Weird Code: Gender and Programming Languages

by

Brandee Easter

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(English)

at the

UNIVERSITY OF WISCONSIN-MADISON

2019

Date of final oral examination: June 10, 2019

This Dissertation is approved by the following members of the Final Oral Committee:

Christa Olson, Associate Professor, English
James J. Brown, Jr., Associate Professor, English, Rutgers University-Camden
Caroline Gottschalk Druschke, Associate Professor, English
Adrienne Massanari, Associate Professor, Communication, University of Illinois at Chicago
Kate Vieira, Associate Professor, Curriculum and Instruction

Table of Contents

Abstract

This dissertation investigates how digital rhetorics are gendered, looking to rhetorics about, with, and in code to understand embodied power asymmetries in digital contexts. Although problems caused by these asymmetries have recently received much popular and scholarly attention, these debates often stall because blame falls to a supposedly bodiless machine. However, this project maintains that it is precisely these ephemeral and disembodied constructions of the digital that are co-constitutive of power differences. Taking weird programming languages as its focus, this dissertation exposes gendered rhetorical mechanisms of computational expression, ultimately arguing that a feminist rhetorical code studies can help interrogate digital forces of inequity, harassment, and embodied danger, as well as imagine feminist and queer computational possibilities.

This project builds from the premise that digital infrastructures, including hardware, software, and code, are powerful and material, yet often invisible, sources of rhetorical force. It focuses on a class of experimental programming languages that are not for "real world" use but rather for creative expression. These are called weird, or esoteric, programming languages, and they include programming languages created as parody, proof of concept, software art, or sport. Through a feminist and queer approach, I find that these programming languages can help interrogate how code itself participates in gendered rhetorical forces because they make arguments about programming through code, ultimately revealing how widespread constructions of digitality as neutral, spaceless, and body-less work to produce and enforce power asymmetries. By imagining the digital as arhetorical, such constructions obscure the ability to see human forces and effects in digital contexts and stall discourse that could move toward change.

Moving through chapters on aesthetics, difficulty, pain, and spatiality in programming, this dissertation develops a theory of programming as embodied, affective, and material. Chapter one, "Weird Code," introduces weird programming languages as ideal sites for interrogating programming's rhetoricity. Chapter two, "Hard Code," examines the most famous weird programming language, named "brainfuck," to understand how and why some weird languages intentionally make coding difficult, especially alongside the accessibility promoted by the coding literacy movement. Chapter three, "Painful Code," further interrogates painful weird programming languages to better understand how theories of sensation can challenge ideals of digitality as unfeeling. Chapter four, "Broken Code," takes up an anti-feminist programming language hoax to theorize how misogyny can occur through gendered claims to space in code. Taken as a whole, this dissertation dismantles ideals of programming as arhetorical and argues for the real, material, embodied, and affective forces of digital infrastructures.

Acknowledgements

I want to thank my family—my mom, dad, brother, grandmother, and grandfather—for their unwavering support.

I am grateful to have had generous and kind mentors. I couldn't have completed this project without the support and encouragement of the members of my committee—Christa Olson, Jim Brown, Caroline Gottschalk Druschke, Adrienne Massanari, and Kate Vieira. I am especially thankful to my chair, Christa, who models daily the kind of scholar I hope to become. She is not only smart and kind, but also thoughtful and transparent about the experience of being in academia. I'm thankful to have learned from Morris Young, who has taught me to approach my work with empathy and care.

I'm further thankful for the mentorship I've had beyond UW-Madison. I would not have pursued a graduate degree without Albert Pionke at the University of Alabama, who introduced me to scholarship and mentored me through my first project. Thanks to Kevin Brock at the University of South Carolina for being encouraging and enthusiastic about my work, and thanks to Jim Brown for being no less a mentor from hundreds of miles away than when he was here.

This dissertation would simply not exist without the communities of friends, writers, and readers I have been fortunate to be surrounded with. Thank you to Lindsay Perrine, Allyson Roller, and Sara Vandagriff for modelling the pursuit of education and supporting each other in the process. Thank you for always being there, even when I got lost. Thank you to the community of graduate students across the English department who I have been lucky to work with and learn from. I'm especially thankful for my writing group—Meg Marquardt, Emily Loney, and Lisa Marvel Johnson—for being generous readers and writers.

To Meg Marquardt I owe much more than words in an acknowledgements page. Meg is endlessly kind, encouraging, and was first to make me feel that I was part of this program. Through coursework to writing "twin" dissertations, Meg has helped me not give up on my work or myself, and I'm excited about our future collaborations—you're stuck with me.

Finally, I could not have pursued graduate study without the support of my partner. Jason is the kind of person who makes moving across the country seem easy, and I am thankful for that calm presence every day.

Preface

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

—A "Hello, World!" program in Malbolge

When Greg Wilson got his first job as a programmer in 1982, he had a revelation: "For the first time, I saw that programs could be more than just instructions for computers. They could be as elegant as well-made kitchen cabinets, as graceful as a suspension bridge, or as eloquent as one of George Orwell's essays" (xv). But, these qualities of code's elegance, grace, and eloquence aren't usually how programming is often talked about or taught. Instead, Wilson suggests that we only seem to look at programs when they are bad—to fix their "bugs." Because of this tendency, Wilson opens his 2008 edited collection *Beautiful Code* with this argument for attending seriously to code as art—as something that could and should be beautiful. He then asks 36 contributors to share an example of beautiful code. Throughout their individual analysis, these contributors emphasize the beauty in concision, clarity, and effectiveness. These values, however, seem at odds with the "Hello, World!" program written in Malbolge that opens this chapter. It isn't elegant or graceful. It's written in all caps with strings of punctuation and a lack of spaces that make it difficult for human readers. It is *ugly*.

In this preface, I lay the groundwork for understanding what it means for code to be ugly as a rhetorical and intentional aesthetic. Throughout this dissertation, I show that it has been a recurring pattern of argument for programmers to assert their masculine (even divine) superior identity through the development and performance of coding aesthetics. When "Real Programmers" argue for *how* coding should look, feel, or be written, they are also arguing for

who can code. I argue that the ugliness of contemporary weird programming languages is a particular aesthetic that works to establish borders between "real" and less-than programmers that has consequences for who has access to code. By taking programming's aesthetic values to an extreme, ugly weird programming languages reveal how masculine expression and identity can be expressed in coding style, ultimately revealing that what is "real" in programming—in both normative and weird contexts—is in fact deeply tied to what is masculine.

This connection between masculinity and code aesthetics has been particularly well captured by Ed Post's 1982 "Real Programmers Don't Use PASCAL," a play on the 55 week bestseller *Real Men Don't Eat Quiche*. Tongue-in-cheek like the original, Post's version lays the ground rules for how to be a real programmer:

> Back in the good old days… the Real Men were the ones that understood computer programming, and the Quiche Eaters were the ones that didn't. A real computer programmer said things like "DO 10 I=1,10" and "ABEND" (they actually talked in capital letters, you understand), and the rest of the world said things like *"computers are too complicated for me"* and *"I can't relate to computers -- they're so impersonal."*

To identify what is real in programming, Post points us to style. Look at how they talk and write—"capital letters, you understand." They speak in numbers, commands, and equations, unlike "the rest" who write about the self and feeling in calmly sequenced sentences. Post continues to argue that the easiest way to make this distinction is to see which programming language they use: "Real Programmers use FORTRAN. Quiche Eaters use PASCAL."

In what follows, I offer an overview of how programming languages became writing that could be beautiful or ugly in order to understand this relationship between programming

languages and masculine programmer identity. Because early programming didn't resemble writing at all, I begin with programming as hardwiring in the 1940s. I then sketch out programming's development as writing through machine, assembly, and high-level languages. This shows how programming languages work increasingly away from the hardware of the machine toward more abstracted, "natural" human (often English) language, which makes programming more accessible, changes who programs, and poses a threat to the "real" programmer.

What is Code?

In 2015, *Businessweek* editor Josh Tyragiel realized that "people have been faking their way through meetings about software, and the code that builds it" for decades because we have so little understanding of it (Ford). He asked writer and programmer Paul Ford, "Can you tell me what code is?" Ford, simply and perhaps surprisingly, replied only, "No." This exchange led to Tyragiel commissioning a 2,000 word article from Ford intended to translate "the shape-shifting beast" that is code for the average American (Nguyen). However, even though Ford has been a programmer for 20 years, this short article quickly ballooned into 38,000 words—an entire issue of the magazine—as he struggled to contain the notoriously ephemeral "code." This now widely recognized article walks through history, personal narrative, and coding examples to try to convey what the vapory concept of code is. Although it never lands on a singular definition, Ford's work is an excellent popular reminder of how difficult software can be to pin down and that this elusiveness is part of software's power (Chun).

A major contribution of critical code studies and digital humanities has been to make space for code as a humanistic object of study through attention to its poetic, artistic, and literary features. In this dissertation, I'm not interested in *if* code is aesthetic. I'm interested in how these aesthetics serve a rhetorical purpose—specifically, to say something about who the programmer is and how accomplished they are. I argue that, in these questions of writing, aesthetics, and identity, composition and rhetoric has important perspectives to contribute. In rhetorical theory, aesthetics has largely been taken up through the lens of style. Although style has often been an overlooked canon of rhetoric, recent scholars have urged us to reconsider this perspective (Duncan and Vanguri). Sometimes understood as an embellishment or decoration, style has been derided as superfluous and not the "real substance" of rhetoric. Barry Brummett argues for the importance of style as its own system of communication with rhetorical force and effects. If we redefine style to include more broadly all the ways we "think about presenting ourselves to others," he argues we can better understand how style is "a major if not the major rhetorical system at work in the world today" (1). Although Brummett's primary focus in on the relationship and development in relationship to capitalism, his project also speaks to style's connection to intersectional identities and and demonstrates rhetorics of style's material consequences. In this view, style is "a quality intrinsic to the writer" as an individual and their voice (Bacon).

In addition to making these connections between style and identity, several rich strands of composition studies also contribute ways to interrogate how aesthetic values, norms, and prescriptions of writing are bound up in intersectional power differentials. Composition scholars have examinited how the teaching of style also communicates what "type of writer" a student

should be (Lockhart 18). In other words, we say what "good" writing is, we're also saying who is, or can be, a good writer. These ideas have been particularly well developed in composition, rhetoric, and literacy scholarship on the interconnectedness of language, pedagogy, assessment, and race (Canagarajah, Smitherman). In fact, any time we make judgments about writing, we are also unavoidably dealing with race (Inoue).

Before Programming was Writing, Computers were People

However, programming has not always been recognizable as writing or as a form that could have an aesthetics. In fact, Maurice Black argues that, unlike technologies such as the automobile or train, the genealogy of the computer is more difficult to trace. Our current computational devices, as well as the types of programming made possible by them, "bear almost no resemblance to the hulking thirty-ton ENIAC," the Electronic Numerical Integrator and Computer, which is a common touchstone in computation's history because it was the first computer to be both electronic and general purpose (40). In fact, programming this 1940s computer would likely be "virtually unrecognizable" to current programmers (40).

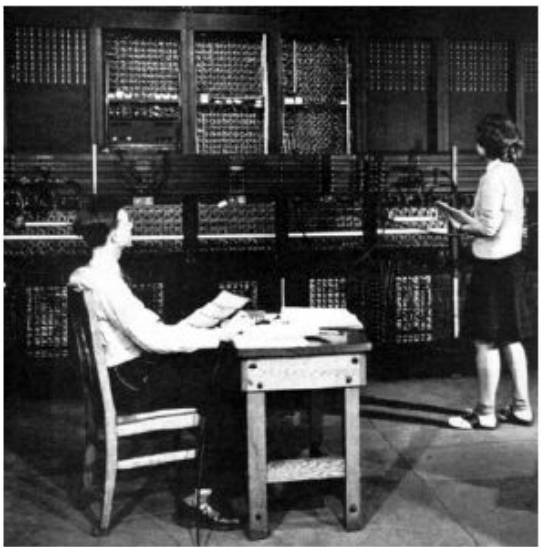


Fig. 1. "ENIAC being set up," *Computer History Museum*, Gift of J.A.N. Lee, https://www.computerhistory.org/revolution/birth-of-the-computer/4/78/319. Accessed 20 May 2019.

Programming before written code is particularly well captured in a 1940s photograph of the ENIAC (fig 1.). A man sits at a table in front of notes. A woman stands a few feet away in front of nearly floor-to-ceiling panels of switches and plugs, listening for his directions. As he—the programmer—reads off instructions, she—the computer—sets switches and connects wires to execute the program. His intellectual work is dictated to her for rote execution.

This scene shows what programming looked like before the creation of programming languages. Programming was plugging in cables or turning dials and had "little relationship to any kind of written word" (Montfort, et al. 160). Instead of manipulating symbols, programming was the arrangement of physical pieces of computer hardware. This was time consuming, as can especially be seen in troubleshooting efforts. Because one of the ENIAC's 18,000 vacuum tubes failed every day or two, checking and maintaining all the hardware was a consuming and difficult task ("ENIAC"). However, the ENIAC was a revolution because the machine could be

reprogrammed electronically. And yet, while the ENIAC could theoretically run more than one program or calculate more than one problem, rewriting the hardware to do so could take *days or weeks*. Troubleshooting errors was no easier. Each time the ENIAC was to perform a different calculation or function, the machine was rewired. Although the ENIAC was a great improvement in speed, waiting days or weeks to run a different program was a major disadvantage that eventually led to the development of electronic memory storage and programming languages.

For Wendy Hui Kyong Chun, this history, especially as in this photograph, demonstrates "the dream of 'programming proper'": a woman computer responding "Yes, Sir" to a series of commands from a male programmer (Chun 33). In this legacy of "Yes, Sir," we need to ask how rhetorical possibility is negotiated with and within gendered digital systems. Furthermore, this moment—when programming started commanding a machine instead of a "girl"—is also when computation *became writing* (Vee). Software, although previously defined as everything that was not hardware, now refers to the programs, or sets of instructions, run on computers, represented and constructed in code. Although this revolution may seem to make the gendered structures of programming obsolete, in reality, these technological changes merely obscured these relationships.

Machine and Assembly Languages

To make the leap from hardwiring to programming languages, at least two important theoretical revolutions are worth reviewing: Turing machines and von Neumann architecture. These two theories made it possible for computers to run more than one program and to store those programs electronically *in memory*, without which programming languages would not

exist. In 1936, Alan Turing theorized a machine that could prove the computability of real numbers. There are three parts of this imagined machine, two of which matter for our purposes here: 1) an infinite tape, divided into cells that each can contain one symbol, 0s and 1s, and 2) a read/write head, or scanner, that could handle input and output from a single cell on the tape. In this fanciful imagining below from *American Scientist* (fig. 2), a Turing machine is illustrated in the process of computing: the machine writes to the cell under the scanner, changes state, and moves to the right or left.

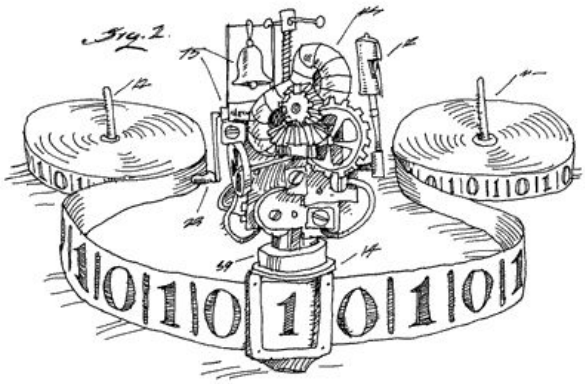


Fig. 2. Chaitin, Gregory J. "Computers, Paradoxes and the Foundations of Mathematics," *American Scientist,* vol. 90, no. 2, 2002, pp. 164-71.

This theoretical machine matters for the history of computation because it allowed computation to be expressed *as writing*. Additionally, the machine was also proof of recursive functions, which is when a function calls itself until a condition is met.[1] Because "the description of the machine can itself be expressed as a sequence of symbols," Turing conceptualized of a universal machine that could carry out the computation of any other machine (Mahoney 79).

[1] Any programming language that can complete these functions is considered "Turing complete" and can theoretically execute any task accomplishable by computers. This benchmark, though achieved by very nearly all higher order programming languages, becomes an important validation of certain esoteric programming languages and practices, which I discuss in chapter 2.

This is the invention of general purpose, re-programmable computing—machines that can run the programs of other machines.

Turing's machine allowed John von Neumann to conceptualize of programs that could be stored as data electronically in computer memory, instead of physically in punched cards or hardwired through cables. Like the Turing machine, von Neumann's computer has three parts: 1) memory, for storing data; 2) a processing unit, for performing calculations; 3) a control unit, for interpreting instructions. The important paradigm shift here is that this development is about "getting the computer to do the programming" (80). That is, instead of hardwiring new functions or inserting punch tape, the computer can store instructions in electronic memory, providing greatly increased speed and flexibility for programming and program execution. Although the finite memory of this architecture seemed unworkable at first, the development of programming languages are themselves proof of the success of von Neumann's computer[2] (79). The Turing machine and von Neumann architecture then are what allows programming to be writing. This becomes the first generation of programming languages.

The 0s and 1s of the Turing machine are now commonplace to us as symbols that communicate directly with computers. This first generation of programming languages, called machine code, indicates instructions that can be executed directly by the computer without an assembler or interpreter. In other words, the instructions given in machine code tell the computer to directly perform a task. As transcriptions of the "on" and "off" states of physical transistors, this binary system produced a symbolic way to represent any value and execute computations. Although, ultimately, all numbers are stored in computers as ones and zeros, or high- and

[2] In fact, this design is still used in most computers today.

low-voltage signals respectively, the specifics of a machine language are tied to that machine's physical architecture. Because the operations in machine code refer to specific locations—transistors and memory storage— and processes, binary code is not universally applicable. In other words, each machine can only understand programs written for its physical architecture.[3]

Machine code, however, is highly unreadable by humans.[4] While efficient for the machine, the gap for the human reader greatly slows down a programmer's work, which makes composing in these languages tedious and error prone. To combat the unreadability of machine code, a second generation of programming languages, called assembly languages, were developed to translate machine codes into more human readable forms. Like machine code, these languages are specific to a particular system's hardware, and they are usually a very close symbolic or mnemonic translation of the machine code. Some of these changes included giving names instead of numbers to storage locations and using English-like abbreviations for operators.

The development of assembly languages and digital assemblers made code much more readable by humans and allowed for faster composition and machine execution. While machine languages remain useful for programming problems that are close to the hardware, such as operating systems, they are otherwise not preferred above high-level languages. Although their close ties to a specific machine make them ideal for hardware-related problems, this is also a major drawback for more general purpose programming because they are not portable to other machines. While the invention of macros, or instructions that can be translated into several

[3] This problem is greatly improved by shared architectures between many computers. For example, Intel's x86 architecture has been compatible with Intel processors since 1976.

[4] One interesting context Mahoney raises for this disparity between human and machine readers is in early debates about the ability to copyright a program written in machine code. These debates reveal how machine code was still unrecognizable as writing, with human audiences and authors, to many in this form.

machine language instructions, made assembly languages much more usable, they were still not very human friendly and have the appearance of "computerese" ("Assembly"). It takes one more generation of languages to get to languages that aren't machine specific. Free from any single machine's architecture, experimentation and expression with style also becomes more possible.

Programming gets BASIC

This desire—for programming with familiar language that worked on many machines—was a primary motivator for the development of high-level programming languages. Languages of this third generation are what we usually mean when we speak of "code." Writing at a high-level means abstracted from any specific machine, which enables the ability to write in familiar (often English) language and to use the same program on many machines. This is possible through the use of compilers or interpreters, both of which translate to the human friendly high-level code machine language. Released from the specifics of any given machine, programmers were able to create and work with languages that were focused on purpose. Languages, like BASIC (Beginner's All-purpose Symbolic Instruction Code), were developed to be general purpose and translatable for many different types of tasks, machines, and authors. Other languages aim to solve more specific problems, such as COBOL (Common Business Oriented Language) which was created for data processing needs of businesses.

Since BASIC, high-level programming languages have been created for a variety of purposes, authors, and contexts. While the hundreds of high-level programming languages in existence are diverse, there are some commonalities that they share. They are unattached to a particular machine. They attempt to use human-friendly language, and they must be translated to

machine code (whether through compiling or interpreting) in order to execute. This separation, between the "source code" as written by the programmer and the "object code" actually executed by the machine, is what has allowed software to materialize as a "thing" that has owners and is often highly valued property (Black, Chun).

High-level languages continue to solve new problems and attract new coders with the promise of lucrative and stable careers. In fact, these languages been so successful that we've reached a moment where coding is emerging as a literacy (Vee). Another sign that high-level languages have been successful in providing access for humans is *Beautiful Code*, the collection I opened with. *Beautiful Code* champions the craft, elegance, and efficiency of programs that solve problems and can even make the world better.

In some ways, this preface has offered a textbook history of programming. It is linear, progress oriented, and there aren't a lot of people involved beyond a few geniuses and nameless others. There's also a triumphant ending and the promise of a better and more beautiful future. However, I begin here because the languages in my dissertation grow alongside, twist into, and disrupt such narratives and ideals. These languages are *ugly*.

Chapter 1

Weird Code: An Introduction

Print ("Hello, World!")

—A "Hello, World!" program in Python

The problems at the heart of this dissertation frequently appear in the news. One recent and persistent example of has been Facebook's defense of itself after the 2016 election. Its representatives consistently argued that Facebook was "just a platform." However, this belief—that our technologies are inherently value neutral—has consequences for how we discuss, develop, and debate technology and its effects on society. When Facebook denies responsibility for its effects by invoking machinic neutrality, our ability to debate solutions and enact technological change stalls.

"Weird Code" builds instead from the premise that digital infrastructures—including hardware, software, and code—are powerful and material, yet often invisible, sources of rhetorical force. They are active and constitutive; far from neutral. To make that case, my dissertation focuses on a class of experimental programming languages that are not for "real world" use but rather for creative expression, including parody, proof of concept, software art, and sport. By stepping away from the "real world," these languages show us the very real architectures that govern programming.

When we talk about code, we usually refer to a symbol system that allows humans to express information, especially instructions, to a machine. In the example program opening this introduction, the computer is instructed to display the words "Hello, World!" on the screen. As

the name implies, this is the first program usually learned by beginning programmers, and there is a version in every language. I'll be working with "Hello, World!" programs throughout my dissertation because they are quick ways to access the overall paradigm of a language. For example, this "Hello, World!" program shows that the language Python prioritizes concision, readability, and simplicity. This has made Python well-suited for many general-purpose programming challenges, including web development and data analysis.

However, the programming languages in my project are not made for this purpose of computational production. They are creative languages. Some are made as jokes, such as only using quotes from Arnold Schwarzenegger movies:

```
IT'S SHOWTIME

TALK TO THE HAND "hello world"

YOU HAVE BEEN TERMINATED
```
(Hartikka)

They are created for artistic expression, such as, in this language named for Piet Mondrian (Morgan-Mar), where you can communicate a "Hello, World!" program using an arrangement of colored blocks (see fig. 1):

Fig. 1. Morgan-Mar, David. "Piet," [programming language], *Esolangs Wiki*, https://esolangs.org/wiki/Piet. Accessed 26 May 2019.

And these languages are largely made as proof of concept to test the limits of computational expression. There is even a language that answers the question: can you code using only the word “chicken”? The answer is yes:

```
chicken chicken chicken chicken chicken chicken chicken chicken
chicken chicken chicken chicken chicken chicken chicken chicken
chicken chicken chicken chicken
chicken chicken chicken chicken chicken chicken chicken chicken
chicken chicken chicken chicken chicken chicken chicken chicken
chicken chicken chicken chicken
chicken chicken chicken chicken
chicken chicken chicken chicken chicken chicken chicken chicken
chicken… (Söderstedt, excerpted)
```

I focus my dissertation here, on what are called “weird” or “esoteric” programming languages, for two primary reasons. First, what weird languages make undeniable is that programming is a way to express things to humans, not only machines. Chicken is certainly meant for our amusement. Some weird languages take this focus on human audiences a step further by resisting execution by the machine altogether, existing then *only* for human audiences. Secondly, weird languages reveal the very real architectures that govern programming by taking norms, values, and structures to their extreme. Unquestioned programming principles, such as concision and functionality, are made visible by pushing them to both extremes. Chicken is funny because efficiency is central to what we believe code is, even if unconsciously. And Chicken is anything but efficient.

Through case studies in weird programming languages, my dissertation forwards three central arguments: 1) Code is always material, embodied, and affective; 2) Ideals about coding as otherwise (spaceless, disembodied, and unfeeling) are themselves rhetorical constructions, built and sustained not only by the ways we talk about code but also through coding practices and in code itself; and 3) Code's supposed arhetoricity works to stall change around issues of embodied inequality, harassment, and harm in digital contexts. Ultimately, I argue that inequality is infrastructural, not incidental, to our digital worlds, and that rhetorical approaches are vital to change. This analysis helps us understand why discourse around digital change so often fails. Because we can't fix technology if we aren't persuaded that technology is the problem, one of the greatest barriers to digital change is *rhetorical*, not technological.

Locating Digital Rhetoric

Scholars in digital humanities, digital rhetoric, literacy studies, critical code studies, and software studies have argued for researchers to attend to digital infrastructures, including code, outside of computer science (Brown, Hayles, Marino, Sample, Vee). However, early engagement with digital contexts and objects has often been marked by a bias toward the interface, or what scholars have critiqued as a "screen essentialism" (Montfort, Kirschenbaum). In composition and rhetoric, this has led to a focus on discourse happening in digital spaces and visual representations on displays as central objects of study. Work in this vein, such as Lisa Nakamura's *Digitizing Race,* has been important for exploring how embodied differences such as race and gender have been represented and imagined in digital contexts. Drawing on visual culture studies, Nakamura investigates representations of racialized and gendered bodies,

including avatars, online signatures, music videos, and films to argue that users are not disembodied online but rather re-embody themselves through these visual forms.

However, informed by object-oriented and posthumanist theories, digital rhetoricians now ask how the computer system itself—its hardware, software, infrastructures, and procedures—is essential to the meaning-making process in digital contexts (Brown, Bogost, Montfort et al., Rieder). In moving beyond the screen, digital rhetoricians have developed and contributed to the interdisciplinary fields of software studies, platform studies, and critical code studies—all of which draw attention to how digital infrastructures shape and exert rhetorical force. These bodies of scholarship, which intersect with and branch out from rhetorical studies, are all grounded on approaching code as writing*,* which includes attention to code's authors, contexts, purposes, and audiences, both human and machine (Vee and Brown). While arguments against code's rhetoricity have relied on presenting code as a value neutral, symbolic representation of mathematical procedures, digital rhetoricians have turned to rhetorical features such as ethos, audience, style, genre, and enthymeme to understand how code is rhetorical, and often by highlighting the human authors and audiences in code (Brock, Brock and Mehlenbacher).

As software becomes increasingly and "deeply woven into contemporary life...in manners both obvious and nearly invisible," this need to attend to software as an object of study has been foundational to the creation of software studies (Montfort et al. xi). Software, as represented and constructed in code, has also been taken up as an object of study in digital humanities scholarship, particularly as developed in critical code studies to apply the tools of textual criticism to code (Marino, Sample). Code is a "peculiar kind of text" that, like writing, is

a symbolic communications system with purposes, authors, audiences, and contexts (Montfort et al. 3, Vee and Brown). As recent digital humanities scholarship has argued, code is material, cultural, social, political, and aesthetic, and these features have been used as evidence for the pressing need for scholarship on code outside of computer science (Cox and McLean). An early contribution of humanities scholars has been emphasize programming as a language, with such work drawing on the methodologies of textual criticism and especially on the Austinian, performative aspects of code (Hayles). In composition, rhetoric, and literacy studies, scholars have further dispelled code's identity as a value-neutral representation of mathematical procedures through attention to its relationship to writing. Annette Vee has argued that coding, as an emerging literacy, carries all the material and embodied consequences of reading and writing literacies.

Ian Bogost, however, highlights the limits of a purely textual approach to digital objects and code, proposing a new type of rhetoric—not exclusive to, but strongly afforded by computational objects. Although procedures are not unique to computation, Bogost argues that they are the "core affordance" of computational objects and demonstrates their rhetorical force through case studies in videogame (ix). Because programming is language that both *says* and *does*, the rules that it enacts have a particular rhetorical force, and a rhetorical critical code studies attempts to understand the aesthetic, expressive, and persuasive possibilities of both language and execution, text and procedure (Brock, Cox and McLean, Chun). This procedural rhetoric captures how processes can be argumentative and persuasive, and this contribution has been foundational to shifting scholars beyond the screen. Platform studies further explores how

those procedures are related to the materiality of computational objects to see how computation is shaping and shaped by its material infrastructures (Bogost, Montfort et al. 7).

This procedural focus has provided new ways to interrogate gender and normativity in digital contexts. Recently, Mark C. Marino has used these foci to ask "where and in what ways does code participate in a heteronormative superstructure?" (185). In this critical code studies reading of the AnnaKournikova worm, Marino explores how this exploit is "sexually charged" and relies on heteronormative logics to reproduce itself and spread, which ultimately demonstrates how attention to code can help us understand the connections between computing structures and "pervasive cultural logics at work," such as heteronormativity (186).

While Marino demonstrates the potential of procedural rhetorics, digital scholars have also called for us to attend more broadly to the complex rhetorical ecologies that include digital objects and contexts. To re-expand where we look for digital rhetoric beyond an often narrow focus on procedures, James J. Brown, Jr. outlines three *rhetorics* of software: "arguing *about* software, arguing *with* software, and arguing *in* software" (173). *Arguing about software* captures the most traditionally rhetorical area of these three: discourse about software, including authors, audiences, arguments, and contexts. *Arguing with software* captures dual meanings. First, software can be used as a tool for rhetorical expression and action. Secondly, it also draws our attention software as an "interlocutor" in rhetorical action: "when I attempt to use computation to build an argument, I work with and struggle against the affordances of a given platform or language" (176). *Arguing in software* describes the questions we might ask about how rhetoric happens in digital environments constructed by software. Brown points out that

these realms are not distinct, and they overlap with each other and as part of complex rhetorical ecologies.

In this project, I pursue such a re-expansion that looks for rhetorics about, with, and in software to help answer recent calls for digital rhetoricians to better understand embodied power differentials in digital contexts (Eyman, Noble). While importantly calling for humanists to attend materiality and specifically to the computer, digital rhetoric's focus on objects has often made the humans creating, using, or interacting with them invisible and even disembodied. Thus, one consequence of locating rhetorical force in objects and infrastructures beyond the screen has been a lack of attention to the body and the enforcement of difference that Nakamura's visual approach captures. This is important because the presumed, unmarked body is always normative, and our technologies reflect these dominant ideologies (Chun). When embodied power differentials are overlooked or misunderstood, they can be sustained—even automated—and blamed on a supposedly bodiless machine. When we say "the computer did it," we really mean that the differences being produced by algorithms and other infrastructures are natural and normal (Noble).

This methodological problem is representative of a broader popular and scholarly inability to connect technology with its gendered and raced consequences (Noble). While visual and textual focused digital scholarship has pursued these questions, such as Nakamura's work, more recent scholarship that focuses on parts of computation beyond the interface—software, hardware, procedure—has been noticeably disembodied. This challenge to reconcile posthuman contexts with humanist concerns is a central paradox for feminist science and technology scholars more broady, and feminist rhetorical science studies forwards embodiment and the

tensions it produces in these contexts as an important point of inquiry (Booher and Jung). This is an especially important challenge for feminist rhetoricians because, as Karma Chávez has argued, this "abstract body" is in fact a rhetorical mechanism of invisibility under which normative bodies maintain and enact power (244).

Meanwhile, feminist and queer scholarship on computation has largely focused either on the historical recovery of women innovators or analyzing gendered discourse online (Abbatte, Jane, Shetterly). While a feminist materialist turn in rhetorical studies has broadened this scope (Hallenbeck, Jack), digital objects and sites have remained challenging. Without this access to the underlying structure of all our digital environments, as Claudia Herbst argues, women are excluded from critiquing masculinist structures and building digital feminist worlds.

"Weird Code" responds to these calls by forwarding weird programming languages as ideal case studies for understanding theorizing a broader rhetorics of software because they productively blur categorical boundaries by using code itself to explore the structures of both specific languages and broad principles and practice. In other words, weird programming languages make arguments *about* code *with* and *in* code.

Chapter Overview

Weird Code moves through chapters on embodiment, feeling, and space to develop a theory of programming as embodied, affective, and material. Chapter two, "Hard Code," examines the most famous weird programming language, named "brainfuck," to understand how and why some weird languages intentionally make coding difficult. Brainfuck is notorious for its difficulty and esotericism, and this case study interrogates how, in contrast to the coding literacy

movement's emphasis on accessibility and ease, brainfuck uses challenges of mastery to assert an exclusive "true" (i.e. white, straight, masculine) programmer identity. Revealing programming's "subhumans," this chapter holds embodied power asymmetries in tension with posthuman contexts, objects, and questions, complicating neat human/posthuman divides particularly endemic to digital studies, and thereby revealing how digital infrastructures are themselves deeply co-constitutive of humanist problems. It argues that rhetorical constructions of the digital as disembodied are co-constitutive with real, material power asymmetries.

Chapter three, "Painful Code," interrogates intentionally painful weird programming languages to better understand how theories of sensation can challenge ideals of digitality as unfeeling. Through analysis of INTERCAL and bodyfuck, this chapter makes visible the gendered, embodied, feeling, even erotic, aspects of programming to challenge programming's supposed unfeeling utility. To trouble such distinctions between feeling and computation, this chapter shows that one way to expand rhetorics of software to better account for the digital sensorium is through *feeling machines*—approaching computation not only as machines that can be felt or machines that can feel, but also as interrelated producers and products of often intense feeling, including feeling built and felt in code.

Chapter four, "Broken Code," takes up an anti-feminist programming language hoax to theorize how misogyny can occur through gendered claims to space in code and demonstrates the rhetorical potential of non-executable code. In a parody language, the authors of C+=, an anti-feminist programming language and harassment campaign, use code to mock conversations and attempts to develop feminist code, but in practice actually disprove their own argument that code is "neutral" or "just math." Instead, they demonstrate how code is rhetorical, has human and

machine audiences, and can have gendered logics, purposes, and effects. I examine the code to discuss how the authors use procedures to argue that "a woman's place," as well as other non-male, non-normative identities, is not online. Specifically, I develop the concept of "digital manspreading" to understand how misogynistic rhetorical strategies used across digital spaces can also occur in code.

Taken all together, this project demonstrates the need for a feminist rhetorical approach to code. By attending to programming as real, material, embodied, and affective, this dissertation exposes masculinist technological structures and rhetorical mechanisms of digital inequality. In conclusion, "Queer Code" uses these critiques to consider implications, future directions, and feminist and queer possibilities for computational expression. Because this dissertation demonstrates that digital rhetorical force is inseparable from code itself, it shows the need to do more than recover and recruit diverse coders. This conclusion explores possibilities for creating transformative access to code in order to change not only who codes, but also how we use, think, talk, and interact with code.

Interchapter: Codework

Following critical code studies and software studies, coding is not only my object of study. It is also one of my methods. Quite simply, I could not have written this dissertation without also writing code. Computing is "one valuable mode in which to think," and as the case studies in this dissertation show, weird programming languages express and persuade through that mode (Montfort 4). In these interchapters, I want to make this method visible. However, these interchapters are not intended to reinforce digital humanities debates over hack/yack binaries. Arguments that all digital scholars "must code" privilege those that have already had the most access, limiting who gets to speak about our digital objects. We can see this even in the ways that such work is championed in digital scholarship using masculinist terms of "tool-building" (Klein). This dissertation would not exist if I needed to prove I had the appropriate skills of masculine "hacking" in order to be allowed to think about and with code.

Instead, I take exploratory and feminist approach to writing code. Drawing on scholars in feminist geography, design, and big data, these interchapters try potential practices for coding differently (D'Ignazio and Klein, Elwood and Leszczynski). I experiment with crafting practices often attributed to women and transparency in learning and writing code. I hope that such exploration and transparency, instead of being exclusionary, moves toward coding possibility and not prescription.

Chapter 2

Hard Code: Mastery, Masculinities, and Machines

++++++++++[>+++++++>++++++++++>+++<<<-]>++.>+.+++++++..+++.>++.<<

+++++++++++++++.>.+++.------.--------.>+.

—A "Hello, World!" program in brainfuck

In 1993, software engineer Urban Müller wanted to make his life easier. More specifically, he wanted to design the Amiga 2.0's smallest possible compiler. A compiler is a program that translates "high-level code"—which we now widely recognize as "code"—into a form executable by a machine, such as binary. To do this, Müller created a minimalist programming language that only used eight symbols, and this simplicity resulted in a compiler more than four times smaller the previous benchmark. For Müller, this success meant the language had "served its purpose," and he shared his creation to an online Amiga archive, expecting others might also find it useful for writing quick compilers. However, he never intended his language to be used for writing programs or to have a sustained following 25 years later. The language's extreme minimalism was too obfuscated and difficult for computational production. To express such "uselessness and confusion," Müller even named his language "brainfuck." Despite the language's apparent uselessness, there are now brainfuck solutions for some of programming's most difficult puzzles, over 250 derivative languages, and implementation in the videogame *Minecraft*. In a 2017 presentation, which he called "probably

the most useless of the whole conference," Müller reflected on brainfuck's unexpected popularity: "There's 700,000 pages on Google that mention brainfuck … What have I done?"

I begin here—with Müller's confusion—because brainfuck's enduring popularity in spite of its difficulty raises questions about the rhetoricity of programming languages. When we talk about code, we refer to a symbol system that allows humans to express information, usually instructions, to a machine. Implied in this understanding is a belief that code is most centrally about production and functionality, and contemporary programming languages reflect these ideals through an emphasis on ease, concision, and readability. For example, the "Hello, World!" program[5] in Python opening this dissertation shows a focus on such functionality through simplicity. Its readability and concision has made Python well-suited for many general-purpose programming tasks. Brainfuck, however, is not suited for such computational production. In the second "Hello, World!" program above, there are no recognizable words, numbers, phrases, or breaks in the code. In fact, this brainfuck program looks much more like a machine-level language than anything we would recognize currently as "code."

This obfuscation makes brainfuck's persistence and popularity difficult to explain if we understand code primarily as a way to communicate with machines. However, if we approach code as writing—with purposes, authors, and audiences, both human and machine—we can better understand brainfuck's peculiar circulation and rhetoricity (Brock, Montfort et al., Vee and Brown). Brainfuck is a "weird" or esoteric programming language, which are programming languages made for creative purposes, such as parody, artistic expression, and proof of concept (Mateas and Montfort). I turn to these weird programming languages, and brainfuck specifically,

[5] A "Hello, World!" program tells the computer to display the words "Hello, World!" on the screen. As the name implies, this is the first program usually learned, and there is a version in every language. I work with "Hello, World!" programs throughout this article because they are quick ways to access the overall design of a language.

for two reasons. First, what weird languages make undeniable is that programming is a way to express things to humans, not only machines. Brainfuck's circulation and use is not explainable as communication with a machine, and some weird languages take this focus on human audiences a step further by resisting execution by the machine altogether, existing then only for human audiences. Secondly, weird languages reveal the very real architectures that govern programming by taking norms, values, and structures to their extreme. Unquestioned programming principles, such as concision and functionality, are made visible through experiments that push at the edges of computational expression. In other words, weird programming languages make arguments *about* code *with* and *in* code.

To return to Müller's question, what he has done is make code hard again. Through this case, I am developing an understanding of code's hardness with three distinct yet concurrent meanings. I most centrally read "hard" as *difficulty*, especially as a challenge or competition. I argue that this difficulty is distinctly *masculine*, which I mean not only in opposition to a "soft" femininity, but also violently and forcefully so. Finally, code's hardness is *solid*. It is real and material, and as looking at this case within a larger context shows, code's hardness is also solidly sedimented and resistant to change.

With this understanding of code's hardness, I show how brainfuck exposes tensions between the human and machine in programming by returning to the theoretical and material structures of the machine as a challenge. In this tension, I argue that brainfuck promises a particular masculinity to those who prove themselves "true programmers" through knowledge of the machine, and I theorize programming itself as a test for being fully human. I use this case to show how rhetorics of machinic neutrality are co-constitutive with ideals of programmer

disembodiment. As brainfuck reveals, our contemporary struggle to connect the effects of our technologies with the people who create them is sustained because their creators perform being machine-like themselves. By interrogating rhetorical mechanisms of disembodiment, we can understand that it's not that the bodies behind the computer happen to be invisible, but that they can be intentionally and rhetorically constructed so.

This work contributes to digital and feminist theories of the expressive, persuasive, and meaning-making potentials and problems between/among humans and machines. I argue that this case calls us to do more than theorize the rhetorics of machinery and the machinery of rhetoric. We also need account for how both approaches can mask bodies and sustain power asymmetries. In doing so, this case demonstrates that one way toward a more embodied digital rhetoric is through interrogating digital *dis*embodiment, a rhetorical construction and performance that is observable in code itself. This focus works to hold embodied power asymmetries in tension with posthuman contexts, objects, and questions, complicating neat human/posthuman divides particularly endemic to digital studies. This contributes more broadly to rhetorical theories of embodiment, especially feminist accounts of rhetoric's "abstract body," by showing that machines need not obscure humanist inquiry, but rather may offer helpful frames.

After situating this project within scholarship on rhetorics and machines, I draw on feminist, queer, and critical race theories of mastery as a gendered and raced force of relationality that makes some fully, and others less-than, human. I argue that such an understanding of mastery—as a difficult, masculine, and material assertion of identity and control—has renewed importance in digital contexts. Through attention to brainfuck's syntax, semantics, and circulation, I explore how composing in brainfuck is an embodied assertion of

mastery, and I read brainfuck back onto the idealized programmer body. Because this disembodied programmer is in fact rendered from normative bodies, I then consider how such supposed digital disembodiment has consequences for how inequality can be sustained in digital contexts.

Rhetoric of Machines, Machinery of Rhetoric

Using rhetorical theory to understand machines has been vital for addressing contemporary possibilities and limits of expression and persuasion. As ecological and posthuman turns have challenged the idea of a tidy rhetorical situation, machines have provided provocative sites for rhetoricians to unpack distributed and complex networks. Such work has often approached the meaning-making potential of machines by applying and nuancing rhetorical concepts to fit new contexts (Brock, Brooke, Gries). The need for such theories of rhetorics among humans and machines has become ever more important as the line between "on" and "offline" has increasingly diminished, and technologies have become part of our lives in ambient, invisible, and increasingly bodied ways (Brown, Boyle, Melonçon). As such, taking the rhetorical potential of machines seriously has been central to digital rhetoric, and this has yielded insights into new media forms, wearable technologies, and machines as rhetorical collaborators (Gouge and Jones, Kennedy).

However, using machines to theorize rhetoric tends to make rhetoricians "squirm" because we are uncomfortable with defining rhetoric as a procedural, rule-based enactment of communication (Brown 496). This unease is rooted in stripping human agency, and thus art and ethics, away from rhetoric and rhetors. Jenny Rice suggests that we can see how complete this

sentiment is in the field's broader distaste for teaching grammar and "mechanics." A similar resistance to bringing machines to bear on rhetorical theory can be seen in the struggle procedural rhetorics has had moving beyond computational contexts, although originally forwarded by Ian Bogost as a theory of rhetoric, not machines. Yet Brown, Rice, and many scholars working from digital and posthumanist frames, encourage theorizing rhetoric with and along machinery because it affords view of a broader networks of rhetorical forces, bodies, and agents. Rhetoric has deeply machinic roots, and this perspective brings the possibility to change how we understand humans, machines, and rhetoric (Boyle, Smith and Brown).

But theorizing rhetoric through/and/as machinery should make us squirm for different reasons. We should be uncomfortable with the ways that raising up the objects of computation risks making the humans creating, using, or feeling their effects invisible. The problem with this is that the presumed, unmarked body is always normative—white, male, cis, hetero, able-bodied, Western, middle or upper class—, and our technologies reflect and constitute these dominant ideologies. As feminist, queer, disability, and racial rhetorical scholars have argued, this abstract, invisible body is a mechanism by which white supremacy and heteropatriarchy quietly and powerfully exert control (Chávez, Dolmage, Flores). The stakes of this invisibility range from mild annoyance to bodily harm and death (Criado-Perez, Nakamura, Noble). To quantify and illustrate these stakes, Carmen Criado-Perez in *The Guardian* traces how non-normative, non-male bodies are at risk because of the invisibility of these normative bodies on which our world in based. There is less research on women's health, and many studies we do presume and use data from a 25-30 year old white "Reference Man." The invisibility of these normative bodies proves particularly fatal when safety devices don't fit or aren't even tested for anyone

other than this abstract, normative body, such as leaving women 17% more likely to die in car crashes.

However, this abstract body is not only a basis for our technologies. It is also central to rhetorical theory (Chávez), and this is made especially apparent in posthumanist digital conversations. What rhetoricians should be uncomfortable with, then, is that the conversations we have about rhetorics and machines risk obscuring those who have yet to achieve the status of fully human. While scholars have argued that the posthuman turn "need not be an *anti*human one," women of color have shown that it is also "not necessarily" a progressive one (Barnett and Boyle 6, Nakamura 134, emphasis original). Feminist and critical race scholars have critiqued posthumanism not because it decenters a human agent, but because human actually means a very particular Man. This is a posthuman presumption that "all subjects have been granted equal access to western humanity and that this is, indeed, what we all want to overcome" (Weheliye 10). For critical race scholars, such a recuperation of the human remains pressing and central work (Browne).

One dehumanizing force that scholars have used to understand how some are granted the status of fully human is mastery. To build an understanding of mastery that focuses on these questions, I turn to postcolonial and queer theory, which consider mastery most often through Hegel's passage on lordship and bondage in *The Phenomenology of Spirit*. In this master/slave dialectic, Hegel theorizes the processes of Recognition, or the emergence self-consciousness. This description illustrates that the other is understood in terms of the self and that the self likewise only exists in being recognized by an other. The meeting of two selves, interpreted as either/both between two people or between two parts of one self-consciousness, causes a

"life-and-death struggle" that establishes one as master and the other as slave (Aching 915). What Hegel establishes for this project is that mastery is centrally concerned with dominance, control, and identity. It is a raced and gendered binary force of relationality that provokes and even necessitates violence to establish the self, and this struggle determines who gets to be "recognized as a *person*" (78, emphasis original).

This concept has been particularly important for scholars of colonialism and slavery, who have contextualized Hegel's dialectic within its specific historical moment, specifically the Haitian Revolution, to better understand slavery not as metaphor but as the material foundational reality for the concept (Aching, Buck-Morss). Julietta Singh's postcolonial accounting of mastery builds on this to demonstrate that mastery, as a term, arises in the early modern period with two meanings: first, as besting an opponent in a competition and second as the attainment of enough competence in a skill or body of knowledge to be qualified to teach it (8). Singh shows how these two meanings are inseparable: "To put it crudely, a colonial master understands his superiority over others by virtue of his ability to have conquered them materially *and* by his insistence on the supremacy of his practices and worldviews over theirs, which renders 'legitimate' the forceful imposition of his worldviews" (9). Thus, even when "mastery" is used to indicate something seemingly harmless or even laudable, such as skill or language mastery, it inevitably invokes mastery's roots in violence, colonization, and slavery. She explores how mastery "invariably and relentlessly reaches toward the indiscriminate control over something—whether human or inhuman, animate or inanimate" (9). Thus, even in cases of self-mastery, Singh insists on understanding this force as always tied up in and acting on gendered and raced histories, powers, and bodies.

Singh's ultimate call to resist mastery echoes similar moves by queer theorists, for whom the resistance of mastery, especially dominating forces of normativity, is an important way to imagine other ways to relate to the world and each other. In one such push toward new social worlds, Jack Halberstam has theorized failure and unmastery as important destabilizing queer forces, exploring how a rejection of mastery, which we always understand through white male heterosexuality, could help us interrogate the limits of some ways of knowing and being. Through "low theory" analysis of popular films like "Dude, Where's My Car?," Halberstam shows how we struggle to see stupidity in straight white men because our understanding of white male heterosexuality is so dependent on the lenses of mastery and intelligence.

I follow Singh and Halberstam to understand mastery as a gendered, sexed, raced, and colonial force of dominance and control, and I argue that these questions have renewed importance in digital contexts because computation raises unique questions about control. Alexander Galloway has theorized control as the central organizing principle of networked technologies, and Wendy Hui Kyong Chun has contextualized software within a larger gendered and military history of "command and control" (29). In such histories and contexts, it's important to consider what role computation plays in the normalization of power differentials. By focusing on mastery, I examine the ways that expressions of mastery and white heteromasculinity can mark the programmer as craftsman, maker, and divine creator of new worlds. In the next section, I read feminist historical scholarship on programming through this lens of mastery to understand how the gendered programmer is marked as less-than fully human, especially through programming's development as a masculine profession. With this understanding of women as "subhuman" in programming, I then analyze code itself to explore how these power divisions are

marked through performances of programming mastery, that can be proved not only in code but also by and on the masterful programmer body.

To be clear, the problem is not in thinking seriously with and through machines. In fact, theorizing rhetoric and writing in terms of systems with rules and procedures has been generative, including for questions of inequality, silencing, and marginalization (Cushman). In the next section, I explore the potential for such machinic inquiry by breaking down brainfuck's structuring logics through a "Hello, World!" program.

Mastering the Machine

Interrogating the name itself is an unmissable beginning for understanding gendered rhetorics about brainfuck. In naming the language, Müller was "trying to express confusion and uselessness." His intentions seemed to have been capturing the humor and awkwardness in meeting a problem that not only exceeds your understanding, but isn't worth grappling with at all. We might also read it as a self-deprecating joke—that Müller would have to be "fucked in the head" to create something so perverse in its obscurity and uselessness. But when we think about brainfuck's name through the lens of masculinities and mastery, we can understand other ways the language circulates. Understood as a competition, brainfuck is a threat of violent, gendered domination. Trying to code in brainfuck is a competition with the machine, and losers end up overwhelmed, mastered, and painfully brainfucked. These meanings have played a large part of brainfuck's fascination and circulation in appealing to programming's rugged masculinities, and this can be seen particularly well in the names of a few other weird languages it has inspired: Agony, HardFuck, and NewbieFuck. However, the names of these programming

languages are not the primary source of their rhetorical force, and looking into the structures of languages themselves reveals further mechanisms of mastery and masculinity at play.

Approaching brainfuck as a programming language turns my attention to the rhetorics with and in the language, including brainfuck's syntax and semantics. Programming syntax indicates the form of languages, including the rules about how symbols can be correctly structured, while semantics is concerned with the procedures a computer executes. Because there is no inherent reason for most symbols to represent any given function, the choices made in a language's syntax also reflect programming beliefs and values. For example, while Java uses a plus sign (+) to concatenate strings, Perl uses a period (.). While these decisions seem arbitrary, diving deeper into such syntactical choices can reveal the ways in which Java emphasizes safety and computational speed versus Perl's focus on minimalism and human speed (Kesteloot). Many of these decisions are made based on what seems clear, easy, and fast, and the use of natural language syntax has been one of the most important developments in programming that has moved coding ever closer to a literacy. The syntax of brainfuck (see table 1) is notable not only for having only eight valid characters but also that they do not include any words, which have become so central in high-level programming, and I'm interested in how this syntax reflects values about programming that can help us understand brainfuck's rhetorical force as a challenge of mastery.

Table 1

Commands in brainfuck

Command	**Description**
>	Move the pointer to the right

<	Move the pointer to the left
+	Increment the memory cell under the pointer by one
-	Decrement the memory cell under the pointer by one
.	Output the character signified by the cell at the pointer
,	Input a character and store it in the cell at the pointer
[	Jump past the matching] if the cell under the point is zero
]	Jump back to the matching [if the cell under the point is nonzero

Source: "Brainfuck," *Esolangs Wiki*, 4 Sept. 2018, esolangs.org/wiki/Brainfuck.

Syntactically, brainfuck removes elements typical of high-level languages that some would call "syntactic sugar." These are commands or operators in a programming language that add ease for the human writer, including having more natural naming or requiring fewer keystrokes, but not additional functionality. These often provide a shorter way for a human to write an already existing function. Powerful high-level languages, thus, often have many ways to solve the same problem, and much of programming style come down to preferences enabled by such "sweetened" syntax. However, without the (often English) words and phrases that make machine communication palatable to humans, brainfuck uses an extremely minimal syntax, and its eight symbols are all primarily mathematical, not "natural" language. While this tension—between difficulty for the human versus the machine—is a frequent trade off in decisions such as syntax, we know from Müller's account that he was aiming for machinic ease, which would enable a small compiler. However, because the syntax is so limited, brainfuck is a

challenge to use in practice, and this syntax's machinic appearance has become a symbol of masterful, exclusive programmer identity. While the language at first sight appears to resist human readers in these syntactical choices, this obfuscation has made brainfuck one way that programmers can mark themselves as true. An example of this can be seen Müller's own shirt during his presentation which read: "`>++++++++[<++++++++>-]<+++++++.--..++++++.`" At the end, he challenged the audience to read his shirt. Only those part of this exclusive community would know that his shirt's code reads "geek." This resistance of human readers is one way that its machinic appearance works to mark a higher class of programmer.

While syntax describes how a program is written, semantics describes how a program behaves, and brainfuck's semantics challenge programmers to work through the logics of the machine. Abstraction from the machine has allowed for the development of higher level programming languages, including those no-code or low-code languages like Scratch that use visual features, such as colored blocks instead of language, meaning that not only syntax but semantics are more abstracted from the machine. This has increasingly meant that a programmer would no longer need to know machine-specific functions, such as how bytes move through memory storage or the location of specific vacuum tubes, in order to compute, and this decreases the training needed and increases the accessibility of digital creation. Brainfuck, however, refuses to blackbox the machine's processes and requires the programmer to return to its design. To understand this aspect of its semantics, I read the "Hello, World!" program in brainfuck that opens this essay to show how the language necessitates an understanding of computational theory and hardware:

++++++++++[>+++++++>++++++++++>+++<<<-]>++.>+.+++++++..+++.>++.<<

+++++++++++++++.>.+++.------.--------.>+.

To do so, I draw on and use images from Fatih Erikli's "brainfuck visualizer," a program that allows for tinkering and experimentation in brainfuck that executes step-by-step alongside a visual model of the stack. Brainfuck is designed on the model of a Turing machine—a theoretical computational model that proved general computability.[6] Like a Turing machine, brainfuck works by moving a pointer left and right on the memory—a stack or tape divided into cells that can each hold a single value of zero or one. The machine then works by moving the pointer on the stack, adding or decreasing values at the pointer, and reading or writing stored values. By stepping back to this architecture, Müller was able to use only eight symbols in his language. However, this doesn't mean that brainfuck is a (theoretically) limited language. In fact, brainfuck, as a Turing machine, is able to theoretically execute the programs of any other computer, including the machines that display this essay.

To begin our "Hello, World!" program, we would first need to print an H. Because brainfuck uses ASCII codes for characters, the program would need to output a value of 72, which is H in ASCII.[7] Simply enough, one way to do this is to add 72 bytes to the first cell on the stack with the "+" operator and output with the "." operator. In brainfuck, this would look like this:

---

[6] Although Turing's name is most popularly attached to artificial intelligence, this references a different computing theory. Being "Turing complete" is a classification of a programming language that indicates it can perform the operations of any other Turing complete language or machine. In weird programming language communities, this designation indicates that the programming language design has achieved a certain level of sophistication.

[7] The American Standard Code for Information Interchange (ASCII) is a system of numeric values assigned to characters, including letters and punctuation, to enable standard communication between computers.

+++

+++.

This could be visualized with a value of 72 in the first cell of the memory stack (see fig. 1).

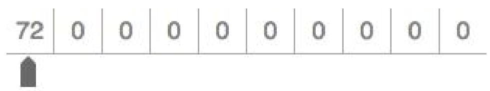


Fig. 1. Erikli, Fatih, et al. "Brainfuck visualizer." *GitHub*, github.com/otomat-hackathon/brainfuck-visualizer. Accessed 19 Oct. 2018.

The pointer, still on the first cell, has incremented to 72, and the period outputs the H character. In this example, it is apparent that there are some aspects of brainfuck that are easy and reflect Müller's original goal. Adding 72 symbols to output a value of 72 is straightforward. However, the inevitable tediousness and inefficiency of this approach is evident, and in this example, I have written a program nearly as long as the full "Hello, World!" but that produces only one of its thirteen characters. To more efficiently create this program, we would need to become better users of the stack's architecture. To show how this works, I break apart the full "Hello, World!" program into three sections alongside descriptions and visualizations of the tape or stack to show how the semantics require the user to work through a mental model of the machine.

Section 1 (operators 1-10): ++++++++++

Fig 2. Erikli, Fatih. "Brainfuck visualizer." *GitHub*, github.com/otomat-hackathon/brainfuck-visualizer. Accessed 19 Oct. 2018.

This first section adds 10 to the first cell and sets up Section 2, which is a loop, or a procedure for executing a set of instructions multiple times. Loops are an important programming tool for decreasing the redundancy, length, and messiness of code. This example sets up a *while* loop, which means that a set of actions will be executed until a terminating condition has been met. In this case, the addition of 10 here means that the set of actions in Section 2, distinguished in brackets, will execute 10 times (see fig. 2).

***Section 2 (11-29)*: [>+++++++>++++++++++>+++<<<-]**

This segment first tells the pointer to begin a loop with the bracket, then to move the second cell with the right arrow, and to add seven with the following seven plus signs. The next arrow moves the pointer to the third cell and adds 10. In the fourth cell, the program adds three. The pointer then moves back three times to the first cell and subtracts one (see fig. 3).

Fig 3. Erikli, Fatih. "Brainfuck visualizer." *GitHub*, github.com/otomat-hackathon/brainfuck-visualizer. Accessed 19 Oct. 2018.

Reaching the closing bracket, the pointer checks its current value, in this case nine. Since this is not zero, the program executes from the beginning again, moving through a total of 10 times until the first cell holds zero, leaving zero, 70, 100, 30, and 10 in each of the cells (see fig. 4). This loop sets up a group of cells that then allows for more efficient output of the letters in the next.

Fig 4. Erikli, Fatih. “Brainfuck visualizer.” *GitHub*, github.com/otomat-hackathon/brainfuck-visualizer. Accessed 19 Oct. 2018.

Section 3 (30-98): **>++.>+.+++++++..+++.>++.<< +++++++++++++++.>.+++.------.--------.>+.**

In this section, the program begins to output the letters “Hello, World!”, and the improved efficiency in this model can be seen in comparison to the first program that printed only “H” with 72 plus signs. Now, the pointer moves right to the second cell, where it then only needs to add two to make the 72 needed to output an “H.” By establishing a wide range of starting points through the loop, the program can now more efficiently target the specific ASCII codes for the needed letters. To print “e” the pointer moves to the third cell and needs to add only one for the necessary 101 that outputs a lowercase “e.” Since a lowercase “l” is 108, the pointer remains on this cell and adds seven, and then prints two “l”s signalled by two periods. The program adds another three to print “o” (111). A space is coded as 32, which means that the most efficient route is to move the pointer to the fourth cell and add two. “World” is then printed in the same fashion by moving among these letters and outputting for the matching ASCII characters. The final character, the exclamation point, is printed by a 33 from the fourth cell , and this leaves the final stack as in fig. 5.

Fig 5. Erikli, Fatih. “Brainfuck visualizer.” *GitHub*, github.com/otomat-hackathon/brainfuck-visualizer. Accessed 19 Oct. 2018.

In the processes of this program, brainfuck poses its challenge through a return to the machine. In contemporary programming, knowledge of a Turing machine or the procedures for storing each byte on the memory stack have become largely unnecessary. Programming languages are capable of abstracting its processes so far from the machine that they can be executed on many different machines, and a programmer may never need to know about the Turing model or stored-program architecture. However, brainfuck doesn't work without understanding how the machine works. This challenge to understand the most inner workings of the machine has contributed to brainfuck's circulation as performances of programming feats, a sampling of which Müller himself marvels at in his presentation. In an era that promotes accessible, easy code with all the stakes of literacy, brainfuck makes code hard again by demanding a return to masculinity, materiality, and mastery. This is a machinic difficulty and intimacy that is achieved through exposing the computer's most inner functionality and becoming united with these processes. In brainfuck, programmers prove their (white, male, heterosexual) humanity by becoming indistinguishable from the machine and demonstrating their control, dominance, and mastery while doing so.

Fully Human, Fully Machine

Such challenges, competitions, and performances of mastery in the *how* of programming can help us understand the *who.* The programmer stereotype—"white, male, middle-class, uncomfortable in his body, and awkward around women"—has so permeated cultural imaginations that Nathan Ensmenger argues it has reached the status of a cliché (41). However, Joseph Weizenbaum, an early artificial intelligence innovator, provides a particularly apt image

of this bodily discomfort and rejection in his 1976 critique of the "compulsive" programmer where such detachment is rendered necessary to the success of mastering the machine and shows how mastery is performed by and on the body.

Weizenbaum draws distinctions between two types of programmers primarily through the relationships between their bodies and machines. For the "professional" programmer, the machine is a useful object that solves problems and makes his job easier. He is almost disinterested in the computer itself and often delegates actual interactions with the machine to more "technical" workers, leaving his body distinct and autonomous from the machine. For the "compulsive" programmer, however, the machine is a source of pleasure from masterful knowledge and control of the machine. This intimate relationship between machines and bodies, and the refusal of the body in order to master the machine, is what Weizenbaum's description captures so well:

> ...bright, young men of disheveled appearance, often with sunken glowing eyes, can be seen sitting at computer consoles, their arms tensed and waiting to fire their fingers, already poised to strike at the buttons and keys on which their attention seems to be as riveted as a gambler's on the rolling dice. … They work until they nearly drop, twenty, thirty hours at a time. Their food, if they arrange it, is brought to them: coffee, Cokes, sandwiches. … Their rumpled clothes, their unwashed and unshaven faces, and their uncombed hair all testify that they are oblivious to their bodies and to the world in which they move. They exist, at least when so engaged, only through and for the computers. (116)

This description might seem commonplace in contemporary stereotypes of programmers, hackers, gamers, and others who work and play with computers and, at first read, might strike us as anything other than masterful. In fact, they may sound like subhumans themselves, with "sunken," "unwashed," and "uncombed" features signalling lack of hygiene, sociality, food, or sleep. However, if we read this through an understanding of how mastery can be expressed in brainfuck's code by becoming indistinguishible from the machine, we can understand this performance in a new way. The labor to make the computer work seems to require the programmer to be focused only on the machine, refusing his own body and becoming machine-like himself—not needing food, sleep, or human interactions. In understanding this as mastery, it's further important to note that it's not that these bodies don't have access to food or sleep. It is instead that they don't need or want it. Such sustenance is of lesser importance than the pleasure of power of and over the computer.

The interesting question, then, is not *are* digital experiences embodied but rather "*how* is performing them as disembodied dangerous?" Brainfuck reveals connections between the ideal of the disembodied programmer with the actual material body and the performances that make it invisible. It shows how digital disembodiment is based on the real bodies of particular men because it shows how the making of invisible bodies is an embodied performance that signals being machine-like. Proving you are fully human, by proving you are fully machine, makes programming appear disembodied. We thus need to consider not only how rhetorics of machinic neutrality influence how we think about computational objects but also how such machinic ideas about the human obfuscate technology's human authors and consequences.

Additionally, these machinic processes that obfuscate white male bodies also construct non-normative bodies as an excess. In contrast to the disembodied programmer, the subhuman is constructed as being so bodied that they can't program because too much body is a hindrance when you are trying to be a machine. One such subhuman emerged as programming professionalized in the 1960s and became more widely useful for a variety of research and business contexts. To make computation accessible, high-level programming languages were developed that transformed the work away from the machinic specificity that built this ideal of coding as difficult, masculine, and solid in its material connection with the machine. These new "high-level languages" use words we recognize and don't rely on deep technological, mechanical, or "tinkering" expertise. This not only made code easier to read but also faster and less error prone. However, these new human-focused programming languages were viewed by some as a deskilling of programming. These concerns were raised with particular force by a consultant at the 1962 RAND conference: "Do we really design languages for use by what we might call professional programmers or are we designing them for use by some sub-human species in order to get around training and having good programmers?" (Ensmenger 83). This imagined subhuman who threatened to fill programming vacancies is particularly well represented in ads from the 1960s, which presented images of women—usually white, blonde, and smiling (Hicks). These ads argue that now—because programming is easier—a young and pretty Susie Meyers is "all the staff you need" (qtd. in Ensmenger 86).

However, programming's non-normative, non-machinic subhumans are not only a matter of history. Such an excess of the body as a marker of subhumanity continues and is especially strongly captured in a derogatory term for less-than programmers: code monkey. While this term

carries mixed meanings, it generally indicates a programmer of lower intellectual ability who isn't capable of producing more than routine code from high-level, blackboxed languages. They aren't familiar with the details of the machine and thus cannot innovate. Visual representations of this programmer are particularly revelatory for showing how this subhuman is marked through an excess of the body. Juxtaposed with a white man in a suit, one illustrates this programmer as a monkey with flailing arms, banging on a keyboard, and knocking over food and papers (Janetakis). Although it is also used jokingly by programmers who enjoy not having management responsibilities, this racializing term is particularly dangerous not only because it is another example of how constructions of disembodiment lend power to normative bodies but also because it is relatively commonplace. In fact, it is so common that O'Reilly Media released their 2016 guide *Programming Beyond Practices* complete with the cover image of a monkey and subtitle "Be more than a code monkey."

As digital rhetoricians wrestle with the increasingly blurred boundaries between human and machine, on and offline, finding ways to work at these intersections of human and posthuman will continue to be a challenge. This chapter shows that interrogating seemingly invisible bodies can be one way to reveal the fleshy excesses not collapsable into the fully "human." Further, it shows that the machinic can be one avenue for uncovering, not obfuscating, those bodies.

Interchapter: brainfucked

++.	H
++.+++++++++++++++++++++++++++.	He
+++[>+++<].	*Error: recursion*
+++[>+++<].	
++++++++++[>+++++++<-]	
++++++++++[>+++++++<-]>++.++++++++++++++++++++++++.++++..+++.--.	Hello,
++++++++++[>+++++++>++++<<-]>++.++++++++++++++++++++++++++.++++..+++.>++.------------.++++++++++++++++++	Hello, W

++.	
++++++++++[>+++++++>++++>+++++++<<-]	*Error: recursion*
++++++++++[>+++++++>++++++++++>+++<<<-]++.	H
++++++++++[>+++++++>++++++++++>+++<<<-]>++.>+.+++++++..+++.>++.<<+++++++++++++++.>.+++.------.--------.>+.	Hello, World!

Chapter 3

Painful Code: Feeling Machines in the Digital Sensorium

After Mark Zuckerberg's first day of congressional testimony about the 2016 election, the internet had a lot of feelings. These feelings were centered around how the simplicity of Congress's questions revealed "just how inscrutable Facebook remains to most Americans" (Lapowsky). Many vented, made jokes, or mocked senators for being out of touch. The simplicity of their questions was compared to "Social Media 101." However, the fear underlying these jokes (and expressed more directly in panicked critiques) centered on the challenges of regulating a technology that so few understand. This feeling was particularly well-captured in a tweet as the frightening realization that "the complexity of our systems has outstripped our abilities to reason about their consequences" (Hinton).

This question—what happens when we can't reason about our technologies—has been of pressing concern in both popular and scholarly contexts. The urgency of these congressional hearings and the rise of the coding literacy movement are especially striking examples of such broad, popular concern for ensuring that more than an exclusive group of highly trained professionals can access computational knowledge. These efforts have been echoed in scholarly calls, particularly in the humanities, to make computation an object of study outside of computer science (Noble, Vee). However, believing that technology is ultimately rational and knowable often underlies debates around these issues that stall. Zuckerberg's frequent defense of Facebook as "just a platform" is only one recent instance of such beliefs (Uberti). This underlying assumption, that technology is predicated on rationality, contributes to our inability to make

substantial changes to how our technologies create and sustain power differences, and it stalls discourse by freeing technology and its creators from responsibility, interrogation, and change.

I am interested in this chapter in a different question about technology and rationality: what are the consequences of thinking technology is inherently rational and can only (or should primarily) be reasoned about? While being *effective* is often the highest virtue in programming, I'm interested in what understanding code as *affective* reveals. Rhetoricians have turned toward theories of affect, feeling, and sensation to better understand meaning-making, especially through ecological models of rhetoric and in posthuman contexts. Code is a particularly interesting place to bring these questions together because discourses of cold rationality continue to shape beliefs about computation's role, responsibility, and effects in our ambiently technological world. To trouble such distinctions between feeling and computational technology, this chapter shows that one way to expand a rhetorics of software to better account for the digital sensorium is through *feeling machines*—approaching computation not only as machines that can be felt or machines that can feel, but also interrelated producers and products of often intense feeling. In other words, this approach calls rhetoricians to attend more broadly to the co-constitutive relationships between feelings and machines, including feeling built and felt in code.

To do this, I focus on experiences of pain in weird programming languages that are intentionally designed to be punishing to the bodymind. After surveying digital rhetoric's approaches and challenges to feelings, I'll go into more historical and contextual detail about why code is such an important site to ask these questions. This chapter then focuses on two case studies. First, I consider how INTERCAL, which is widely accepted to be the first weird

programming language, procedurally enforces politeness to produce feelings of pain. Then, I turn to bodyfuck, a gestural programming language, to show how we can see these gendered dynamics of pain amplified when coding is made an undeniable challenge to the entire bodymind. I argue that pain is not an accidental or unintended consequence of our technologies, but that it is always already baked in. To have a more embodied digital rhetoric, then, we must also pursue a more feeling digital rhetoric.

Feelings and Machines

Although feelings may seem like a relatively recent focus for rhetoricians, Debra Hawhee tells a different story in her 100-year survey of *The Quarterly Journal of Speech*. Throughout the century, Hawhee traces waves and breaks in scholarship on sensation. She calls for rhetoricians to move forward by interrogating how sensation plays a role in meaning making and publics, including in digital contexts. Drawing on media theorist Marshall McLuhan, she defines the sensorium as "a locus of feeling, and yet that locus is not confined to presumed bodily boundaries, especially when technology is considered" (5). This includes not only the body and its senses but also material and technological ecologies. Hawhee further notes that the shift of "digital lives toward the social" in platforms such as Facebook, Snapchat, and Instagram "calls for a more robust theory of the sensorium" (12).[8]

Though the digital thus seems an ideal and important site for theorizing sensation, digital rhetoric has been challenged to address such embodied questions (Eyman, Noble). While rhetoricians working at the intersections of feeling and digitality have been largely interested in

[8] In thinking about such digital contexts, it is also worth noticing that Hawhee finds the language of "energy" and "circuitry" used throughout rhetorical scholarship to understand sensation.

affective discourse that happens on or through digital spaces, digital rhetoric as a whole has been less attentive to feeling aspects and consequences of computation. Important advances in these conversations include understanding that *machines can be felt*. They are material, part of our ambient environment, and ever closer to and part of our bodies (Gouge and Jones). Additionally, Liz Losh has pushed for a more capacious approach to machinic rhetorical force by exploring the ways that *machines can feel*. Examining how machines themselves are sensing entities, Losh challenges rhetoricians to better account for "the sensorium of the machine" by moving away from human-centered interfaces and objects, and she examines sensors (over optics) to show the value of this approach.

But, to better understand the embodied, feeling aspects and consequences of digital rhetorics, we also need to approach machines and feelings as co-constitutive. As I argue in chapter 2, rhetoricians have been especially challenged to address such questions because of rhetoric's long-standing discomfort with the machinic. In fact, machines and feelings have often been imagined as opposites. Hawhee's survey includes a note about a 1915 *QJS* article on the "mechanical" nature of failed emotional oratory (8). However, like rhetoric itself, computation is often imagined to be primarily concerned with Western and masculine values of reason and rationality. Sensation and feeling have been one way rhetoric has worked to move past these limits, and this focus also affords a way forward for digital rhetoric.

In particular, feelings of pain draw our attention back to the surfaces of our bodies, making it especially useful for interrogating feeling when we think there isn't any (Ahmed). This makes pain particularly important for understanding our digital sensorium because programming has been upheld as cold, rational, and unfeeling through the persistence of its military history in

our ideas about what code is and does (Goffey). This history has meant that programming has carried "an extreme understanding of technology as a utilitarian tool predicated on a reframing of the world as a set of calculable quantities" (21). Code, then, is primarily useful, efficient, and productive. Wendy Hui Kyong Chun has theorized this connection between militarization, unfeeling rationality, and gender by examining code *as logos*. Programming is "command and control." In computation, there is no difference between speaking and execution. While scholars of this military history push on its tensions to explore new genealogies, I want to linger with this history to understand how these values and ideals contribute to continued perception of computational arhetoricity.

Programming's most unquestioned tenet is that good code *works*. This particularly well expressed in the contributions from 36 programmers in *Beautiful Code: Leading Programmers Explain How They Think*. Good code solves problems, executes (ideally quickly), and produces a useful solution. This is a central belief, spoken or unspoken, in each of chapter, and it is tied to understanding code as a "right" or "wrong" tool (353). Beautiful code, then, is that which is legible to the machine and can be executed with productive results. This binary understanding of code seems to completely separate it from feeling. However, we can complicate this separation even by looking more closely at this emphasis on functioning because productive code is linked to the programmer's experience of working with it: "I find beauty in code that I can trust—code that I am confident will produce results that are correct and applicable to my problem… I can use and reuse without any shred of doubt in the code's ability to deliver results" (254). Although trying to express code's rationality and utility, this programmer (and I) are already, inescapably slipping into feeling. Effective code can inspire feelings of confidence, trust, and closeness that

produces "results." Although a utilitarian theory of code would seem to limit feeling, utility still shapes a particular relationship between human and machine that moves us into feeling.

The potential feelings produced from computation have been particularly well captured by Joseph Weizenbaum's delineation between "professional" and "compulsive" programmers, which I unpack with more detail in Chapter 2. The professional programmer sees the computer as a tool. He keeps his distance from the machine, both physical and emotional. In fact, he often delegates the actual computer work to employees. His only interest is in utility, and the only appropriate feeling is "satisfaction" from rationality and perfect order. The compulsive programmer, however, pursues computation *because* of the feelings coding produces. His work is obsessive, "feverish," and "a thrill." In fact, Weizenbaum ties *all* of this programmer's feelings to the computer. If the computer is taken away, the programmer will become "depressed, begin to sulk," and the only way to change those feelings is "a new opportunity to compute."

In this critique of the compulsive programmer, Weizenbaum depicts a co-constitutive relationship between ineffective code and feeling. More specifically, he shows how computational feeling is a deviant pleasure, a perversion that produces ineffective code. In other words, bad feelings produce bad code—chasing "a thrill" produces systems built without a big-picture plan, which inevitably fail and are more difficult and inefficient to fix. And bad coding also leads to bad feelings—chasing an error brings the programmer's feelings "to its highest, most feverish pitch," and he begins taking risks, which often leads to destroying "weeks and weeks of his own work." It seems that effective and affective coding are irreconcilable. If you feel, you're doing it wrong.

However, if we start from feeling—not as a corruption of programming but as a central experience and organizing principle—we can see how feeling is always at play in code and coding. In the case studies that follow, these weird programming languages expose programming norms and values through their exaggeration. These exaggerations, especially of programming's supposedly rational perfect order, make visible the gendered, embodied, feeling, even erotic, aspects of programming and challenge programming's supposed unfeeling utility.

PLEASE DO GIVE UP: Submitting to INTERCAL

"The Compiler Language With No Pronounceable Acronym" is, for reasons both obvious and unclear, more commonly called "INTERCAL." Widely recognized as the first weird programming language, it was designed by in 1972 by two Princeton students for the purpose of being unlike any other programming language then in existence, such FORTRAN, COBOL, and APL. The language was updated and rebooted in 1990 as C-INTERCAL,[9] an implementation written in C that expanded the language's abilities. This created what its designer, Eric S. Raymond, describes as a 1990s take on a 1970s language that parodied 1960s languages. In each decade, the language circulates as an intricate and technical esoteric joke. Matthew Mateas and Nick Montfort focus on INTERCAL as an in-group joke to explain its unfamiliar syntax, counterintuitive structures, and puzzling operators. The parody continues and becomes undeniable in INTERCAL's accompanying manual,[10] which takes the form of a scientific white

[9] The original version of INTERCAL, now designated INTERCAL-72, was designed to use punched cards. Because C-INTERCAL compilers, programs, and forums are more widely available and standardized, in addition to having expanded features, I focus my analysis of code on C-INTERCAL. I use "C-INTERCAL" to refer to specific updates unique to this version, "INTERCAL-72" to refer to a feature that only is part of the original implementation, and "INTERCAL" to refer more broadly to the language as a whole throughout implementations.
[10] C-INTERCAL included and updated INTERCAL-72's manual, only making additions.

paper, yet contains drawings of nonsense circuitry and whole pages of blank space in the place of solutions.

However, writing in INTERCAL is a significantly less lighthearted experience. Read through feeling, INTERCAL is domineering, punishing, and painful. In fact, Raymond has noted that even he isn't "twisted enough" to *use* the language (Tempkin). As looking at INTERCAL's syntax, structures, and manual shows, what makes INTERCAL too "twisted" for Raymond is that it uses obfuscation, puzzle, and difficulty to produce distinctly gendered feelings of pain. More specifically, INTERCAL procedurally enforces gendered norms of politeness and perfection to place the coder in a feminized position. To explore these gendered dynamics of pain, I'll begin with the "Hello, World!" program in INTERCAL below, with line numbers added for clarity:

```
1     DO ,1 <- #13
2     PLEASE DO ,1 SUB #1 <- #238
3     DO ,1 SUB #2 <- #108
4     DO ,1 SUB #3 <- #112
5     DO ,1 SUB #4 <- #0
6     DO ,1 SUB #5 <- #64
7     DO ,1 SUB #6 <- #194
8     PLEASE DO ,1 SUB #7 <- #48
9     DO ,1 SUB #8 <- #26
10    DO ,1 SUB #9 <- #244
11    PLEASE DO ,1 SUB #10 <- #168
```

```
12      DO ,1 SUB #11 <- #24
13      DO ,1 SUB #12 <- #16
14      DO ,1 SUB #13 <- #162
15      PLEASE READ OUT ,1
16      PLEASE GIVE UP
```

This may not seem, at first, to be a particularly painful coding experience. For non-coders, it may feel just as obfuscated as a contemporary high level programming language, like Python or Java. For coders, it might actually seem recognizable because of its stylistic similarities to assembly languages, such as FORTRAN and COBOL, that INTERCAL parodied. However, a closer read—or worse, an attempt to write—reveals that this language is counterintuitive and punishing to the bodymind.

Before walking through how this program works, it's important to note that the point of doing this is to show that INTERCAL is intentionally counterintuitive. Even for non-coders, this language doesn't do what we might popularly understand about how computation works. Unsurprisingly, INTERCAL uses an architectural design unique to itself. Called a "Turing text model," INTERCAL's architecture is derived from a Turing machine, which is used by a programming language called "brainfuck" that I discuss chapter 2. This "text model" changes the Turing machine by making the infinite tape into a finite, continuous loop. Compared to brainfuck, this means that the "stack" is not a strip but a connected loop that contains 256 spaces, one for each character the machine can output. Unlike brainfuck, where you add values into any position on the stack to output a character, such as adding 72 to output the letter "H", this model

requires you to move the pointer to a specific location on the loop to output its corresponding character.

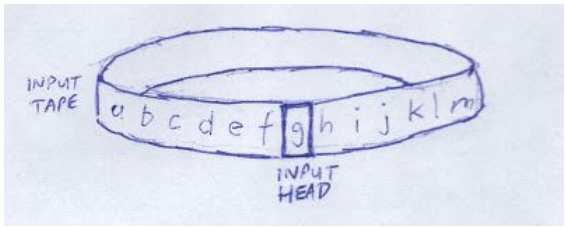


Fig. 1 and 2: On the left, an example Turing machine tape (as in brianfuck), which extends infinitely on both sides. On the right, a sketch of a Turing text model (INTERCAL), which is a connected loop with 256 cells, each containing one character (Source: Forbes, "Diving into Intercal").

Looking back to this specific "Hello, World!" example, we can see how this architecture poses its challenge. To begin, every line in INTERCAL must begin with a statement identifier: DO, PLEASE, or PLEASE DO. The manual notes that "these may be used interchangeably to improve the aesthetics of the program." In our "Hello, World!" program, we can see the use of all three. Line 1 uses a "DO" identifier, followed by a space, comma, and 1, which creates a 16-bit array named ",1." An array is a collection of objects that are all the same size and type, which is useful because operations can then be performed on all objects in the array at the same time. Line 1 then continues with a less-than sign and a hyphen which makes the assignment of the following "#13" the first value to array ",1". The hash indicates that 13 is a constant, which tells the program not to alter it during execution. Lines 2-14 each add a value to this array through this same assignment function. A more contemporary programming language, such as Python, would likely allow for the formation of this array with a single line of code: "array1 = [13, 238, 108, 112, 0, 64, 194, 48, 26, 244, 168, 24, 16, 162]".

Now that the array has been set, operations can be performed on all values at the same time. In this case, the program will print out character to spell out "Hello, World!" However, unlike the Turing machine in Chapter 2, these values do not simply correspond to the intended letters' ASCII equivalent. Instead, they indicate the number of times the pointer should move forward on the looped stack. Thus, when line 15 "prints" out the characters array named ",1" through the "READ OUT" command, the program begins by moving 13 times on the loop to print out a carriage return, which tells the program to output on a new line. The pointer then moves to the second value in the array and moves 238 steps forward on the loop, continuing through the remaining values, always moving forward. Thus, instead of being ASCII equivalents, the numbers in our array indicate the number of moves around the looped stack the pointer needs to move. In other words, "236" has nothing to do with the letter "H," as it does in ASCII. Instead, it is the difference between two numbers, not the actual numbers needed.[11] Finally, Line 16, "PLEASE GIVE UP", terminates the program.

If that description wasn't confusing enough, explaining the program through the language of INTERCAL produces new challenges. In its effort to be unlike any other programming language, INTERCAL also renames the symbols it uses. For example, instead of explaining, as above, that line 1 identifies an array using a comma and assigns the value of 13 to that array using the less-than sign and hyphen, I would say: "line 1 identifies an array using a *tail* and assigns the value of 13 using an *angle-worm*." These name changes might seem like an obnoxious, but ultimately not too complicated, feature. However, looking into the manual's

[11] However, outputting the characters in our program is made even more complicated by the fact that INTERCAL's tape loop has both an "inner" and an "outer" ring, or an "input" tape and an "output" tape. This results in a complicated relationship between the pointer position and the desired output character that I won't dive into here. For more details on this unique architecture, see INTERCAL's manual or Clinton Forbes's blog "Diving into Intercal."

explanation of more complex features of INTERCAL shows how this quickly accumulates in difficulty and blocks understanding. For example, the manual explains that unary operators should be “inserted between the spot, two-spot, mesh, or what-have-you, and the integer.” This is nearly meaningless without translating from the manual’s character table (which the authors note includes characters that are not part of the language).

These confusing naming conventions also extend to INTERCAL’s data types. For example, the manual explains that “.123, :123, #123, ,123, and ;123 are all distinct.” Representing a 16-bit variable, 32-bit variable, constant, 16-bit array, and a hybrid array, this series of values are all named “123” but they represent types that are not interchangeable. Yet, they are only distinguished from each other by similar-looking punctuation. This greatly increases the possibility for error and program failure, which the manual makes exceeding clear by pointing out that ’#165¢#203’~#358 has a value of 15, #165¢’#203~#358’ has a value of 34815, and yet #165¢#203~#358 is invalid. All of this matters because it makes INTERCAL obfuscated, difficult to use, and highly error prone. While many weird programming languages are interested in difficulty and challenge, INTERCAL uses that difficulty for the explicit and distinct purpose of inflicting pain. As looking into other features of the language shows, INTERCAL is not only difficult to work in. It also punishes you for failing. More specifically, INTERCAL procedurally demands politeness and perfection from programmers, and this enforcement happens through hidden procedures, mocking error messages, and failing programs.

To see how this happens, we need to reconsider the operators DO, PLEASE, and PLEASE DO in our “Hello, World!” program. While it is true that all of these ask the computer to perform the same action, this choice is not purely aesthetic but also procedural. If we had

written this program using too many "DO"s, it would throw an error message: "PROGRAMMER IS INSUFFICIENTLY POLITE." However, if we had used too many "PLEASE"s, the program would also be rejected: "PROGRAMMER IS OVERLY POLITE." This feature—a politeness requirement—is INTERCAL's most unique procedure. It is also unrelated to any other part of the program: "It serves no other purpose than to make the programmer polite, but not too polite" ("INTERCAL"). Including politeness as a central feature of INTERCAL creates a particular human-machine relationship that is not based around cold rationality but instead embodied feeling. Politeness is a classed, raced, and gendered expression that includes a wide range of tones, behaviors, and feelings (Mills). It is shaped by cultural and social norms that dictate the appropriate way for certain bodies, usually upper class white women, to interact with others. One important function of politeness, especially when thinking about the power imbalances therein, is to avoid disagreement, humiliation, threats, and even violence.[12]

In this politeness requirement, INTERCAL further dominates the programmer because it procedurally disrupts ideals of code as logos, the conflation of command and control, word and action. An INTERCAL programmer has to *ask* this program to run through expressing the appropriate level of politeness. They are not in a position of control. The computer makes the decision. Thus, programmers have to behave in a way that they hope avoids being shamed. Furthermore, because this feature was undocumented in INTERCAL-72's original manual, pleasing the program was even more difficult. Though we now know that the "golden rule" is to use PLEASE modifiers on 1/3rd to 1/5th of your lines, the only way to figure this in 1972 was to

[12] One recent public campaign around gender, politeness, and harm has been "Stop Telling Women to Smile." This campaign included documenting a range of aggressions received for not expressing the smile that is required by politeness norms when being a woman in public.

try, be shamed, try again, and be shamed again. And INTERCAL doesn't respond well to being bossed around, commanded or controlled.

Being insufficiently polite is not the only way INTERCAL inflicts painful gendered feelings. The language also enforces perfection, and in doing so, exaggerates ideals of objectively rational "correct" code. In fact, its demand of absolute perfection inhibits effectiveness. Like the politeness requirement, INTERCAL uses many other error messages to correct the coder, and these vary in their helpfulness. Usually, error message are an important tool for debugging code, and they are short, to the point, descriptions of what type and/or where an error was found. For example, if we were to assign a value to a constant variable that was too large for its data type, such as assigning a number greater than 65,535 to a data type called a "short," the language C++ would display: "warning: integer constant too large for 'short' type." This message explains exactly why the code won't execute. We could then change the code to either use a value less than 65,535 or change the data type of that variable to an "integer" or a "long," which can hold values above 2 million. However, if we were to make the same mistake in INTERCAL, it would reply: "DO YOU EXPECT ME TO FIGURE THIS OUT?" There is no suggestion or direction on how to debug the program. Instead, INTERCAL mocks its user. It is also interesting that the error doesn't express a failure of the program but something closer to a lack of motivation, engagement, or respect for the programmer. INTERCAL's other error messages largely follow this style. They are insulting and belittling. Many address the programmer directly as "YOU" to establish the deficiencies in the coder and the dominance of the computer.[13]

[13] Error messages in this vein include, "EXCUSE ME, YOU MUST HAVE CONFUSED ME WITH SOME OTHER COMPILER" (E998); "HOW DARE YOU INSULT ME!" (E652); and "YOU HAVE TOO MUCH ROPE TO HANG YOURSELF" (E991).

It is further interesting to note that not all of INTERCAL's error messages actually indicate a programmer's error. One error is thrown randomly, even when there are not actually errors in the code. A few others can be a result of INTERCAL's own limitations, not programmer error, and yet these error messages still deflect blame to the programmer. For example, the error message "YOU CAN'T HAVE EVERYTHING, WHERE WOULD YOU PUT IT?" occurs when the compiler is unable to account for the number of variables used in the program. Yet, it points demands that the programmer "CORRECT SOURCE AND RESUBMIT." INTERCAL also uses warnings to exert this pressure before the program is finished. It can interrupt with "DON'T TYPE THAT SO HASTILY" if the programmer is typing too quickly. Criticism, then, happens throughout the writing process.[14] As the updated manual notes, these warnings are quite sensitive "to make sure that they aren't useful." Not only are error messages not helpful and blamed on programmer failures no matter the cause, INTERCAL also lashes out for being *too good*.

One of the easiest errors to make in INTERCAL is around the input and output of numbers. INTERCAL only accepts numbers when they are spelled out. For example, if I want to assign a value of 65,536, I couldn't type the numerals "65,536." Valid entry would be "SIX FIVE FIVE THREE SIX."[15] Furthermore, if you meet this challenge, INTERCAL also then doesn't output numbers in a format that is easy to work with. INTERCAL-72 only output Roman numerals, though INTERCAL's 1990 update added the option to change its input/output to numerals. This option is called WIMP mode. When we talk about WIMP modes, we usually

[14] Another warning, "YOU CAN'T EXPECT ME TO CHECK BACK THAT FAR" (W276) also appears before compile-time if the program is unsure the optimizer can guarantee there won't be an overflow.
[15] INTERCAL also accepts "OH" for zero and "NINER" for nine, "to accommodate pilots." It also recognizes numbers written in "Sanskrit, Basque, Tagalog, Classical Nahuatl, Georgian, Kwakiutl, and Volapuk."

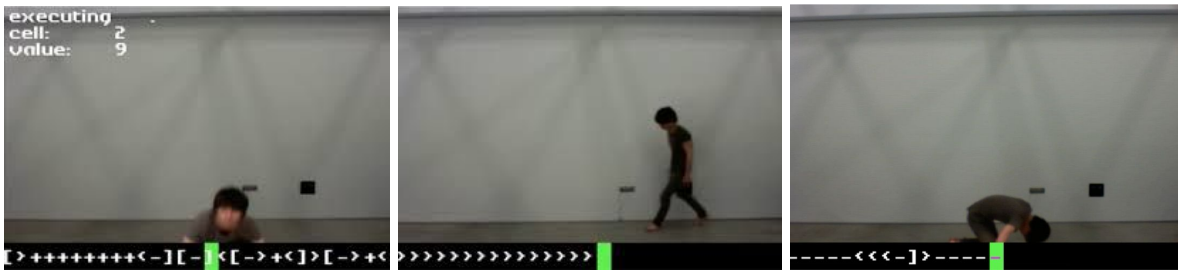
refer to features of the user interface—Windows, Icons, Mice, Pointer—that are now commonplace in our technologies. However, INTERCAL's WIMP mode has nothing to do with the graphical interface or the features this acronym represents. It doesn't meant that you can write INTERCAL on a nicely organized desktop instead of at the command line. It means that its difficulty has been lowered. And because WIMP here doesn't actually refer to any of the features it stands for, the choice to call this option WIMP mode highlights its gendered and ableist message. In other words, while "WIMP" usually describes the computer, in INTERCAL, it describes the programmer. We can see this in the message INTERCAL displays if WIMP mode is selected upon beginning: "How sad… you have selected to run an INTERCAL program in WIMP MODE." The message then briefly explains this version's rules for number input/output, and ends "You are a WIMP!" As this feature's creator explained, the punishment for choosing the "less painful" version is being "subjected to a humiliating message about what a wimp" they are.

To return to Raymond's remark that he was not "twisted enough" to use INTERCAL himself, we can see the fullness of what he is not taking on—the everyday, mundane rules for being a woman and the feelings therein. INTERCAL's procedures require the programmer to jump through the everyday gendered hoops of politeness and perfection that men not only find unfamiliar but "twisted," and the experience of them is painful, shameful, and exhausting.

"The computer didn't get sore. I did": Getting bodyfucked

Like INTERCAL, the gestural language "bodyfuck" also troubles the belief that computation is inherently separate from feeling. If INTERCAL procedurally enforces politeness

and perfection, bodyfuck also moves where feeling and perfection are located to the programmer's entire body. Created by Nik Hanselmann as part of this 2009 MFA thesis, bodyfuck is an implementation of brainfuck that accepts input through a web camera, like a Kinect.[16] While brainfuck has inspired over 250 languages or derivatives, bodyfuck is unique among these because it turns brainfuck into a gestural language. Instead of typing, programming happens through whole-body movements, such as jumping, ducking, and even crawling in front of a web camera. Brainfuck's difficulty, which I explore in Chapter 2, is compounded by bodyfuck's extension to gesture. *WIRED Magazine* captured this difference between brainfuck and bodyfuck in comparion to a quite unusual writing challenge: "Coding in Brainfuck is like writing *War and Peace* using a rotary telephone … using bodyfuck to write Brainfuck code is as if the rotary phone is hooked up to a game of *Dance Dance Revolution*" (Moynihan).



Images from Hanselmann's performances of bodyfuck

Although I have used "Hello, World!" programs throughout this project to explore how a language works, bodyfuck is so difficult that I will instead look at a program that prints only the word "HI". In brainfuck, we could print "HI" like this: "++++++++++[>++++++++<-]++.+." In bodyfuck, this same program requires a full minute of ducking and jumping—up, right, left. Not

[16] When Hanselmann created bodyfuck, technologies like the Kinect weren't available. Hanselmann has remarked that this would have undoubtedly "made certain parts easier … But that would be kind of missing the point, wouldn't it?"

only do these movements need to be in the right order, but to write the brackets in the original program, you also need to pause and hold still for the right amount of time. In Hanselmann's recorded performance of "HI," we can see how bodyfuck is punishing. Hanselmann uses required pauses to catch his breath, which grows more rapid throughout. We can hear the impact of landing and his clothes rustling against his body. We can see feeling—focus, exhaustion, intensity, and finally, relief and pleasure when the program successfully executes.

Bodyfuck collapses the (supposed) distance between a feeling body and computation. It further disrupts an ideal of perfect rationality by pushing it to its extreme or absurdity. Like INTERCAL, bodyfuck also demands perfection, but unlike INTERCAL, an additional challenge is that there are no error warnings that (though usually unhelpful) redirect you and allow for revision. In bodyfuck, there is no way to delete a mistake. Instead, you can only clap twice to reset the program and start all over. For his exhibit, Hanselmann captured his performances of several programs, and to get these, Hanselmann of course failed—a lot. And this was punishing to his bodymind. He had to "sweat and bleed" to code, his body was sore from so many attempts, and he often experienced "despair" before reaching the joy of a correct program.

"Leather-clad and spike-heeled": Submitting to INTERCAL

In this chapter, I've argued that weird programming languages like INTERCAL and bodyfuck blur boundaries between bodies, feeling, and code. Feelings of pain, which have been connected to exhaustion, defeat, punishment, and even pleasure, are been produced through the code's syntax, semantics, and procedures. In this final section, I want to draw together some threads that have been happening throughout these cases but that I haven't dug into yet. These

feelings of pain have been complexly related to pleasure, and they can help better understand the feelings we *do* allow computation in popular discourse. To return to Weizenbaum, pleasure in coding is deviant. It is a pleasure from the pain of being dominated. INTERCAL is "perverse" or, as many users as well as the updated manual puts it, "technomasochism." INTERCAL's manual, in a true 1990s hallmark, includes a note on Y2K that speaks to this experience: "INTERCAL is fully Y2K-compliant. Indeed, it is Y2K-obsequious, and loves nothing better than to be punished by relays of leather-clad and spike-heeled calendricists. Bite me and go buy your own shotgun." Because of these readings, we can better understand how the language of my final case study exists at all:

```
Lick Trisha's feet ten times
Make Clara moan
        Until Amy is dominant toward Alicia
        Have Mistress torture Betty
        If Betty is Tricia's little pet
                Call safeword
        Have Betty hogtie Clara
Make slave scream Clara's name
```

Although I don't want to dig into the procedures of how this language works, I do want to point out that this program, written in FetLang (portmanteau for "Fetish Language"), is fully executable (i.e coldly rational) code. What I am interested in is how the previous cases can help us understand and contextualize a language like this. If I had started this chapter with this language, it might seem like an aberration. It would look like a cherry-picked, very sexy hook

that isn't representative of larger programming languages, practices, or feelings. Instead, INTERCAL and bodyfuck show that a fetish programming language actually makes sense as an exploration of programming, pain, and pleasure.

To return to the story I began with, Facebook's "just a platform" defense only works if code's unfeeling rationality is unquestioned. To counter this argument, and how it forecloses change, we need to be able to question code's arhetoricity. As these case studies show, we need to develop theories of feeling and machines that accounts for their co-constitutive relationship in order to participate in and understand our digital sensorium. Although there are many places to focus such an inquiry, these cases show that code needs to be an important part of the conversation. Furthermore, these cases show that a focus on pain is an important feeling to account for. We tend to talk about our technologies when they hurt us and when they cause us pain. However, if we believe they are rational perfect order, this pain can't be legitimized—If you are harmed by technology, it's your fault. But, if we can see programming as an inherently feeling relationship with the machine, we might be able to better acknowledge and address the very real harm that technology can cause.

Interchapter: Knit & Perl

Single Rib Stitch Square

```
row = 0
column = 0

CO 20                                   // cast on

while row < 20
        while column <20
                k1
                p1
                column = column + 2
        row = row + 1                   // increase row number
        TW                              // turn work
else
        END                             // bind off
```

FizzBuzz Scarf

```
row = 0
column = 0

CO 20                                                   // cast on

while row < 100
        if (row%3 === 0)                                // numbers divisible by 3
                k3, p2, k2, p6, k2, p2, k3

        else if (row%5 ===0)                            // numbers divisible by 5
                k5, p2, k6, p2, k5

        else if (row%3 === 0 && row%5 === 0)            // numbers divisible by 3 and 5
                k5, p2, sl3, ksts 3, k3, p2, k5

        else                                            // numbers not divisible by 3 or 5
                k20

        row = row + 1                                   // increase row number
        TW                                              // turn work
else
        END                                             // bind off
```

Chapter 4

Broken Code: Space, Silencing, and Digital Infrastructures

That misogyny permeates our digital spaces has been firmly established by feminist media scholars (Jane, Mantilla, Phillips). This scholarly attention has been echoed in popular conversations about high-profile events, most notably GamerGate, that have produced material and embodied danger to non-male, non-normative identities. While popular and scholarly conversations have often focused on events like GamerGate, this special issue is evidence of growing feminist efforts to better understand online misogyny's various forms and forces. In this article, I argue that code is an important site for such digital feminist inquiry. Examining code itself, and not only the interfaces and environments produced by it, provides an opportunity to more fully understand the everyday realities of misogyny in digital contexts.

To explore this potential of code as a feminist object of study, I examine an anti-feminist programming language hoax: C+= (pronounced "C plus equality" or "see equality"). This case study demonstrates how claims to space are a persistent and deep-running source of misogyny online. This "digital manspreading"—a concept I develop through this case—happens because online interactions are unavoidably embodied, material, and spatial. In the same ways that a seemingly benign, wide-legged body position reinscribes power differentials in physical space, so do acts of spatial domination online. Although C+= is a relatively tame example of online harassment, by reading this case as digital manspreading, I hope to capture the everydayness of sexism online that, just as in physical space, includes actions that stem from complex raced, gendered, and classed privileges to reinforce difference. Ultimately, I use C+= and digital

manspreading to argue that feminist scholarship must move beyond the interface to interrogating how digital infrastructures, and code in particular, participate in misogyny.

Case Study

In December 2013, graduate student Arielle Schlesinger posted a blog on the Humanities, Arts, Science, and Technology Alliance and Collaboratory (HASTAC)[17] website about her latest research question: "What is a feminist programming language?" The next day, the Feminist Software Foundation provided a resounding answer. In a play on C++, they created the programming language C+=. This work was shared on discussion boards and social media as "a feminist programming language, created to smash the toxic Patriarchy that is inherent in and that permeates all current computer programming languages."

Their intentions, however, were just the opposite. Although many initially believed C+= was a genuinely feminist project, research from news sites and users on harassment forums revealed that this was the collaborative, misogynist effort of users in the Technology (/g/) and Politically Incorrect (/pol/) forums on 4chan, an anonymous imageboard website notorious for producing offensive material. Although the code purported to explore feminist principles in programming, the authors were actually mocking feminism and harassing Schlesinger. Beginning in the /g/ forum, commenters took up Schlesinger's post, often expressing outright hostility: "There's already other programming languages. Why don't you just learn those? You women always make things complicated and stupids!!! FUCK YOU!" ("4chan").

[17] HASTAC is a large academic network, and this is important for contextualising this hoax. That is, C+= should not be read as ridiculing one graduate student's idea. Instead, it more broadly and indirectly attacks feminist inquiry in digital humanities.

Sexist remarks like these culminated in the development of C+=, an anti-feminist programming language and harassment campaign. Users[18] from /g/ and /pol/ collaboratively developed six sample programs, logos, websites, a license, and a README file, which included a manifesto, 13-point philosophy, and direct credit to "the ground-breaking feminist research of Arielle Schlesinger." However, because of its attacks on feminism broadly and Schlesinger specifically—including trivializations of rape, trigger warnings, and transgender identities—the code was taken down from several other sites within days, which only fueled further involvement from misogynist users.

Space, Gender, and Code

To read this case study through a lens of digital manspreading, I draw on feminist theory and rhetorical studies. Feminist scholarship in architecture has provided rich analysis of how built environments, or "man-made" spaces, affect the kinds of access and power people have within them (Weisman). Such gendered division of space is born out in a trope so familiar that Roxanne Mountford suggests we have forgotten its spatiality: "a woman's place" (52). Feminist scholars have further connected these issues with the body, showing how gender can be performed through spatial behaviors (Young). One stereotypical embodied expression of masculinity is expanding to take up space while a feminine expression would be taking up less, particularly to avoid violence (Acaron 144). Brian Pronger has theorized this embodied expression as the entire point of masculinity: territorialization that works "to become larger, to take up more space, and yield less of it" (72). This scholarship shows how embodied expressions of gender in physical

[18] Because of 4chan's anonymity, one of the site's primary affordances to harassment and other offenses, I refer to the unidentifiable authors of C+= only as "authors" or "users." Additionally, because the original GitHub repository was removed, the number of collaborators is also unclear.

space stem from and constitute gendered power differentials that have can have social and material consequences. However, these theories have been posed with challenges when looking to digital contexts because the internet continues to be conceptualized as ephemeral and disembodied.

These concerns with bodies, gender, power, and space have come to popular attention recently through the portmanteau "manspreading," referring to the tendency of men to take up more space than needed on public transportation, specifically by spreading their legs into a wide V shape across two or three seats. Although this behavior was discussed online as early as 2008, manspreading became popular after New York City's 2014 MTA campaign asking men to "stop the spread" (Martin). In her genealogy of the term, Emma A. Jane explains that manspreading is "a powerful—yet also ridiculous—symbol" of this stereotypical masculine claim to both physical and social space (2). It is an act of entitlement, privilege, and "toxic masculinity" that symbolizes and produces gendered power differentials (4). I build on this research to explore how manspreading might also help us understand misogyny online.

Although paradoxically seeming to shrink, blur, or eliminate space, the internet is a built, man-made structure both physically as well as metaphorically as "cyberspace" (Frederick, Osenga). Scholars in human geography have even explored how we can understand that software, the most vapory of digital infrastructures, is social and material through its mutually constitutive relationship with space (Kitchin and Dodge). Digital manspreading builds on this scholarship to help demonstrate how claims to online space, made through the affordances of digital infrastructures, are gendered, material, and embodied. It responds to calls for digital scholars to more fully attend to materiality and embodiment (Eyman, Noble). This is important

because when embodied power differentials are overlooked or misunderstood, they can be sustained—even automated—and blamed on a supposedly immaterial, bodiless machine (Chun). When we say the "computer did it," we really mean that the differences being produced by algorithms and other infrastructures are natural and normal (Vee 37, Noble).

My approach to these concerns is rhetorical, meaning that I focus on how expression and persuasion occur, especially through discourse. In particular my methods are rooted in feminist and procedural rhetorics to identify meaning-making forces about, around, and by non-male, non-normative identities. While a cornerstone of feminist rhetorical work sought to recover women's voices, feminist rhetoricians also now identify contemporary forms of gendered silencing through attention to a broader network or ecology of actants, including space and time (Hallenbeck, Jack). In such complex rhetorical ecologies, digital rhetoricians highlight how digital objects are actants and interlocutors (Brown). Procedural rhetorics further attunes these concerns to digital infrastructures' rules and systems (Bogost).

These foci lead me to identify the online discussions about C+=, as well as the code itself, as important rhetorical resources of this case study. This approach is grounded in understanding that code *is* writing—it has authors, contexts, purposes, and audiences, both human and machine (Marino, Vee). A critical and rhetorical approach to code attends to its human and machinic readers to understand the aesthetic, expressive, and persuasive possibilities of both language and execution, text and procedure (Bogost, Brock). Programming is language that both *says* and *does*, and the rules that it enacts have a particular rhetorical force that digital scholars must consider (Chun).

These tensions—between human and machine, saying and doing—are perhaps best evidenced by esoteric of "weird" programming languages like C+=. Weird programming languages are creative works that test the boundaries of programming itself and are designed as parody, proof of concept, software art, codework, or for play/sport (Mateas and Montfort). Weird programming languages are also an ideal site to examine an expanded rhetorics of software: "arguing *about* software, arguing *with* software, and arguing *in* software" (Brown 173). Because weird languages use code itself, to explore the structures of both specific languages and broad principles and practice, they explore and comment on the both specific languages and broad coding principles and practices. In other words, weird programming languages make arguments *about* code *with* and *in* code.

In the following sections, I focus on C+='s "Hello, World!" program to show how digital manspreading happens not only through online discourse but also can be created and afforded by digital infrastructures. A "Hello, World!" program is an ideal focus because it is a simple program that is often used to briefly demonstrate the styles and logics of a programming language. In this analysis, I turn to both the textual and procedural of C+='s "Hello, feminists!" program to explore how space is used to argue that a woman's place, as well as other non-male, non-normative identities, is not online. In looking textually, I examine the language of code and code comments to consider how the program's discursive elements work spatially. However, because code *does* and not only *says*, I also consider how the procedures themselves make claims to space. This approach is important because looking only to the language of code, such as through the methods of textual criticism, could lead to a misinterpretation of the misogyny of C+=. Considering both the *says* and *does*, language and procedure, gives a more complete

picture of not only the project's misogyny but also the integral role of digital manspreading. Through its procedural arguments, C+= highlights the limits of attending only to *says* and draws our attention to the material impact of the *does* for understanding how gendered power differences are created and enforced online.

Discursive Excess

In this section, I focus on the code comments in C+=’s “Hello, feminists!” As the most obviously discursive elements, code comments are ideal sites for focusing on language and human audiences. Unlike code proper, code comments are *only* intended for human audiences; they are overlooked entirely by the machine. Typical code comments might explain a variable, note a change, or provide a rationale for a choice because this can help future programmers more easily work with their code. In C+=, however, code comments do something quite different. In these comments, I identify two ways the authors make spatial claims that enact misogyny. Although code comments are often necessary for clarity, those in this program are so numerous that they overwhelm and obscure the program itself. Additionally, its imitations of feminism produce a territorialization not only of the space of feminist programming, but also feminist voices.

The discursive excesses of this program can be seen by comparing it to C++. To print “Hello, World!” on the screen, a C++ programmer could write seven lines, including extra spacing for clarity. In “Hello, feminists!,” this program is *89 lines long*, and this expansion can be attributed to large, multi-line comments written through feminist voices. The most unusual aspect of these comments includes a 57-line satirical epic of the C+=’s creation. This tale begins

with its exigence—Schlesinger's proposal that coding logics privilege certain ways of thinking and being:

> For millennia, human society has taught young, new programmers
> their toxic, seminal first lesson in the form of the Patriarchal
> "Hello, World!" From that very first moment, when the gleam of the
> first lustre in the eyes is still found in the new software engineer,
> all code henceforth is poisoned by the toxic implantation of the
> poisonous seed that is the Patriarchal "Hello, World!" (10-15)

The story tells how this first coding experience embeds programmers in patriarchal systems. The scene of this dystopia is hellish, and the narration slips into silly imitations of earlier forms of English in describing it: "Forsooth a tragedy moost infexioous!" (30). Then, hope arrives as a bright light that overthrows patriarchy, crowning a queen instead of a king. From this, a new order, which "throws Men down only to raise Women up," culminates in the creation C+=''s version of "Hello, World" (46). After this narrative, the comments introduce the authors as members of the Feminist Software Foundation, and the code of the program begins, interspersed with a quatrain about Juno, not Zeus, as the creator of the world.

In expanding a simple program to nearly ninety lines full of this narrative, the authors of C+= show one way asserting dominance online happens spatially. Before a feminist "Hello, World!" could be created out of the conversations around Schlesinger's post, the authors write one that is overwhelmingly, and almost incomprehensibly, large. However, saying more is not the only way this program is digital manspreading. The voice and style of these comments

reveals a further, more troubling spatial dominance—a territorialization of women's voices and identities.

Likely because of the program's collaborative origins, voice switches quickly throughout, but one constant is their positioning as feminists. The primary voice in this program is the mythic narrator who tells the epic and discusses patriarchy, oppression, privilege, and the important role of language in all of these. This use of feminist concepts can be seen throughout the project, where issues such as rape, gender identity, and trigger warnings are mocked. In particular, "trigger warnings" are used throughout C+= as error messages.[19] All of these choices are justified based on the claimed feminist identities of the authors: "we understand first-hand / the oppressions that true feminists worldwide have to endure / every single microsecond (54-6)."

This co-opting of feminist concepts and identities is a key feature throughout this program and can be seen in a second voice: a woman online. Using Internet speak, including abbreviations, emojis, creative spelling and punctuation, and hyperbole, this voice attempts to position women programming, and likely Schlesinger in particular, as incompetent:

> Today I wrote my first reclamation program for the
>
> Feminist Software Foundation1 I'm sooooo excited! ^_^
>
> Imma cccdddrrr loll
>
> I'm such a nerd!
>
> I think I'll write an essay on this triumph over the Patriarchy! (82-6)

[19] One such trigger warning is described in the README file, although it is not included in any of the sample programs. The authors create a condition called "consent" that would control whether or not a variable can be acted on. Under this imagined condition, variables are randomly assigned "a percentage of consent" that affects whether or not the variable can be used. They imagine that ignoring a variable's consent "will result in C+= throwing a ForcedInsertionTriggerWarning."

By taking on this voice, the authors leave little room for the possibility of a competent woman coder. Standing in as a programmer for the Feminist Software Foundation, this voice paints women as necessarily novice at coding and perhaps computer use in general. In expanding their discourse beyond themselves and into the voice of another, the authors attempt to invalidate the actual voice of Schlesinger.

This strategy continues in forum discussions, as users created accounts that used real women's photos and names as well as parodied feminist theorists, such as "bellhooks666." This is an effort, as in the code comments, to not only take up the space of the conversation but also shrink other perspectives. For example, one post argues that "as a programmer, woman, and partial feminist, I really don't see the problem" with C+= ("Issue"). However, as users in the forum debate the code, questions are also raised about the authenticity of these women's accounts, including research by users to contact the women who appear to be posting. Although this forum begins as a discussion on the authenticity of the Feminist Software Foundation, it quickly turns into a debate over the authenticity of all visibly non-male identities participating in the discussion. In this scenario, even men who are saying problematic things get to be real or legitimate.

This rhetorical strategy has been observed in other anti-feminist harassment campaigns. One such comparable event was Operation: Lollipop, which was also the collaborative effort of 4chan users. By posing as feminist women of color on Twitter, a group of men's rights advocates attempted to "jeopardize the legitimate online feminist community" (Ganzer 1098). What digital manspreading further reveals about this strategy is that it has spatial methods and effects. As in Operation: Lollipop, the authors of C+= take over and take up the space of women and feminists

to speak and act—or, even be present—online. In the forum discussions, visible maleness becomes safe whereas women might always be untrustworthy betrayers, and it makes true and gives material consequence to long-standing stereotypes about women as untrustworthy. This is a crucial feature of digital manspreading: it is not only taking up space online, but doing so in ways that cause others to have less. This attention to digital manspreading helps show how such imitation of feminism is much more than parody; it is also territorializing and silencing.

Procedural Excess

As the code comments and online discussions show, the language of online misogyny is indeed rich for feminist analysis. However, I argue that the code itself is essential to understanding C+=’s misogyny and reveals that digital manspreading is not only discursive but also infrastructural to our digital interactions and lives. Although there are many variations on “Hello, World!” programs (and, since it is rhetorical, no “right” way to write it), one way to render this program in C++ could be done, including extra spaces for clarity, in seven lines of code:

```
1    #include <iostream>
2
3    int main()
4    {
5    std::cout << “Hello, World!”;
6    return 0;
7    }
```

In line 1, this program tells the preprocessor to include the standard input/output library, which includes a class of objects for use later in this program. Line 3 creates a value returning function, which means that the program will produce an integer as a result of the program's actions. In this case, the integer is used to signal whether or not the program has properly executed. The curly brackets contain these actions. Line 5 uses the standard console out object (std::cout) as defined by the iostream library to display the characters in quotation marks on the screen. Line 6 would return 0, signalling that the program has properly executed, and the final bracket on line 7 terminates the program.

In "Hello, feminists!," we can see how the program territorializes textually through define directives and procedurally through an additional function:

```
1  #consider <feminist_brevity_in_light_of_masculine_long-windedness.Xir>
2  #consider <iostream>
3
4  xe womain ()
5    {
6       PrivilegeCheck();
7       sti::cout of_the_following "Hello, feminists!/n";
8    ENDMISOGYNY
```

Line 2 calls the same standard library as the original, but line 1 includes an additional file. Although this file, "feminist_brevity_in_light_of_masculine_long-windedness.Xir," does not exist in any other documents related to the language as of January 2018, we can deduce that this file contained (or was intended to contain) define statements, creating renamed macros, or

instructions, to be used in the program. The define macro is a simple, but effective, text replacement function, allowing for the authors to create a working version of C++ with misogynistic concepts and wordplay switched out. For example, this file might contain a short line of code: "#define int xe", which would cause line 4 to display "xe" instead of "int" as in C++.

In the README, the authors explain these replacements are intended to be a correction of "anything that can be construed as misogynist," and in this program, the define statements have made several attacks on non-male, non-normative identities possible. This use of "xe" instead of "int" (integer) in an attack on transgender identities because, as the README file explains, "xe" replaces numerical data types integer, long, and double because identity is the choice of the variable. "Womain" replaces the "main" function in a play on woman replacing man, which is a central critique of feminism, as understood by the authors, throughout. As in the creation story, the feminism being mocked in C+= is not equality, but woman over man. "ENDMISOGYNY" is used in C+= for all structure terminators to, as the authors describe, "turn patriarchal control structures into liberation statements," and textual switches like these occur throughout the language.

Although attending to this language in code is crucial and can certainly be a site of misogyny, racism, and bias, this case also demonstrates the limits of textual analysis for understanding misogyny online. If analyzed only at the level of language, this case might appear to be nothing more than poor satire or clumsy feminism. In fact, those who believed the project was authentic used this clumsiness as evidence in online discussions that women couldn't or

shouldn't code. However, looking at the rules and execution of the code reveals how the authors also use procedures to express misogyny.

While define directives do not change the way the code functions, the addition of a function in line 6 does, and this procedure powerfully demonstrates how spatial dominance is central to the program's misogyny. Textually, this line plays on the concept of computing privileges to mock feminist concerns. However, if we look procedurally, this function can help us better understand the spatial claims being made. Defined in a separate PrivilegeCheck file, the PrivilegeCheck function begins by calling another PrivilegeCheck function, which would cause the program to execute like this:

```
PrivilegeCheck() {
        PrivilegeCheck() {
                PrivilegeCheck() {
                        PrivilegeCheck() {
```

When a program begins, a section of memory is reserved for executing it. This section of memory, called "the stack," is further allocated into smaller sections for each function called. In this program, each time a PrivilegeCheck is called, a segment of the stack is reserved. As each PrivilegeCheck creates new PrivilegeChecks, it takes up additional space on the stack. However, because the memory on the stack is finite, once the function calls itself more times than can fit on the stack, the program would crash and throw a stack overflow error, or more accurately in C+=, a "trigger warning." Depending on the stack size, this loop could initiate hundreds of times, and take up hundreds of lines, before ending the program in an error, allowing the program to grow to take up as much space as possible. Because of this, the program will never fully execute

and will always crash on line 6, meaning that “Hello, feminists!” will never actually print on the screen.

In the code comments, the authors reference the failure of the program often, indicating the error is undoubtedly intentionally: “If this program fails to operate, it is due to the Patriarchy backfiring upon itself, and no refunds will be issued” (93). By overwhelming the memory space and crashing, this line of code argues that feminism is illogical and doesn’t belong in computer science. The code comments note that the PrivilegeCheck function calling itself “appears to be problematic, but only if you view it / from a CIS HETRO or MALE point of view” (78-9). Through this procedural dominance of memory space, the program mocks serious discussions about the executability of feminist code and shrinks the perspectives of those who would discuss them. Digital manspreading causes a failure of the feminist project by taking up too much space.

In the textual and procedural uses of space in the code and code comments, the authors of C+= enact the first file they call in this piece of code: “Feminist_brevity_in_light_of_masculine_long-windedness” (59). Through this discursive and procedural excess that stretches their own positions across as much space as possible, the authors of this program use “long-windedness” as a distinctly spatial and masculine act. Their harassment is long-winded in both size and relentlessness. The program is overwhelmingly large, nearly nine times longer than it need be. The program is also nonstop. The recursive function runs endlessly until the entire program crashes. Importantly, the procedures of the program don’t call for “feminist brevity.” They demand it. This is a digital manspreading that, like its physical counterpart, uses spatial domination to shrink the power and presence of others.

This case study shows the need for feminist scholars to turn to digital infrastructures, especially code, to understand misogyny online, and digital manspreading offers one way to draw on existing feminist theories to do so. Although I have focused here on the potentials of code, digital manspreading could be a helpful lens in a variety of digital contexts. For example, the spatial claims and tactics used in the code are also taken up more broadly in C+=’s multi-faceted harassment campaign. Using some variation on the username “FeministSoftwareFoundation” and a C+= logo, the authors created a web presence designed to reach into as many spaces as possible. In addition to GitHub and other code-specific sites, the Feminist Software Foundation had presences through Reddit, Facebook, YouTube, their own website, email address, and Bitcoin account, where it appears that others could donate to their efforts. Using these sites, the authors attempt to take over a variety of spaces, and their posts were designed to create conversation, discussion, and resharing to take up more online spaces. Concerns for space can also be seen within the language of the posts and comments. When C+= was reported as harassment and removed from GitHub, users responded with outrage at a felt loss of a masculine space. This can be seen in a variety of comments that urge the authors to “make your own website and host it on your own hardware” to avoid C+= being taken down (“GitHub”).

It is especially worth observing that their message dominated on sites where spatiality determines how the content is expressed, such as displaying more popular posts at the top, and this draws our attention to how the structures of digital environments can afford such misogyny. As Adrienne Massanari has shown, Reddit in particular affords such “toxic technocultures” in its platform politics. This digital manspreading, however, is not exclusive to such blatant examples

of misogyny as C+=. The tendency for men to take up more online space in all kinds of everyday contexts has been strikingly confirmed by feminist research. In moderated news sites, Fiona Martin found that men represent 80% of commenters. These findings are supported and made more consequential through an analysis of space, and digital manspreading attempts to capture these everyday realities of misogyny online by drawing attention to its spatiality.

Digital manspreading also offers an avenue for feminist scholarship to interrogate digital materiality and embodiment. One result of C+=’s aggressive and pervasive spatial claims is an overwhelming colonization of digital feminist space. Digital manspreading draws our attention to how such misogynistic claims to space are problematic because there is even a limited amount of space in the seemingly boundless internet. If misogynists take “Feminist Software Foundation” not only in name but also on all of these sites, then it is no longer available to feminists. In fact, searching for “feminist software” on Google turns up results that are overwhelmingly associated with this hoax and not real feminist inquiry into gender and software. This silencing calls for feminist scholars to attend seriously to the everyday spatial, material, and embodied structures and forces of online misogyny. If we allow our online spaces to be conceptualized as immaterial, boundless, or disembodied, we risk missing not only how our raced, classed, and gendered bodies affect online interactions but also what the real, material consequences are for differently embodied users.

Queer Code: In Conclusion

```
t = 0
while True:
        print("Nothing lasts forever.")
        t += 1
```

—Source Code Poetry submission

Throughout writing this dissertation, there has been an ambient feeling at the edges of everything it touches—the hope and promise that "Everyone Can Code."[20] Government, corporate, and non-profit campaigns to promote coding literacy have been well underway for nearly a decade, and they are supported by the development of ever more accessible no-code and low-code programming languages. It is an exciting time to think about code.

But this promise has always felt hollow. This project has shown that while everyone can code, not everyone does become a *fully human* coder. To show how non-normative bodies can be marked as less-than, these chapters have unpacked the rhetorical mechanisms about, with, and in code that construct coding as immaterial, disembodied, and unfeeling. Case studies in weird programming languages make these rhetorics visible by taking coding norms to their extreme. In each chapter, I've shown how these ideals about code also shape what we believe about the ideal programmer, who is invisibly and abstractly Western, white, cis, straight, male. Altogether, these cases show that digital inequality is infrastructural, not incidental, which makes code is an important site for understanding digital rhetorics.

Although this project has focused on mechanisms of inequality, it has also given me hope that we can do and be otherwise in code. These cases demonstrate that while beliefs about code's

[20] Apple's "Everyone Can Code" initiative is only one example of the coding literacy movement's growth and saturation.

rigid, unfeeling, impersonal enforcement persist, code is authored—and can be reauthored. In this conclusion, I want to linger with some remaining tensions and imagine other possibilities these cases show for reauthoring code that are disruptive, transformative, and create more livable futures: Can weird code be queer code?

The queer potential of weird programming languages has been one of the greatest questions of this project. It has also been one of its greatest tensions. I've turned to many scholars imagining queer computing possibilities in conceptualizing this dissertation. I found they would often invoke weird programming languages as examples of such queerness, highlighting their play, disruption, and unproductivity. However, these invocations usually made me uncomfortable. If we take a broad, strong understanding of queer as "everything that is antinormative," then weird programming languages are certainly that. But, it also feels unsettling to champion objects made largely by normative bodies for normative bodies as models of queerness. What the rhetorical approach of this dissertation has offered, then, is a more nuanced, contextual frame for this tension, what Jean Bessette argues for as a methodology of "queer rhetoric in situ." By considering circulation, reception, authors, and audiences, rhetoric reveals how we can see "a single act as both *queer* and *not queer*" (153, emphasis original). These languages do demonstrate disruptive, antinormative computing possibilities, but they also can use those very same features to shore up normative structures and power. In doing so, these cases demonstrate the vital importance of a rhetorical approach to digital objects. By putting digital objects firmly within their space, time, and embodiments—refusing to allow it arhetoricity—this dissertation ultimately argues that rhetoric offers an important perspective for rebodying digital studies.

This project attunes to such rhetorical mechanisms to situate weird programming languages within systems of white supremacy and heteropatriarchy, and as such, I want to be clear that weird programming languages themselves are not inherently and automatically enacting queerness toward a more livable world, as more speculative scholarship has often invoked them. While this dissertation has used critique to show the ways that these languages participate fully in normative power structures, I am invested in critique as a mode that also seeks queer possibility. In chapter 4, I show how the authors of an anti-feminist programming language make their argument that women, queer, and transgender bodies can't or shouldn't code by writing intentionally broken code. I focus on how this particular piece of broken code works by taking up space. However, in thinking toward the future of this project and toward potentiality, this case is most interesting because it demonstrates that not only is nonexecutable code expressive but that breaking *intentionally* can be a powerful argument. This opens up the possibility for computational expression that is not only nonbinary or machine noncompliant but intentionally and fiercely resistant. In pursuing these directions, considering codework and code poetry, like the example opening this chapter, could be helpful expansions of what we can do in code.

This dissertation has also shown that we can also do and be otherwise in digital rhetoric. Before conceptualizing this dissertation around weird programming languages, I was concerned with making space for embodiment in digital rhetoric. Working at the intersections of feminist rhetorics and software studies has always placed incredible tension on questions of embodiment, difference, and power. I've been challenged to take the rhetorical force of objects seriously while not flattening real embodied inequality into an abstract "human," who, as I argue in chapter 2, is

actually a very particular Man. By coming to questions of the less-than human, instead, this project contributes one way to dwell in that posthumanist tension. It goes after the ways that objects work—not only in the world but also within rhetorical theory—to build and sustain the power of abstract, invisible bodies. This shows the need to learn from black feminist, queer, and indigenous scholars, who have long theorized what it means to have a range of human/nonhuman relationships, including the rhetorical challenges and strategies of the less-than human. As digital rhetoricians wrestle with the increasingly blurred boundaries between human and machine, on and offline, interrogating the seemingly invisible bodies can be one way to reveal the fleshy excesses not collapsable into the fully "human."

This approach is especially important because it also gives us a way to move beyond the recovery of untold computing innovators. Recovery is important and vital work in both scholarly and popular contexts, and it has the potential to help us rethink computing structures and rhetorics, as seen in queer recovery of Alan Turing (Blas and cárdenas, Fancher, Halberstam). Margot Lee Shetterly's popular breakthrough *Hidden Figures* has also made black women visible in computing. However, recovery carries the risk of obscuring the systemic forces and structures of programming, erasing differences between "nameless disappearing computer operators" and singular heros (Chun 34). Furthermore, as queer scholars have shown, such inclusion politics usually don't lead to a "coming back" for everyone that is left behind (Chávez).

Like recovery, it also isn't enough to recruit diverse coders. In fact, Safiya Umoja Noble calls this "an excuse" for not fixing the real problem. This dissertation shows that this excuse is premised on the belief that the technology itself is arhetorical and thus couldn't be the problem. The emptiness of this excuse is compounded by a literacy studies perspective. Not only are the

socioeconomic benefits of literacy not granted equally but the ways we teach literacy often also work to sustain existing power structures (Vee).

What my dissertation demonstrates, then, is the need for transformative access that changes not only *who* codes but also how we use, think, talk, and interact with code. At this moment, when coding is emerging as a literacy and there are ever new ways to code, it is an exciting and important time for such a rhetorical shift. Taking coding bootcamps as a focus, future qualitative research could explore if and how rhetorics of the disembodied, immaterial, and unfeeling circulate in pedagogies otherwise meant to upend power structures. Because many bootcamps are intentionally designed to recruit underrepresented populations, a central goal is often to resist and counter programming's exclusionary structures. Yet, as these case studies show, such beliefs about code are so pervasive as to be invisible, and a focus on job acquisition might intensify such a focus. As a common entry point, bootcamps offer the possibility to think and do code differently from a coder's beginning. Weird programming languages might compliment these efforts by not only exposing norms and code's rhetorical force but also by expanding the boundaries of computational possibility when first encountering code. Such a study could develop pedagogies that move us from imagining to actually building a better digital world.

Works Cited

Abbate, Janet. *Recoding Gender. Women's Changing Participation in Computing.* MIT Press, 2012.

Acaron, Thania. "Shape-in(g) Space: Body, Boundaries, and Violence." *Space and Culture* 19.2 (2016): 139-49. Web. *SAGE*. 24 March 2017.

Aching, Gerard. "The Slave's Work: Reading Slavery through Hegel's Master-Slave Dialectic." *PMLA,* vol. 127, no. 4, 2012, pp. 912-7, www.mlajournals.org/doi/abs/10.1632/pmla.2012.127.4.912 Accessed 10 Sept. 2018.

Ahmed, Sarah. *The Cultural Politics of Emotion*. Edinburgh University Press, 2004.

"Assembly Language Instruction." *ScienceDirect*, https://www.sciencedirect.com/topics/engineering/assembly-language-instruction. Accessed 28 May 2019.

Bacon, Nora. *The Well-Crafted Sentence: A Writer's Guide to Style*, Macmillan Higher Education, 2018.

Barnett, Scot and Casey Boyle. *Rhetoric, through Everyday Things*, University of Alabama Press, 2016.

Bessette, Jean. "Queer Rhetoric in Situ." *Rhetoric Review* 35.2 (2016): 148-164.

Black, Maurice J. "The Art of Code." Dissertation, University of Pennsylvania, 2002. *ProQuest Dissertations and Theses Global*, search.proquest.com.ezproxy.library.wisc.edu/docview/305507258?accountid=465. Accessed 10 Sept. 2018.

Blas, Zach and Micha Cárdenas. "Imaginary Computational Systems: Queer Technologies and Transreal Aesthetics," *AI & Society*, vol. 28, no. 4, pp. 559-66, 2013.

Bogost, Ian. *Persuasive Games: The Expressive Power of Videogames*. MIT Press, 2007.

Booher, Amanda K. and Julie Jung, eds. *Feminist Rhetorical Science Studies: Human Bodies, Posthuman Worlds.* Southern Illinois University Press, 2018.

Boyle, Casey. *Rhetoric as a Posthuman Practice*, Ohio State University Press, 2018.

Brock, Kevin. *Rhetorical Code Studies: Discovering Arguments in and around Code*, University of Michigan Press, 2019.

Brock, Kevin and Ashley Rose Mehlenbacher. "Rhetorical Genres in Code." *Journal of Technical Writing and Communication* (2017): 1-29. Web. *SAGE*. 24 Nov 2017.

Brooke, Colin Gifford. *Lingua Fracta: Toward a Rhetoric of New Media*, Hampton Press, 2009.

Brown, Gregory T. *Programming Beyond Practices: Be More Than Just a Code Monkey.* O'Reilly, 2016.

Brown, James J., Jr. *Ethical Programs: Hospitality and the Rhetorics of Software*. University of Michigan Press, 2015.

Browne, Simone. *Dark Matters: On the Surveillance of Blackness*, Duke University Press, 2015.

Brummett, Barry. *A Rhetoric of Style*, Southern Illinois University Press, 2008.

Buck-Morss, Susan. *Hegel, Haiti, and Universal History.* University of Pittsburgh Press, 2009.

Canagarajah, A. Suresh. *Resisting Linguistic Imperialism in English Teaching*, Oxford University Press, 1999.

Chaitin, Gregory J. "Computers, Paradoxes and the Foundations of Mathematics," *American Scientist,* vol. 90, no. 2, 2002, pp. 164-71.

Chávez, Karma R. "The Body: An Abstract and Actual Rhetorical Concept." *Rhetoric Society Quarterly,* vol. 48, no. 3, 2018, pp. 242-250.

Chun, Wendy Hui Kyong. *Programmed Visions: Software and Memory*. MIT Press, 2011.

Criado-Perez, Caroline. "The Deadly Truth about a World Built for Men—From Stab Vests to Car Crashes," *The Guardian*, 23 Feb. 2019, https://amp.theguardian.com/lifeandstyle/2019/feb/23/truth-world-built-for-men-car-crashes?fbclid=IwAR1iaZAcmk6stz_qxMB2S6zlJXFOFO7J_F15XhJR6eeyrsWPjz9AgVlXLLc, Accessed 26 May 2019.

Cox, Geoff and Alex McLain. *Speaking Code: Coding as Aesthetic and Political Expression*. MIT Press, 2012.

Cushman, Ellen. *The Cherokee Syllabary: Writing the People's Perseverance.* University of Oklahoma Press, 2012.

D'Ignazio, Catherine and Lauren Klein. *Data Feminism*, MIT Press Open, 2018.

Dolmage, Jay. *Disability Rhetoric*. Syracuse University Press, 2014.

Duncan, Mike and Star Medzerian Vanguri. *The Centrality of Style,* WAC Clearinghouse, 2013.

Elwood, Sarah and Agnieszka Leszczynski. "Feminist Digital Geographies," *Gender, Place & Culture: A Journal of Feminist Geography*, vol. 25, no. 5, pp. 629-644.

"ENIAC: Birth of the Computer," *Computer History Museum*, https://www.computerhistory.org/revolution/birth-of-the-computer/4/78. Accessed 26 May 2019.

Ensmenger, Nathan L. *The Computer Boys Take Over: Computers, Programmers, and the Politics of Technical Expertise.* MIT Press, 2012.

ErisBlastar. "cplusequality." *GitHub*. 4 May 2016. Web. 10 May 2016.

Eyman, Douglas. *Digital Rhetoric: Theory, Method, Practice.* Univ of Michigan Press, 2015.

Fancher, Patricia. "Embodying Turing's Machine: Queer, Embodied Rhetorics in the History of Digital Computation." *Rhetoric Review,* vol. 37, no. 1, 2018, pp. 90-104, doi:10.1080/07350198.2018.1395268. Accessed 18 Oct. 2018.

Flores, Lisa A. "Between Abundance and Marginalization. The Imperative of Racial Rhetorical Criticism," *Review of Communication*, vol. 16, no. 1, pp. 4-24, 2016.

Forbes, Clinton. *Diving into INTERCAL*, [website], Mar. 2007, http://divingintointercal.blogspot.com/, Accessed 26 May 2019.

Ford, Paul. "What is Code?" *Bloomberg Businessweek*, 11 June 2015, https://www.bloomberg.com/graphics/2015-paul-ford-what-is-code/. Accessed 26 May 2019.

Frederick, Christine Ann Nguyen. "Feminist Rhetoric in Cyberspace: The Ethos of Feminist Usenet Newsgroups." *Information Society* 15.3 (1999): 187-97. Web. EBSCOhost. 8 March 2017.

"/g/ introduces the specification for C+=." *4chan*. Web. 4 May 2016.

Galloway, Alexander R. *Protocol: How Control Exists after Decentralization,* MIT Press, 2004.

Ganzer, Miranda. "In Bed with the Trolls," *Feminist Media Studies*, vol. 14, no. 6, pp. 1098-1100, 2014.

"GitHub Takes Down Satirical 'C Plus Equality' Language." *Slashdot*. 14 Dec 2013. Web. 20 April 2016.

Goffey, Andrew. "Technology, Logistics and Logic: Rethinking the Problem of Fun in Software," in *Fun and Software: Exploring Pleasure, Paradox and Pain in Computing,* ed. Olga Goriunova, Bloomsbury Academic, 2014.

Gouge, Catherine and John Jones. "Wearable Rhetorics: Bodies, Cities, Collectives." *Special issue of Rhetoric Society Quarterly*, vol 46.3, 2016.

Gries, Laurie. *Still Life with Rhetoric: A New Materialist Approach for Visual Rhetorics*, University Press of Colorado, 2015.

Halberstam, J. Jack. *The Queer Art of Failure.* Duke University Press, 2011.

Hallenbeck, Sarah. "Toward a Posthuman Perspective: Feminist Rhetorical Methodologies and Everyday Practices." *Advances in the History of Rhetoric,* vol. 15, no. 1, 2012, pp. 9-27, doi: 10.1080/15362426.2012.657044. Accessed 18 Oct. 2018.

Hanselmann, Nik. "bodyfuck," http://nik.works/project/bodyfuck/. Accessed 6 May 2019.

Hartikka, Lauri. "ArnoldC," [programming language], *Github*, https://github.com/lhartikk/ArnoldC. Accessed 26 May 2019.

Hawhee, Debra. "Rhetoric's Sensorium." *Quarterly Journal of Speech*, vol 101.1, 2015, pp. 2-17.

Hayles, Katherine N. *My Mother Was a Computer: Digital Subjects and Literary Texts*. Chicago: University of Chicago Press, 2005. Print.

Hegel, G. W. F. *Phenomenology of Spirit,* Translated by A. V. Miller. Oxford UP, 1977.

Herbst, Claudia. *Sexing Code: Subversion, Theory and Representation*, Cambridge Scholars Publishing, 2008.

Hicks, Marie. *Programmed Inequality: How Britain Discarded Women Technologists and Lost its Edge in Computing.* MIT Press, 2017.

Inoue, Asao. *Antiracist Writing Assessment Ecologies: Teaching and Assessing Writing for a*

Socially Just Future, WAC Clearinghouse, 2015.

"INTERCAL." *Progopedia*, http://progopedia.com/language/intercal/. Accessed 5 May 2019.

"Issue #8629: Harassing Repository." *Bitbucket*. 13 Dec 2013. Web. 10 April 2016.

Jack, Jordynn. "Acts of Institution: Embodying Feminist Rhetorical Methodologies in Space and Time." *Rhetoric Review,* vol. 28, no. 3, 2009, pp. 285-303, doi: 10.1080/15362426.2012.657044. Accessed 18 Oct. 2018.

Jane, Emma A. "'Back to the kitchen, cunt': speaking the unspeakable about online misogyny." *Continuum: Journal of Media & Cultural Studies* 28.4 (2014): 558-70. Web. *Communication & Mass Media Complete*. 28 March 2017.

---. "'Dude … stop the spread': antagonism, agonism, and #manspreading on social media." International Journal of Cultural Studies (2016): 1-17. Web. *SAGE*. 28 March 2017.

Janetakis, Nick. "Are You a Computer Scientist or a Code Monkey?" *Nick Janetakis*, 27 Nov. 2016, /nickjanetakis.com/blog/are-you-a-computer-scientist-or-a-code-monkey. Accessed 10 Sept. 2018.

Kennedy, Krista. "Designing for Human-Machine Collaboration: Smart Hearing Aids as Wearable Technologies," *Communication Design Quarterly Review*, vol. 5, no. 4, pp. 40-51, 2018.

Kesteloot, Lawrence. "Introduction to Programming." *Team Ten*. www.teamten.com/ lawrence/programming/intro/. Accessed 10 Sept. 2018.

Kirschenbaum, Matthew. *Mechanisms: New Media and the Forensic Imagination*, MIT Press, 2008.

Kitchin, Rob and Martin Dodge. *Code/Space: Software and Everyday Life*, MIT Press, 2011.

Klein, Lauren F. "The Carework and Codework of the Digital Humanities," Digital Antiquarian Conference 2015, http://lklein.com/2015/06/the-carework-and-codework-of-the-digital-humanities/, Accessed 26 May 2019.

Lapowsky, Issie. "If Congress Doesn't Understand Facebook, What Hope Do Its Users Have?" *WIRED*, 10 Apr. 2018, https://www.wired.com/story/mark-zuckerberg-congress-day-one/, Accessed 26 May 2019.

Lin, Charles. "Understanding the Stack." *CMSC 311 Computer Organization course site*, 22 June 2003. Accessed 24 March 2017.

Lockhart, Tara. "The Shifting Rhetorics of Style: Writing in Action in *Modern Rhetoric*," *College English*, vol. 75, no. 1, 2012.

Losh, Liz. "Sensing Exigence: A Rhetoric for Smart Objects." Computational Culture, vol 5, 2016, http://computationalculture.net/sensing-exigence-a-rhetoric-for-smart-objects/. Accessed 6 May 2019.

Mantilla, Karla. *#Gendertrolling: How Misogyny Went Viral*. Praeger Press, 2015.

Mahoney, Michael S. *Histories of Computing*, Harvard University Press, 2011.

---. "Software: The Self-Programming Machine," *From 0 to 1: An Authoritative History of Modern Computing,* Eds. Atsushi Akera and Frederik Nebeker, pp. 91-100, Oxford University Press, 2002

Marino, Mark C. "Of Sex, Cylons, and Worms: A Critical Code Study of Heteronormativity."

Leonardo Electronic Almanac, vol. 17, no. 2, 2012, pp. 184-201, www.leoalmanac.org /vol17-no2-of-sex-cylons-and-worms/. Accessed 19 Oct. 2018.

---. "Reading *exquisite*_code: Critical Code Studies of Literature." *Comparative Textual Media: Transforming the Humanities in the Postprint Era.* N. Katherine Hayles and Jessica Pressman, eds. Minnesota: University of Minnesota Press, 2013. Print.

Martin, Katherine Connor. "Manspreading: how New York City's MTA popularized a word without actually saying it." *Oxford Dictionaries Blog*. 4 Sept 2015. Web. 24 March 2017.

Martinez, Dagan. "FetLang," [programming language], *Github*, https://github.com/fetlang/fetlang/blob/master/docs/tutorial.md, Accessed 6 May 2019.

Massanari, Adrienne. "#Gamergate and The Fappening: How Reddit's Algorithm, Governance, and Culture Support Toxic Technocultures." *New Media & Society*, vol. 19, no. 3, pp. 329-46, Nov 9, 2015.

Mateas, Michael and Nick Montfort. "A Box, Darkly: Obfuscation, Weird Languages, and Code Aesthetics." *Proceedings of the 6th Annual Digital Arts and Culture Conference,* 2005, elmcip.net/critical-writing/box-darkly-obfuscation-weird-languages-and-code-aesthetics. Accessed 18 Oct. 2018.

Melonçon, Lisa, ed. *Rhetorical Accessability: At the Intersections of Technical Communication and Disability Studies*, Baywood Press, 2013.

Milner, Ryan M. *The World Made Meme: Public Conversations and Participatory Media*. MIT Press, 2016.

Montfort, Nick. "Continuous Paper: Early Materiality and Workings of Electronic Literature," *Modern Language Association 2004, Philadelphia, PA*,

http://nickm.com/writing/essays/continuous_paper_mla.html. Accessed 26 May 2019.

---. Exploratory Programming for the Arts and Humanities, MIT Press, 2016.

Montfort, Nick, et al. *10 PRINT CHR$(205.5+RND(1)) :GOTO 10*. MIT Press, 2013.

Morgan-Mar, David. "Piet," [programming language], *Esolangs Wiki*, https://esolangs.org/wiki/Piet. Accessed 26 May 2019.

Moynihan, Tim. "With an NSFW Name, This Tech Makes Coding Pure Torture." WIRED, 30 June 2015, https://www.wired.com/2015/07/bodyfuck/. Accessed 5 May 2019.

Müller, Urban. "Brainfuck, or how I learned to change the problem." Tamedia TX, 13 June 2017,

Tamedia AG, Zürich. Conference presentation. www.youtube.com/watch?v=gjm9irBs96U&feature=youtu.be&t=8722. Accessed 10 Sept. 2018.

Nakamura, Lisa. *Digitizing Race: Visual Cultures of the Internet*, University of Minnesota Press, 2007.

Nguyen, Clinton. "What is 'What is Code?'" *VICE*, 12 June 2015, https://www.vice.com/en_us/article/qkv9vd/what-is-what-is-code. Accessed 27 May 2019.

Noble, Safiya Umoja. *Algorithms of Oppression: How Search Engines Reinforce Racism*. NYU Press, 2018.

Oram, Andy and Greg Wilson, eds. *Beautiful Code: Leading Programmers Explain How They Think.* O'Reilly, 2007.

Osenga, Kristen. "The Internet is Not a Super Highway: Using Metaphors to Communicate

Information and Communications Policy." *Journal of Information Policy*, vol. 3, pp. 30-54, 2013.

Phillips, Whitney. *This Is Why We Can't Have Nice Things: Mapping the Relationship between Online Trolling and Mainstream Culture*. Cambridge: MIT Press, 2015. Print.

Post, Ed. "Real Programmers Don't Use PASCAL," *Datamation*, vol 29., no. 7, July 1983, https://www.pbm.com/~lindahl/real.programmers.html. Accessed 29 May 2019.

Pronger, Brian. "On Your Knees: Carnal Knowledge, Masculine Dissolution, Doing Feminism." *Men Doing Feminism.* Tom Digby, ed. New York: Routledge, 1998.

Rice, Jenny Edbauer. "Rhetoric's Mechanics: Retooling the Equipment of Writing Production," *College Composition and Communication*, vol. 60, no. 2, pp. 366-87, 2008.

Rieder, David M. *Suasive Iterations: Rhetoric, Writing, and Physical Computing*, Parlor Press, 2017.

Sample, Mark. "Criminal Code: Procedural Logic and Rhetorical Excess in Videogames," *Digital Humanities Quarterly,* vol. 7, no. 1, 2013.

Schlesinger, Arielle. "A Feminist & A Programmer." *HASTAC*. 13 Dec 2013. Web. 19 April 2016.

Selfe, Cynthia L., and Richard J. Selfe. "The Politics of the Interface: Power and Its Exercise in Electronic Contact Zones." *Computers and Composition,* vol. 45, no. 4, 1994, pp. 480-504, doi: 10.2307/358761. Accessed 10 Sept. 2018.

Shetterly, Margot Lee. *Hidden Figures: The Untold True Story of Four African American Women*

who Helped Launch Our Nation into Space. William Morrow, 2016.

Singh, Julietta. *Unthinking Mastery: Dehumanism and Decolonial Entanglements.* Duke University Press, 2018.

Smith, Dale M. and James J. Brown, Jr. "For Public Distribution," *Circulation, Writing, and Rhetoric*, Eds. Laurie Gries and Collin Gifford Brooke, Utah State University Press, 2018.

Smitherman, Geneva. *Talkin that Talk: Language, Culture, and Education in African America*, Routledge, 2000.

Söderstedt, Torbjörn. "Chicken," [programming language], Esolangs Wiki, https://esolangs.org/wiki/Chicken. Accessed 26 May 2019.

"Stack overflow." *Wikipedia*. 6 March 2017. Web. 24 March 2017.

Tempkin, Daniel. "Interview with Eric S. Raymond." *Esoteric.Codes*, 6 Oct 2015, https://esoteric.codes/blog/interview-with-eric-s-raymond. Accessed 6 May 2019.

"Timeline of esoteric programming languages." *Esolangs Wiki*, 25 June 2018, esolangs.org/wiki/Timeline_of_esoteric_programming_languages. Accessed 10 Sept. 2018.

Uberti, David. "Facebook wants you to think it's just a platform. It's not." *Columbia Journalism Review*, 11 May 2016, https://www.cjr.org/innovations/in_at_least_one_respect.php. Accessed 5 May 2019

Vee, Annette. *Coding Literacy: How Computer Programming is Changing Writing*. MIT Press, 2017.

---. "Coding Values," *enculturation*, 6 Oct. 2012, http://enculturation.net/node/5268. Accessed 26

May 2019.

Vee, Annette and James J. Brown, Jr., eds. “Rhetoric and Computation.” Special Issue of *Computational Culture*, vol. 5, 2016, http://computationalculture.net/issue-five/. Accessed 10 Sept. 2018.

Weheliye, Alexander G. *Habeas Viscus: Racializing Assemblages, Biopolitics, and Black Feminist Theories of the Human*, Duke University Press, 2014.

Weisman, Leslie Kanes. “Women’s Environmental Rights: A Manifesto.” *Gender Space Architecture: An Interdisciplinary Introduction*. Jane Rendell, Barbara Penner, and Ian Border, eds. New York: Routledge, 2000.

Weizenbaum, Joseph. *Computer Power and Human Reason: From Judgment to Calculation.* W.H. Freeman, 1976.

Woods, Donald R., James M. Lyon, Louis Howell, and Eric S. Raymond. “The INTERCAL Programming Language Revised Reference Manual,” https://www.muppetlabs.com/~breadbox/intercal-man/. Accessed 5 May 2019.

Young, Iris Marion. “Throwing Like a Girl: A Phenomenology of Feminine Body Comportment Motility and Spatiality.” *Human Studies* 3.2 (Apr. 1980): 137-156. Web. *JSTOR*. 12 March 2017.