlés, 1968.
- Aspray, William. "John von Neumann's Contributions to Computing and Computer Science." *Annals of the History of Computing* 11 (1989): 189–95.
- Aspray, William and Arthur Burks, eds. *Papers of John von Neumann on Computing and Computer Theory*. Cambridge, MA: MIT Press, 1987.
- Augarten, Stan. *Bit by Bit—An Illustrated History of Computers*. New York: Ticknor & Fields, 1984.
- Baase, Sara. *A Gift of Fire: Social, Legal, and Ethical Issues in Computing*. Upper Saddle River, NJ: Prentice Hall, 1997.
- Backus, John. "Programming in America in the 1950s—Some Personal Impressions." *Metropolis, Howlett, and Rota* 125–35.
- Backus, John. "The History of FORTRAN I, II, and III." *Annals of the History of Computing* 1 (1979): 21–37.
- Barnes, Susan B. "Computer Interfaces." Akera and Nebeker 133–47.

- - -. *The Development of Graphical User Interfaces from 1970 to 1993*. Ph.D. thesis, New York University, New York, 1995.
- Baudrillard, Jean. *The System of Objects*. New York: Verso, 1996.
- Baum, Joan. *The Calculating Passion of Ada Byron*. Hamden, CT: Archon Books, 1986.
- Beckett, Samuel. "Dante... Bruno. Vico.. Joyce." *An Exagmination of James Joyce*. New York: Haskell House, 1974. 3–22.
- Bell, David and Barbara Kennedy, eds. *The Cybercultures Reader*. New York: Routledge, 2000.
- Bennington, Geoffrey. "Aberrations: De Man (and) the Machine." *Reading de Man Reading*. Ed. Lindsay Waters and Wlad Godzich. Minneapolis: University of Minnesota Press, 1989. 209-22.
- Benstock, Shari. "The Letter of the Law: *La Carte Postale* in *Finnegans Wake*." *Philological Quarterly* 63 (Spring 1984): 204–24.
- Bentley, Jon. *Programming Pearls*. Reading, MA: Addison-Wesley, 2000.
- Bentley, Joseph. "The Stylistics of Regression in Ulysses." *James Joyce and His Contemporaries*. Ed. Diana Ben-Merre and Maureen Murphy. Westport, CT: Greenwood Press, 1989. 31–35.
- Berard, Edward. *Essays on Object-Oriented Software Engineering*. Englewood Cliffs, NJ: Prentice Hall, 1992.
- Bergin, Thomas J. and Richard G. Gibson, eds. *History of Programming Languages–II*. Reading, MA: Addison-Wesley, 1996.
- Berners-Lee, Tim. *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by Its Inventor*. San Francisco: HarperSanFrancisco, 1999.
- Birkerts, Sven. *The Gutenberg Elegies: The Fate of Reading in an Electronic Age*. London: Faber and Faber, 1994.
- Birkerts, Sven, ed. *Tolstoy's Dictaphone: Technology and the Muse*. Saint Paul, MN: Graywolf Press, 1996.

- Bolter, Jay David. *Writing Space: The Computer, Hypertext, and the History of Writing*. Hillsdale, NJ: Lawrence Erlbaum Associates, 1991.
- Borgmann, Albert. *Crossing the Postmodern Divide*. Chicago: University of Chicago Press, 1992.
- . *Holding on to Reality: The Nature of Information at the Turn of the Millennium*. Chicago: University of Chicago Press, 1999.
- Bové, Paul A. *Destructive Poetics: Heidegger and Modern American Poetry*. New York: Columbia University Press, 1980.
- Brookshear, Glenn J. *Computer Science: An Overview*. New York: Basic Books, 1997.
- Budd, Timothy. *An Introduction to Object-Oriented Programming*. Second edition. Reading, MA: Addison-Wesley, 1997.
- Budgen, Frank. "James Joyce." *James Joyce: Two Decades of Criticism*. Ed. Seon Givens. 18–24.
- Burks, Arthur W. and Alice R. Burks. "The ENIAC: First General-Purpose Electronic Computer." *Annals of the History of Computing* 3 (October 1981): 310–99.
- Burrell, Harry. *Narrative Design in Finnegans Wake: The Wake Lock Picked*. Gainesville, FL: University Press of Florida, 1996.
- Calcutt, Andrew. *White Noise: An A–Z of the Contradictions in Cyberspace*. New York: St. Martin's Press, 1999.
- Campbell-Kelly, Martin. "Programming the EDSAC: Early Programming Activity at the University of Cambridge." *Annals of the History of Computing* 2 (1979): 7–36.
- Campbell-Kelly, Martin and William Aspray. *Computer: A History of the Information Machine*. New York: Basic Books, 1996.
- Ceruzzi, Paul. *A History of Modern Computing*. Cambridge, MA: MIT Press, 1998.
- Chase, George C. "History of Mechanical Computing Machinery." *Annals of the History of Computing* 2 (1980): 198–226.

- Childs, Bart and Johannes Sametinger. "Analysis of Literate Programs from the Point of View of Reuse." *Software—Concepts and Tools* 18 (1996): 35–46.
- Cifuentes, C. *Reverse Compilation Techniques*. Ph.D. thesis, Queensland University of Technology, Queensland, 1994.
- Cohen, I. Bernard. "Howard Aiken and the Dawn of the Computer Age." *The First Computers—History and Architectures*. Ed. Raúl Rojas and Ulf Hashagen. 107–20.
- . "Howard Aiken on the Number of Computers Needed for the Nation." *Annals of the History of Computing* 20 (1998): 27–32.
- . *Howard Aiken: Portrait of a Computer Pioneer*. Cambridge, MA: MIT Press, 1999.
- Cohen, Julie E. and Mark A. Lemley. "Patent Scope and Innovation in the Software Industry." *California Law Review* 89 (2001): 1–57.
- Colburn, Timothy R. *Philosophy and Computer Science*. London: M. E. Sharpe, 2000.
- Comer, Douglas E. *Internetworking with TCP/IP*. 4th edition. Upper Saddle River, NJ: Prentice-Hall, 2000.
- Conley, Verena Andermatt, ed. *Rethinking Technologies*. Minneapolis, MN: University of Minnesota Press, 1993.
- Corcoran, Elizabeth. "Soft Lego." *Scientific American* (January 1993): 145–46.
- Courtois, P. "On Time and Space Decomposition of Complex Structures." *Communications of the ACM* 28 (1985).
- Cox, Brad. "There Is a Silver Bullet." *Byte* (November 1990): 209–18.
- Dahl, Ole-Johan, Edsger W. Dijkstra, and C. A. R. Hoare. *Structured Programming*. London: Academic Press, 1972.
- Davidson, Michael. *Ghostlier Demarcations: Modern Poetry and the Material Word*. Berkeley: University of California Press, 1997.
- Davis, Alan M. *201 Principles of Software Development*. New York: McGraw Hill, 1995.

- Davis, Martin. *The Universal Computer: The Road from Leibniz to Turing*. New York: Norton, 2000.
- de Kerckhove, Derrick. *Connected Intelligence: The Arrival of the Web Society*. Toronto: Somerville House, 1997.
- de Man, Paul. *Allegories of Reading: Figural Language in Rousseau, Nietzsche, Rilke, and Proust*. New Haven: Yale University Press, 1979.
- DeGrandpre, Richard. *Digitopia: The Look of the New Digital You*. New York: AtRandom.Com Books, 2001.
- Denning, Peter J., ed. *Talking Back to the Machine: Computers and Human Aspiration*. New York: Copernicus, 1999.
- Derrida, Jacques. *Of Grammatology*. Trans. Gayatri Spivak. Baltimore: Johns Hopkins University Press, 1974.
- Derrida, Jacques. "Two Words for Joyce." *Post-structuralist Joyce: Essays from the French*. Ed. Derek Attridge and Daniel Ferrer. Cambridge: Cambridge University Press, 1984. 31–35.
- DiBona, Chris, Sam Ockman, and Mark Stone, eds. *Open Sources: Voices from the Open Source Revolution*. Sebastopol, CA: O'Reilly, 1999.
- Dijkstra, Edsger. "Go To Statement Considered Harmful." *Communications of the ACM* 11 (1968): 147–48.
- Dreyfus, Hubert L. *What Computers Still Can't Do: A Critique of Artificial Reason*. Cambridge, MA: MIT Press, 1992.
- Dreyfus, Hubert L. and Stuart E. Dreyfus. *Mind Over Machine: The Power of Human Intuition and Expertise in the Era of the Computer*. New York: The Free Press, 1996.
- Eagleton, Terry. *The Ideology of the Aesthetic*. Oxford: Blackwell, 1990.
- Eckel, Bruce. *Thinking in C++*. Second edition. Upper Saddle River, NJ: Prentice Hall, 2000.
- Eckley, Grace. *Children's Lore in Finnegans Wake*. Syracuse, NY: Syracuse University Press, 1985.

- Ensmenger, Nathan L. *From "Black Art" to Industrial Discipline: The Software Crisis and the Management of Programmers*. Ph.D. thesis, University of Pennsylvania, Philadelphia, 2001.
- Ermann, David M., Mary B. Williams, and Glaudio Gutierrez, eds. *Computers, Ethics, and Society*. New York: Oxford University Press, 1990.
- Evans, Bob O. "System/360: A Retrospective View." *Annals of the History of Computing* 8 (1986): 155–79.
- Feenberg, Andrew. *Questioning Technology*. New York: Routledge, 1999.
- Ferrer, Daniel. "The Freudful Couchmare of \wedge d: Joyce's Notes on Freud and the Composition of Chapter XVI of *Finnegans Wake*." *James Joyce Quarterly* 22 (Summer 1985): 367–82.
- Frederick P. Brooks, Jr. *The Mythical Man-Month*. New York: Basic Books, 1995.
- Freeman, Eva. "Building Gargantuan Software." *Scientific American* (September 1999): 28–31.
- Fritz, W. Barkley. "The Women of ENIAC." *Annals of the History of Computing* 18 (1996): 13–23.
- Gabriel, Richard. *Patterns of Software: Tales from the Software Community*. Oxford: Oxford University Press, 1996.
- Gancarz, Mike. *The UNIX Philosophy*. Boston: Digital Press, 1995.
- Garfinkel, Simson L., Richard M. Stallman, and Mitchell Kapor. "Why Patents Are Bad for Software." *Ludlow* 35–45.
- Gelernter, David. *Mirror Worlds, or the Day Software Puts the Universe in a Shoebox . . . How It Will Happen and What It Will Mean*. Oxford: Oxford University Press, 1991.
- Gelernter, David. *Machine Beauty: Elegance and the Heart of Technology*. New York: Basic Books, 1998.
- Gibbs, W. Wyatt. "Software's Chronic Crisis." *Scientific American* (September 1994): 86–95.
- Gillespie, Michael Patrick. *Joyce through the Ages: A Nonlinear View*. Gainesville, FL: University Press of Florida, 1999.

- Givens, Seon, ed. *James Joyce: Two Decades of Criticism*. New York: Vanguard Press, 1963.
- Glass, Robert. *In the Beginning: Recollections of Software Pioneers*. Los Alamitos, CA: IEEE Computer Society, 1998.
- Glass, Robert. *Software Runaways*. Upper Saddle River, NJ: Prentice-Hall, 1998.
- Goldsmith, Kenneth. "Soliloquy." URL:
<http://epc.buffalo.edu/authors/goldsmith/soliloquy/> (2001).
- Goldstine, Herman H. *The Computer from Pascal to von Neumann*. Princeton: Princeton University Press, 1993.
- Greenbaum, Joan. *Windows on the Workplace: Computers, Jobs, and the Organization of Office Work in the Late Twentieth Century*. New York: Monthly Review Press, 1995.
- Habermas, Jürgen. *The Philosophical Discourse of Modernity*. Cambridge, MA: MIT Press, 1987.
- Hansen, Mark. *Embodying Technesis: Technology Beyond Writing*. Ann Arbor: University of Michigan Press, 2000.
- Harries, Karsten. "Philosophy, Politics, Technology." *Martin Heidegger: Politics, Art, and Technology*. Ed. Karsten Harries and Cristoph Jamme. New York: Holmes & Meier, 1994. 225–45.
- Hart, Clive. *Structure and Motif in Finnegans Wake*. Evanston, IL: Northwestern University Press, 1962.
- Heckel, Paul. "Debunking the Software Patent Myths." *Ludlow* 63–107.
- Heidegger, Martin. *Poetry, Language, Thought*. New York: Harper & Row, 1971.
- . *The Question Concerning Technology and Other Essays*. New York: Harper & Row, 1977.
- Heim, Michael. *Electric Language: A Philosophical Study of Word Processing*. New Haven: Yale University Press, 1987.
- . *The Metaphysics of Virtual Reality*. Oxford: Oxford University Press, 1993.

- Herf, Jeffrey. *Reactionary Modernism: Technology, Culture, and Politics in Weimar and the Third Reich*. Cambridge: Cambridge University Press, 1984.
- Himanen, Pekka. *The Hacker Ethic and the Spirit of the Information Age*. New York: Random House, 2001.
- Hopkins, Sharon. "Camels and Needles: Computer Poetry Meets the Perl Programming Language." URL: <http://shroop.net/~sharon/perlpoetrypaper.ps> (1992).
- Hopper, Grace Murray. "The Education of a Computer." *Annals of the History of Computing* 9 (1988): 271–81.
- Horowitz, Ellis, ed. *Programming Languages: A Grand Tour*. Rockville, MD: Computer Science Press, 1987.
- Ichbiah, Daniel and Susan Knepper. *The Making of Microsoft*. Rocklin, CA: Prima, 1991.
- Johnson, Deborah G. *Computer Ethics*. Upper Saddle River, NJ: Prentice Hall, 2001.
- Johnson, Lucanne. "A View from the Sixties: How the Software Industry Began." Akera and Nebeker 101–109.
- Jolas, Eugene. "My Friend James Joyce." *Givens* 3–18.
- Jones, Capers. "Sizing Up Software." *Scientific American* (December 1998): 104–109.
- Kay, Alan. "The Early History of Smalltalk." *History of Programming Languages—II*. Ed. Thomas J. Bergin and Richard G. Gibson. 511–78.
- Kestner, Joseph. "Virtual Text/Virtual Reader: The Structural Signature Within, Behind, Beyond, and Above" *James Joyce Quarterly* 16 (Winter 1978): 367–82.
- Kidwell, Peggy and Paul Ceruzzi. *Landmarks in Digital Computing*. Washington, D.C.: Smithsonian Institute, 1994.
- Kittler, Friedrich. *Discourse Networks, 1800/1900*. Stanford: Stanford University Press, 1990.
- . "There Is No Software." *Stanford Literature Review* 9 (Spring 1992): 81–90.
- Knuth, Donald. *METAFONT: The Program*. Reading, MA: Addison-Wesley, 1986.

- - -. *T_EX: The Program*. Reading, MA: Addison-Wesley, 1986.
 - - -. *Literate Programming*. Stanford: Center for the Study of Language and Information, 1992.
 - - -. *The Art of Computer Programming Volume 1: Fundamental Algorithms*. Reading, MA: Addison-Wesley, 1997.
 - - -. *The Art of Computer Programming Volume 2: Seminumerical Algorithms*. Reading, MA: Addison-Wesley, 1998.
 - - -. *The Art of Computer Programming Volume 3: Sorting and Searching*. Reading, MA: Addison-Wesley, 1998.
 - - -. *Digital Typography*. Stanford, CA: CSLI Publications, 1999.
 - - -. *Things A Computer Scientist Rarely Talks About*. Stanford: CSLI Publications, 2001.
- Knuth, Donald E. and Luis Trabb Pardo. "The Early Development of Programming Languages." *Encyclopedia of Computer Science and Technology*. Ed. J. Belzer, A. G. Holzman, and A. Kent. Volume 6 . New York: Dekker, 1977. 419–93.
- Koenig, Andrew and Barbara Moo. *Ruminations on C++*. Reading, MA: Addison-Wesley, 1997.
- Koepsell, David R. *The Ontology of Cyberspace: Philosophy, Law, and the Future of Intellectual Property*. Chicago: Open Court, 2000.
- Kohanski, Daniel. *Moths in the Machine: The Power and Perils of Computer Programming*. New York: St. Martin's Griffin, 1998.
- - -. *The Philosophical Programmer: Reflections on the Moth in the Machine*. New York: St. Martin's Press, 1998.
- Lacoue-Labarthe, Philippe. *La fiction du politique*. Paris: Christian Bourgois, 1987.
- Landow, George. *Hypertext: The Convergence of Contemporary Critical Theory and Technology*. Baltimore, MD: Johns Hopkins University Press, 1992.
- Langley Moore, Doris. *Ada, Countess of Lovelace: Byron's Legitimate Daughter*. New York: Harper and Row, 1977.

- Lanham, Richard. *The Electronic Word: Democracy, Technology, and the Arts*. Chicago: University of Chicago Press, 1993.
- Laplante, Phillip, ed. *Great Papers in Computer Science*. St. Paul, MN: West Publishing Company, 1996.
- Lee, J. A. N. "Pioneers in Computing." Akera and Nebeker 76–87.
- Lessig, Lawrence. *Code and Other Laws of Cyberspace*. New York: Basic Books, 1999.
- . *The Future of Ideas: The Fate of the Commons in a Connected World*. New York: Random House, 2001.
- Levinson, Paul. *The Soft Edge: A Natural History and Future of the Information Revolution*. New York: Routledge, 1997.
- Levy, Steven. *Hackers: Heroes of the Computer Revolution*. New York: Anchor Press/Doubleday, 1984.
- . *Insanely Great: The Life and Times of Macintosh, the Computer that Changed Everything*. New York: Viking, 1994.
- Lions, John. *Lions' Commentary on UNIX 6th Edition with Source Code*. San Jose, CA: Peer-to-Peer Communications, 1996. n. pag.
- Ludlow, Peter, ed. *High Noon on the Electronic Frontier: Conceptual Issues in Cyberspace*. Cambridge, MA: MIT Press, 1996.
- Mahaffey, Vicki. *Reauthorizing Joyce*. Gainesville, FL: University Press of Florida, 1995.
- . *States of Desire: Wilde, Yeats, Joyce, and the Irish Experiment*. Oxford: Oxford University Press, 1998.
- Mahoney, Michael. "The History of Computing in the History of Technology." *Annals of the History of Computing* 10 (1988): 113–125.
- . "Software: The Self-Programming Machine." Akera and Nebeker 91–100.
- Manes, Stephen and Paul Andrews. *Gates: How Microsoft's Mogul Reinvented an Industry—and Made Himself the Richest Man in America*. New York: Touchstone, 1994.

- Mathews, Harry and Alastair Brotchie, eds. *Oulipo Compendium*. London: Atlas Press, 1998.
- McCartney, Scott. *ENIAC: The Triumphs and Tragedies of the World's First Computer*. New York: Walker and Company, 1999.
- McHugh, Roland. *The Sigla of Finnegans Wake*. Austin, TX: University of Texas Press, 1976.
- - -. *Annotations to Finnegans Wake*. Baltimore, MD: Johns Hopkins University Press, 1980.
- Meltzer, Kevin, ed. "The Perl Poetry Contest." *The Perl Journal* 4 (2000): 49–54.
- Merges, Robert P. "One Hundred Years of Solicitude: Intellectual Property Law, 1900–2000." *California Law Review* 88 (2000): 2187–2240.
- Metropolis, N., J. Howlett, and Gian-Carlo Rota, eds. *A History of Computing in the Twentieth Century*. New York: Academic Press, 1980.
- Moody, Glyn. *Rebel Code: The Inside Story of Linux and the Open Source Revolution*. Cambridge, MA: Perseus, 2001.
- Morrison, Philip and Emily Morrison, eds. *Charles Babbage and His Calculating Engines*. New York: Dover Publications, 1961.
- Morrison, Philip and Phylis Morrison. "100 or So Books That Shaped A Century of Science." *American Scientist* 87 (1999): 57–62.
- Nalley, Elliot Turner. "Intellectual Property in Computer Programs." *Business Horizons* 43 (July/August 2000): 43–51.
- Naur, Peter, Brian Randall, and J. N. Buxton, eds. *Software Engineering: Proceedings of the NATO Conferences*. New York: Petrocelli/Carter, 1976.
- Nichols, Peter. *Modernisms: A Literary Guide*. London: Macmillan, 1995.
- Noland, Carrie. *Poetry at Stake: Lyric Aesthetics and the Challenge of Technology*. Princeton, NJ: Princeton University Press, 1999.
- Norris, Margot. *The Decentered Universe of Finnegans Wake: A Structuralist Analysis*. Baltimore, MD: Johns Hopkins University Press, 1976.

- Nunberg, Geoffery, ed. *The Future of the Book*. Berkeley: University of California Press, 1996.
- O'Connor, Erin. *Raw Material: Producing Pathology in Victorian Culture*. Durham: Duke University Press, 2000.
- Ong, Walter J. *Orality and Literacy: The Technologizing of the Word*. New York: Routledge, 1982.
- Papert, Seymour. *Mindstorms: Children, Computers, and Powerful Ideas*. New York: Basic Books, 1980.
- . *The Children's Machine: Rethinking School in the Age of the Computer*. New York: Basic Books, 1993.
- Paulson, William. *The Noise of Culture: Literary Texts in a World of Information*. Ithaca, NY: Cornell University Press, 1988.
- Pavlicek, Russell. *Embracing Insanity: Open Source Software Development*. Indianapolis, IN: Sams, 2000.
- Perloff, Marjorie. *Radical Artifice: Writing Poetry in the Age of Media*. Chicago: University of Chicago Press, 1991.
- Piaget, Jean. *Structuralism*. London: Routledge and Keegan Paul, 1971.
- Pirsig, Robert M. *Zen and the Art of Motorcycle Maintenance*. New York: Vintage, 1974.
- Poovey, Mary. *Uneven Developments: The Ideological Work of Gender in Mid-Victorian England*. Chicago: University of Chicago Press, 1988.
- Pugh, Emerson. *Building IBM: Shaping an Industry and Its Technology*. Cambridge, Mass.: MIT Press, 1995.
- Rabaté, Jean-Michel. *Joyce upon the Void: The Genesis of Doubt*. New York: St. Martin's Press, 1991.
- Rawlins, Gregory J. E. *Moths to the Flame: The Seductions of Computer Technology*. Cambridge, MA: MIT Press, 1996.

- Raymond, Eric. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. Sebastopol, CA: O'Reilly, 1999.
- Rhodes, Neil and Jonathan Sawday, eds. *The Renaissance Computer: Knowledge Technology in the First Age of Print*. New York: Routledge, 2000.
- Rice, Thomas Jackson. *Joyce, Chaos, and Complexity*. Urbana, IL: University of Illinois Press, 1997.
- Ritchie, David. *The Computer Pioneers*. New York: Simon & Schuster, 1986.
- Ritchie, Dennis and Ken Thompson. "The UNIX Time-Sharing System." *Communications of the ACM* 17 (July 1974): 365–75.
- Rosen, Saul. *Programming Systems and Languages*. New York: McGraw-Hill, 1967.
- Rosenberg, Donald. *Open Source: The Unauthorized White Papers*. Foster City, CA: IDG Books Worldwide, 2000.
- Rothenberg, Jeff. "Ensuring the Longevity of Digital Documents." *Scientific American* (January 1995): 42–47.
- Rothenberg, Jerome and Steven Clay, eds. *A Book of the Book: Some Works and Projections about the Book and Writing*. New York: Granary Books, 2000.
- Roughley, Alan. *Reading Derrida Reading Joyce*. Gainesville, FL: University Press of Florida, 1999.
- Russo, Mary. *The Female Grotesque: Risk, Excess, and Modernity*. New York: Routledge, 1995.
- Ryan, Marie-Laure, ed. *Cyberspace Textuality: Computer Technology and Literary Theory*. Bloomington: Indiana University Press, 1999.
- - -. *Narrative as Virtual Reality: Immersion and Interactivity in Literature and Electronic Media*. Baltimore: Johns Hopkins University Press, 2001.
- Salus, Peter H. *A Quarter Century of UNIX*. Reading, MA: Addison-Wesley, 1994.
- Sammet, Jean E. *Programming Languages: History and Fundamentals*. Englewood Cliffs, NJ: Prentice-Hall, 1969.

- Sanders, Barry. *A Is for Ox: Violence, Electronic Media, and the Silencing of the Written Word*. New York: Pantheon, 1994.
- Scholes, Robert. *Structuralism in Literature: An Introduction*. New Haven: Yale University Press, 1974.
- Sculley, John and J. A. Byrne. *Odyssey: Pepsi to Apple . . . A Journey of Adventure, Ideas, and the Future*. New York: Harper & Row, 1987.
- Sewell, Wayne. *Weaving a Program: Literate Programming in WEB*. New York: Van Nostrand Reinhold, 1989.
- Shasha, Dennis and Cathy Lazere. *Out of Their Minds: The Lives and Discoveries of 15 Great Computer Scientists*. New York: Copernicus, 1995.
- Shneiderman, Ben. "The Relationship Between COBOL and Computer Science." *Annals of the History of Computing* 7 (1985): 348–52.
- Simon, Herbert. "How Complex are Complex Systems?" *Philosophy of Science Association* 2 (1976): 507–522.
- . "The Architecture of Complexity." *The Sciences of the Artificial*. Cambridge, Massachusetts: MIT Press, 1981. 192–229.
- Simpson, Lorenzo. *Technology, Time, and the Conversations of Modernity*. New York: Routledge, 1995.
- Slouka, Mark. *War of the Worlds: Cyberspace and the High-Tech Assault on Reality*. New York: Basic Books, 1995.
- Sluga, Hans. *Heidegger's Crisis: Philosophy and Politics in Nazi Germany*. Cambridge, MA: Harvard University Press, 1993.
- Smith, R. E. "A Historical Overview of Computer Architecture." *Annals of the History of Computing* 10 (1989): 277–303.
- Solomon, Margaret C. *Eternal Geometer: The Sexual Universe of Finnegans Wake*. Ithaca, NY: Cornell University Press, 1988.

- Sondheim, Alan, ed. "Codework." *American Book Review* 22 (September/October 2001): 1–8.
- Stallman, Richard. "The GNU Operating System and the Free Software Movement." *Open Sources: Voices from the Open Source Revolution*. Ed. Chris DiBona, Sam Ockman, and Mark Stone. 53–89.
- . "The GNU Manifesto." URL: <http://www.gnu.org/gnu/manifesto.html> (1993).
- Standage, Tom. *The Victorian Internet: The Remarkable Story of the Telegraph and the Nineteenth Century's On-line Pioneers*. New York: Walker, 1998.
- Stephenson, Neal. *In the Beginning . . . Was the Command Line*. New York: Avon Books, 1999.
- Stoustrup, Bjarne. *The C++ Programming Language*. Reading, MA: Addison-Wesley, 2000.
- Sussman, Herbert. "Machine Dreams: The Culture of Technology." *Victorian Literature and Culture* (2000): 197–204.
- Swade, Doron. *The Difference Engine: Charles Babbage and the Quest to Build the First Computer*. New York: Viking, 2000.
- Sypher, Wylie. *Literature and Technology: The Alien Vision*. New York: Random House, 1968.
- Talbot, Stephen. *The Future Does Not Compute: Transcending the Machines in Our Midst*. Sebastopol, CA: O'Reilly, 1995.
- Theall, Donald. *Beyond the Word: Reconstructing Sense in the Joyce Era of Technology, Culture, and Communication*. Toronto: University of Toronto Press, 1995.
- . *James Joyce's Techno-Poetics*. Toronto: University of Toronto Press, 1997.
- Thomson, Iain. "From the Question Concerning Technology to the Quest for a Democratic Technology: Heidegger, Marcuse, Feenberg." *Inquiry* 43 (June 2000): 203–16.
- Torvalds, Linus and David Diamond. *Just for Fun: The Story of an Accidental Revolutionary*. New York: HarperCollins, 2001.
- Tucker, Allen. *Programming Languages*. Reading, MA: Addison-Wesley, 1977.

- Turing, Alan. "On Computable Numbers, with an Application to the Entscheidungsproblem." *Proceedings of the London Mathematical Society* 46 (1936): 230–65.
- Turkle, Sherry. *Life on the Screen: Identity in the Age of the Internet*. New York: Simon & Schuster, 1995.
- Valente, Joseph. "The Politics of Joyce's Polyphony." *New Alliances in Joyce Studies*. Ed. Bonnie Kime Scott. Newark, DE: University of Delaware Press, 1988. 57–69.
- Van der Spiegel, Jan, James F. Tau, Titiimaea F. Ala'ilima, and Lin Ping Ang. "The ENIAC: History, Operation, and Reconstruction in VLSI." *The First Computers—History and Architectures*. Ed. Raúl Rojas and Ulf Hashagen. 121–78.
- Von Neumann, John. *The Computer and the Brain*. New Haven: Yale University Press, 1958.
- . *Theory of Self-Reproducing Automata*. Urbana, IL: University of Illinois Press, 1966.
- Wall, Larry, Tom Christiansen, and Jon Orwant. *Programming Perl*. 3rd edition. Sebastopol, CA: O'Reilly, 2000.
- Wallace, James and J. Erikson. *Hard Drive: Bill Gates and the Making of the Microsoft Empire*. New York: Wiley, 1992.
- Warren F. Motte, Jr. *Oulipo: A Primer of Potential Literature*. Normal, IL: Dalkey Archive Press, 1998.
- Wayner, Peter. *Free for All: How Linux and the Free Software Movement Undercut the High-Tech Titans*. New York: HarperCollins, 2000.
- Wegner, P. "Programming Languages—The First 25 Years." *IEEE Trans. Computers* 12 (1976): 1207–25.
- Weinberg, Gerald. *The Psychology of Computer Programming*. New York: Van Nostrand Reinhold, 1971.
- . *Understanding the Professional Programmer*. New York: Dorset House Publishing, 1998.
- Weir, Lorraine. *Writing Joyce: A Semiotics of the Joyce System*. Bloomington, IN: Indiana University Press, 1989.

- Weisfeld, Matt. *The Object-Oriented Thought Process*. Indianapolis, IN: Sams, 2000.
- Weizenbaum, Joseph. *Computer Power and Human Reason*. San Francisco: W. H. Freeman, 1976.
- Wertheim, Margaret. *The Pearly Gates of Cyberspace: A History of Space from Dante to the Internet*. New York: Norton, 1999.
- Wexelblat, Richard, ed. *A History of Programming Languages*. New York: Academic Press, 1981.
- Wilbur, Marcia. *The Digital Millennium Copyright Act*. Lincoln, NE: Writers Club Press, 2000.
- Wilkes, M. V., D. J. Wheeler, and S. Gill. *The Preparation of Programs for an Electronic Digital Computer*. Reading, MA: Addison-Wesley, 1951.
- Williams, Michael. *A History of Computing Technology*. Washington, D.C.: IEEE Computer Society Press, 1997.
- Winograd, Terry. "Heidegger and the Design of Computer Systems." *Technology and the Politics of Knowledge*. Ed. Andrew Feenberg and Alastair Hannay. Bloomington: Indiana University Press, 1995. 108–27.
- Ziarek, Krzysztof. "Powers to Be: Art and Technology in Heidegger and Foucault." *Research in Phenomenology* 28 (1998): 162–94.
- Zimmerman, Michael E. *Heidegger's Confrontation with Modernity: Technology, Politics, and Art*. Bloomington: Indiana University Press, 1990.