AESTHETICS OF CODE

Cynthia Li

Submitted to the Department of English of Pomona College in partial fulfillment of the requirements of the Senior Exercise

Spring 2023

Faculty Readers:
Kara Wittman
Joseph C. Osborn

acknowledgements

To my readers and the faculty who listened to and egged on my multidisciplinary rambling: Kara Wittman, Joe Osborn, Lucas Bang, stef torralba. I often feel that my mixture of backgrounds is unwelcome and alien. Thank you for letting me talk about technology in literature classes and critical theory in CS classes, allowing me to make a mess of your various departments, and encouraging the weird mishmash of knowledges this thesis indulges in. Prof O, here’s to many more games of Turf War. Woomy.

To my kid sister Carolyn, no matter how sporadic our Facetimes get, even though you’ll probably never read this. Thanks for all the duck videos.

To my dear friends, in Claremont and farther away; in particular, Alyssa Zhang, Matt Gonzo, Caroline Kim, E. Stultz, Héctor, the Pals of Delight, the Microwave, the splort zone. Your company has been much cherished throughout this past year. I owe you everything.

table of contents

preface: In defense of code

This is a thesis about code style: about reading code like literature for aesthetic enjoyment, about how we write code, and about what decisions go into a judgement of its quality. It's a somewhat niche topic for an English thesis. I've lost track of the number of people who ask me if I'm writing it for a computer science major, or get confused about what I'm studying. I tend to work between these two disciplines, programming with the tools and vocabulary CS gives me while using the language and theory my literature background trains me in to deepen my understanding of those processes. I find that to my peers in literature or other humanities fields, computer science is scoffed at or considered utterly incomprehensible, while to my peers in computer science, critique is unnecessary and excessive. But with the proliferation of tech—with ChatGPT purporting to mimic human speech, with online ads and tracking, with the turn to Zoom and remote work spurred by COVID-19—we can no longer afford to treat computers as incomprehensible black boxes.

It's increasingly valuable to know how computers work in the modern day. A myriad of factors push us toward fluency in the language of machines. We're asked to learn to code by White House initiatives like Computer Science for All, kindergarten curricula, and economic pressures that suggest software engineering as one of a shrinking number of jobs that might offer financial security.[1] Annette Vee notes the way the logics of programming are swallowing everything around it; computers today are ubiquitous.[2] We send emails, type in Word documents, and browse the Internet daily. Computer literacy is no longer a niche skill, siloed into computer science classes. We use computers for data processing, conducting surveys, creating graphs, searching PDFs, accessing archives. Enrollment in computer science classes has skyrocketed

[1] Vee, "Introduction"

[2] Vee, ibid.

over the past decade.[3] And it's not just software developers learning to code—tools used in programming are relevant in other fields as well. Computer science is growing to encompass a wide variety of cross-disciplinary interests, generating different ways of writing code to serve their purposes, not just the very particular needs of software. Conversing with machines is quickly becoming a necessary skill applied in a wide variety of ways. More than just talking to machines, software is worth taking the time to understand in full, both the effects it causes when run, and the conditions of its production.

We're increasingly positioning code literacy alongside textual, financial, and social literacies. The term "literacy" stresses an obligation to learn that skill. To have code literacy is to be a better, moral citizen, capable of navigating modern life.[4] Reading and writing code is not just a fun hobby or a professional skill—it's rapidly becoming an everyday necessity. Literacy as a term was first used to refer to the ability to read and write, arising in the 1700's, as printed texts were disseminated and widely spread expecting reading.[5] As texts were written and dispersed with quill, then the press, then typewriters, then telegrams and telephones, writing (and by extension literacy) continues to be linked to technology. This makes the jump from literacy-as-technological to coding-as-literacy a smaller one than necessarily expected. Analogies to computer literacy and reading/writing literacy have consistently been made over the history of computing pedagogy. Vee in particular notes an analogy from John Kemeny, who argues that computing should exist across the curriculum just as writing should.[6] These practices are repeated rhetorically linked, but their methodologies remain distinct. However, if we're to

[3] Camp et al., "Generation CS."

[4] Vee, "Introduction."

[5] Vee, ibid.

[6] Vee, ibid.

introduce programming to the *whole* curriculum, letting code touch various other ways of working with information, perhaps it's inevitable that these practices should collide. Writing in computer science is very different from writing in literature and critical theory. Similarly, computation will change how we do literary work, and literary fields will understand computers differently, rooting code in a very different context and tradition of knowledge production.

As is the nature of cross-disciplinary work, literature and computer science have wildly different vocabularies to describe their respective objects of study. Here, I define some programming terms that I'll be using in this thesis. These definitions are not set in stone; some of them are debated and used flexibly across computer science. However, I'm setting down these definitions as a guideline for how I'm engaging with code throughout this thesis.

Code itself is a way for humans to communicate with machines. It's used to express a way of completing a task using the language of the computer. We create tools that facilitate that communication with the computer, as well as human readers of our code. Those tools are called programming languages. I use a more flexible definition of "programming language" than usual here—a programming language, here, is *a set of rules that dictate how a computer works*. Programming languages also detail what rules code has to follow in order to be understood, or "parsed," by the machine: a semicolon at the end of the line, how functions are called, how whitespace is handled, how indentation should work. Some of those rules are mostly arbitrary, primarily chosen for aesthetic purposes; others reflect core principles of the programming language in question. An *algorithm* is a series of steps for completing a particular computational task, sometimes described in code, but often also in natural language. The communications produced by programming languages, combined in a file to do something, are called "programs." I use this term somewhat interchangeably with "code," though "code" might refer to shorter snippets of programmatic language, while programs are more cohesive documents. I also consider “program” more appropriate for code that’s being *run*, doing something on a machine. Your web browser, file browsers, Powerpoint, and your operating system itself are all programs. But if you opened up the files those programs are stored in (“Firefox.exe,” for example), you would see very little text you could recognize as code. The act of translating programs from a human-readable form to a machine-readable one is called compilation.[7] And finally, when we run a machine-readable program, code is executed—it’s done something to the machine it’s been run on.

Ultimately, computer programs *describe ways of doing things*. I am interested in the nitty-gritty of how we make choices about expressing various series of operations beyond so-called “pure” functionality. Fields like rhetorical code studies and critical code studies look at how code makes arguments to its users, presenting particular worldviews to be encoded in algorithms. In “Rhetorical Code Studies,” to describe the scope of his work, Kevin Brock points specifically to “the set of rhetorical qualities and capacities of code,” “the discourse surrounding the development and use of code,” and “the set of social, cultural, and historical contexts that facilitate its composition, dissemination, and critique” as objects of study.[8] However, while style and rhetoric are linked, I care less about the aspects Brock describes than the more minute details code carries with it. Why indent here? why use this language, paradigm, data structure *in particular?* These choices are rhetorical ones at times, but they also reveal understandings of code that aren’t only about its function but also its aesthetics—not just code, but *good* code.

This approach to code style is also distinct from the notion of stylish machines. Many people ask me about large language models (LLM) like ChatGPT, which are trained on massive amounts of data and spit out text that tries to sound human, when I tell them about this work.

[7] This is a simplified version of this process, but it’s detailed enough for our purposes here.

[8] Brock, *Rhetorical Code Studies*.

However, I'm writing less about machines that speak than the humans who write code that *makes* them speak. I'm interested more in the choices of humans than the choices of a massive program that attempts to mimic human speech. While the text LLMs generate is derived from human writing, these models are more accurately huge probability machines than writers of anything *intentional* or *usable*. Allison Parrish goes so far as to claim that "language models can only write poetry": their output can never be considered a speech act.[9] She gives the example of an AI outputting the text "class dismissed" during a lesson—that utterance cannot dismiss the class, putting it into a genre of text that does nothing, just as poetry does.

In contrast, human-written code almost always does something. I'm interested in the poetics of text that *does* do work. I want to bring to attention the programs that cause ChatGPT to run, the people who wrote scripts to scrape the web for data to train that program on, the lines of code where someone decided: *this* is how exactly I'll describe the algorithmic steps in my head in the language of the machine. *Here* are the particular codes I'll use to say what I mean, and *here's* how I imagine the machine to work—do you understand? I prefer thinking about the materiality of code's production, of the ultimately human labor that makes these machines run.

Despite my best efforts, these terms might seem foreign and unfamiliar. In "On 'Sourcery,' or Code as Fetish," Wendy Hui Kyong Chun notes that while learning to read programs "supposedly enables pure understanding and freedom," but also that "tellingly, this move to source code has hardly deprived programmers of their priest-like/wizard status."[10] From a non-computer science perspective, this description of "wizardry" may seem intimidating. My tech skills are often viewed as some sort of magic by my peers in the humanities. This thesis is

[9] Parrish, "Language Models Can Only Write Poetry."

[10] Chun, "On 'Sourcery,' or Code as Fetish."

written for both a technical and critical audience (primarily the latter)—if you're reading from one field and not the other, there may be unfamiliar terminology, modes of writing, languages.

The approach I'm taking is less about *perfectly* understanding what a program does, but about the *experience* of writing, reading, and otherwise interfacing with code. micha cárdenas—in her book theorizing trans of color poetics, *Poetic Operations*—points out that

> to understand algorithms, you do not need to be a programmer. You can also understand an algorithm as a recipe. A recipe has ingredients and steps, just as an algorithm has variables and instructions. Think of the algorithm for cooking chicken: get the chicken, oil, spices, and a pan. Preheat the oven. Oil the pan. Put the chicken in the pan. Spice the chicken. Put the pan in the oven.[11]

Not all structures in programming might map cleanly onto this analogy, but the idea's the same: programs offer a language through which we can describe how problems should be solved. Sometimes that language is catered to technical specificities of computer work, digging into memory safety or multithreading, but much of the time it's more like a choose-your-own-adventure book. We follow recipes, we knit, we give out instructions, we use spreadsheets. Nowadays, we even interact with computers daily out of necessity. These modes of encoding information are ways into understanding coding *for* computers. We are all familiar with programs and algorithms. When I talk to friends who don't study computers about code, I find that it isn't as mysterious and enigmatic as they may first seem. It may take more effort to understand if you're not used to the language and styles of both these disciplines; for this I ask your patience. I hope to hold both my knowledges together, letting them work with and on each other. As they settle, necessarily there will be debris. Please pardon the construction.

[11] cárdenas, *Poetic Operations*.

chapter 1: What *is* style?

For poetry makes nothing happen:

W. H. Auden, *In Memory of W. B. Yeats*

Decoding a poem is not always easy, its meaning often remaining elusive.
Poem-ing code to the contrary goes automatically, like magic.

Jan de Weille, “This Code = this code”

cárdenas makes programming into a cooking analogy, but we *already* talk about how we manipulate computers through a metaphor of language. I’ve already casually invoked this metaphor: we “write” using programming “languages,” which have “syntax” and “grammar.” The language we speak of is an abstraction over the manipulation of silicon and electricity that happens at a hardware level of computing machines. Sitting at a keyboard and typing into a text editor is far from the only way of producing programs, but it’s the mode that decades of computer science (and longer, of the printed word) have decided should be dominant. Ian Arawjo traces the way we construct the act of programming through the interfaces of type and computers.[12] We consider programming a linguistic endeavor because of a long history of technologies that bind writing and computation together. In this thesis I seek to push this relationship as far as possible, treating code as a written object just as we look at prose or poetry. But code is a different genre from both those categories. It’s text that *works*, that when run, produces some output, pushes bits around on a machine, or makes *something* happen—send an email, tick up numbers in a bank account, suggest the most likely next set of words.

Of course, code differs formally—in multiple senses of the word—from other mediums of communication. In literary terms, code takes on different shapes than those seen in prose or

[12] Arawjo, “To Write Code”

poetry, abiding by different organizational rules. Where prose has its paragraphs and poetry has stanzas, code sections itself off into blocks marked by indentation or brackets. Computer science uses “formal” to mean something that can be precisely described to produce the exact same result over and over. Often, this means defining a vocabulary in a very particular way that can be mathematically understood and repeated, rather than in one that matches up with the fuzzy intricacies of natural language. Code written according to a set of rules which can be described formally. We can talk about the form of poetry (using terminology to describe rhyme scemes, meter, alliteration, etc), but code’s formal regulations are much stricter. Poets can break or bend the rules surrounding a poetic form. In contrast, programmers *must* follow the rules set down by the computer or their code will not run, unable to be understood by the machine. However, that doesn’t mean we have to set it apart from poetry and prose entirely.

Treating code as akin to literary writing allows us to imagine it as a creative medium, not just one that’s meant to perform labor. It gives us room to consider aesthetics that flower above the purely functional. Other modes of programming *do* exist beyond written ones, such as drag-and-drop languages like Scratch, using hand gestures when playing Wii Sports using the handheld remote, or even simply navigating a computer with a mouse and keyboard. These three interfaces all are ways for humans to communicate with machines—they give a computer some inputs with intent behind them, and the computer reacts. The fact that code is written rather than gestured, played, or drawn is not a coincidence. There’s a history here: Western writing and print strongly influence the way computers are designed. ASCII characters, the tiny subset of characters meant to handle the Latin alphabet and punctuation, are most highly prioritized in computing and in digital software in general. Western print assumes that text is written and read from left to right, that characters will not be modified after being printed by cursive or ligatures, that words are separated by spaces. These assumptions are built into a great deal of modern software, which goes on to treat Latin-alphabet text as first class. We use English as the lingua franca of code solely because of colonialist technological precedent. But surprisingly, despite this, there isn't a strict formal definition for what counts as a programming language or code in computer science. The one I'm using is purposefully broad, but because this thesis focuses specifically on the aesthetics of language, I'm limiting the main focus of this project to programs that are readable as text.

I also focus mostly on code that's written, rather than what code in its compiled form, when it's run—although the two cannot be reliably untangled from each other. What's intelligible to machines is often incomprehensible to us, and vice versa; code straddles the two. Code never stands alone. A piece of text being called code implies its compiled form, where it's actually being run, or an intent to *do something* on a machine. However, the act of expressing what a programmer imagines to be the solution to a problem, then putting that on paper—or typing it into a digital document, at least—is *distinct* from what happens when a program is run.[13] Chun writes that "the goal of software is to conflate an event with a written command," but code cannot be mapped one-to-one with a running program, which operates in an operating system running many other processes and interacts with physical hardware and electric signals on the machine.[14] By focusing on code's primarily human-facing, I'm primarily concerned with the way code is read and understood by the people who interact with it. The effects that code has *on* people when run is valuable to critique, but is simply not what I am interested in here.

***

If coding is about writing, then critical work is arguably about reading, looking at what writing does at various scales, at the level of the word, sentence, paragraph, chapter, novel, literary

[13] Chun, "On 'Sourcery,' or Code as Fetish."

[14] Chun, ibid.

movement. Coding has units, too: statements, functions, programs, libraries. We can read at all these levels, but we can start with a program. This one's written in Python:

```
flavor = input("what's your favorite ice cream flavor? ")
if flavor is "chocolate":
    print("that's mine too!!")
else:
    print("mine is chocolate, but I'm sure that's delicious too.")
```

The same program in a different language, Java, might look like this:

```
import java.util.Scanner;

class Main {
    public static void main(String args[]) {
        System.out.println("what's your favorite ice cream flavor? ");
        Scanner s = new Scanner(System.in);
        String flavor = s.nextLine();
        if (flavor.equals("chocolate")) {
            System.out.println("that's mine too!!");
        } else {
            System.out.println("mine is chocolate, but I'm sure that's delicious too.");
        }
    }
}
```

Both these programs use the same steps: they print the text "what's your favorite ice cream flavor?", read a line of user input, then check if the answer was "chocolate." If so, it tells you "that's mine too!!", or "mine is chocolate, but I'm sure that's delicious too" otherwise. The alternate paths that hinge on the answer of the question are described through an if statement, marked by the keywords "if" and "else" above. If we wanted these lines of code to repeat—for example, if we wanted to ask "what is the best ice cream flavor?", refusing to let the user continue until they answered "chocolate"—we would use a loop, returning the program to the first line of the code inside the code block until a particular condition is reached:

```
while input("what's the best ice cream flavor? ") is not "chocolate":
    print("wrong.")
print("it sure is!")
```

Here, we read the user input and check if the answer is “chocolate” or not. If not, we print “wrong.” then ask for input and check the condition again. If so, we exit the loop, and print “it sure is!”.

Dictating the conditions under which particular lines of code should be run is a core part of programming, called “control flow.” Without control flow, programs would only be able to execute a bunch of commands in strict sequence, then exit—but we need programs that keep running until asked to close and that can reason about varied conditions. Almost every programming language has ways of manipulating control flow, and certainly every commonly used one has if statements and loops, at the *very* least. These are the structures that build up the forms of programs. Through these building blocks, we’re able to more easily break apart the pieces of what these programs are doing. We can identify patterns running through code by looking at the core pieces of control flow.

I return to the first two code snippets juxtaposing Python and Java. While both have the same structure, the Java program is significantly longer. Whole lines are saved just for a single bracket (lines 12–14). There are significantly more brackets, too, boasting four pairs of curly braces where the Python program has none. The condition of the if statement on line 8 is also wrapped in parentheses, while the equivalent line (2) in the Python version has no such punctuation. And, of course, the Python program doesn’t have semicolons at all, while the Java version ends each statement with a semicolon (lines 5–7, 9, 11).

These differences in punctuation between the languages exist because of their respective priorities and the decisions made when designing them. Python seeks to be beginner-friendly, often mimicking natural language. To meet expanding needs for computing, many modern programming languages are designed to be more and more like natural language in hopes that people will be able to more easily learn them. Python uses keywords like is, not, and, and or as alternatives to ==, !, &&, and || respectively. It also avoids curly braces and parentheses, using whitespace to organize code blocks instead rather than the curly brackets that Java does. This means programs written in Python are often shorter and less cryptic, potentially making them more readable. But readability is not *natural*, but a quality of text created by the aesthetic expectations we consider normal. That “normalcy” must be taught, through practice and exposure to standards of what we consider “clear.”

However, Java was made for a different purpose. On line 6 we see a statement that doesn’t seem to have an equivalent in our Python program, “Scanner s = new Scanner(System.in);”. This line creates a “Scanner”, which allows us to read input from “System.in”—or, the characters a user is typing. It turns out that reading user input in a program is more complicated than we might initially think. Python, with its beginner-friendliness in mind, wants you to be able to do this quickly and easily, and lets you invoke “input” instead, skating over the nitty-gritty of how exactly it tells the program what keys you’re tapping on your keyboard (line 1). It’s such a common task that novice programmers in particular are asked to do that Python wants you to be able to do it easily. Java’s version of the same task instead points to a different priority: it introduces one of its core aspects, the object-oriented paradigm (OOP). OOP describes a world by filling it with various objects, which hold their own data and interact with and act on each other. In Java, we might say that a “Car” object has a brand, color, and license plate number, and that someone could give it a new paint job, changing its color. Here, a scanner is an object that reads from a stream of characters, which in this case is the user’s input, and “s” is the name of the particular scanner we’ve created. We can ask the scanner to get the next line of characters with “s.nextLine()” on line 7. Python, again, doesn’t reveal any of this to the programmer.

We can also see that the "System.out" portions of lines 5, 9, and 11 echo the "System.in" on line 6. If "in" is the user's input being read by the machine, "out" is the machine's output written out for the user. Java hints at this dichotomy of input/output ("I/O") and reading/writing. I/O extends beyond reading user inputs to tasks like accessing and editing the contents of files. As the abbreviation suggests, it's another action that computers have to execute frequently. In this particular case, however, Python doesn't make this connection, opting for something more easily usable at the cost of obscuring the computer's model of its internal functions, and relinquishing more fine grained control. These programs might do the same thing, but differ because of the programming languages I wrote them in. That choice of programming language partially dictates what aspects of this program are emphasized, and what others are handwaved away. This economy of attention means some aesthetics, as well as practical and rhetorical functions, of code will be easier to achieve than others. Writers may push the boundaries of this syntax, but ultimately, those restrictions are something you *want* for your project.

These two examples are simple, but just changing the language they're written in reveals various priorities of their respective languages by showing off what's easy or natural to write. In *Exercises in Programming Style*, Cristina Vidiera Lopes explores how these priorities are expressed through code by rewriting the same programmatic task over and over.[15] Unlike the variants I show above, she *only* writes in Python, revealing a wide array of styles that a single programming language can be wrangled into expressing. Lopes offers a cursory definition of style as writing under constraints, however in- or explicit those might be. For example, the English language itself provides us with a set of affordances and grammars, however unnoticed we might find them. On the other end of the spectrum, Oulipian writers list out their constraints

[15] Lopes, *Exercises in Programming Style*.

very explicitly, writing within strictly-set rules: don't use the letter e, the only vowel you're allowed is "a," rewrite the same story 99 times. It's from this tradition that Lopes writes her exercises. She *explicitly* links coding to writing prose.

Following Lopes, I focus on style in computational writing specifically because code is written by humans. Our modes of expression are anything *but* binary, and chafe against the limited vocabulary and syntax of code. Therefore, to squeeze natural language into the strict rules of programming languages means we eschew much of the variety that lets our voices shine. However, we also manage to reach for a multitude of methods of self-expression (as Lopes does) *within* those restrictions—ones that are purely aesthetic. When a program is *run*, the stylistic and aesthetic choices a programmer made when writing that code practically disappear in the executed output, swallowed to make sense for the machine, which can't parse most human-written code directly. Machines may understand subtle differences between stylistic choices at the machine language level, but those differences are all-but invisible to people viewing the effects of that code, unable to perceive differences on the *tiny* scale of the computer. Style, then, becomes a mostly human-facing element of code, meant to be read, not run. Therefore, before discussing style in code, I want to first offer an overview of writing about literary style.

***

First, perhaps the more widely utilized sense of the word "style" refers to a set of grammatical conventions for authors, usually for consistency's sake. In *The Elements of Style*, William Strunk Jr. and E. B. White advocate for writing that minimizes authorial voice and pushes clarity and concision over flair.[16] The book is a list of maxims for what "proper" writing is, offering a

[16] Strunk and White, *The Elements of Style*.

particularly prescriptive and closed view into the English language. Despite this, White does spend a chapter on writing style in the more literary sense. He repeatedly calls this chapter "a mystery," unable to articulate exactly what style is and going so far as to suggest that there's "no satisfactory explanation of style."[17] Style remains out of reach, unable to be defined and actively *escaping* definition—White describes its rules as "disturbingly in motion."

The book offers some suggestions for what good literary style is, focusing on the sound and syntax of text. The former is centered around sounding natural and fluid. White rails against the clunky and overthought; in fact, simply writing with too much attention to style is detrimental to a piece of writing. Style has to come without thinking—content comes first, and style rises from it unbidden. Still, the sonorous quality of style is the only reason White seems to allow breaking grammar rules, insisting that "the question of ear is vital."[18] Style is a product of comfort with spoken language as well as written, and that speech should be natural, correct, and *fluent*. Like the sound of written words, syntax isn't meant to stand out. White declares that "…when you become hopelessly mired in a sentence, it is best to start fresh; do not try to fight against the terrible odds of syntax."[19] Grammar must be battled to come out with your intended meaning, and content and structure seem at odds with each other.

White privileges the former over the latter. For him, the writer's ideas are more important than the material conditions of their writing. This separation from context refuses to engage with the author alongside their text, as though all writers operate from the same position. The English language itself presents an epistemology, however invisible and embedded in a so-called universal standard. Yet standardized grammar is a relatively recent invention, grammar books

[17] Strunk and White, ibid.

[18] Strunk and White, ibid.

[19] Strunk and White, ibid.

only really arising in the late 1700's, which produced a slew of "standards" for grammar that caused confusion as rules conflicted.[20] Today, even under a purportedly "standard" set of rules for grammar, language that disrupts standard American English then becomes a site for alternate, non-standard styles—for immigrants, speakers of African American Vernacular English, and speakers of the many dialects and creoles of English around the world. Style is produced not just by fluent speakers of a language but by those who talk outside of a standard.

Strunk and White emphasize briskness and practicality in formal English writing. But while clarity is valuable, it's not necessarily the end-all, be-all of style. Style extends beyond grammatical "correctness" and Strunk and White's insistence on "clarity" above all else. In "The Aesthetic Structure of the Sentence," William Gass posits that style is in part created by the distance between words through formal elements—grammar and syntax.[21] He asks, how many syllables, words, clauses stand between two elements of a sentence? Sentences then become a web of positions, and as such, relations, since something cannot be situated in a place without the context of what surrounds it. Gass notes, however, that "'The man at the door was an encyclopedia salesman' and 'The dog at the door was a Doberman pincher' have the same grammatical form as 'The flea on the dog was a nervous Nellie.'" In other words, form can't be the whole story. He points, also, to connotation, sound, rhythm, and meter as aspects of style. These aspects concern themselves with the sensoral experience of these words, of affect and mouth shapes and auditory sensation.

Notably, neither of these are focused solely on content, on meanings, yet are aspects of how style is decided nevertheless. Roman Jakobson points out that the repetition of rhyme and meter (as well as the breaking from established patterns) draws attention to meaning, suggesting

[20] Watson, "Introduction."

[21] Gass, "The Aesthetic Structure of the Sentence."

a link between particular words. He writes that the imposition of formal properties on the meaning of a sentence “gives the experience of a double, ambiguous shape to anyone who is familiar with the given language and with verse.”[22] Style exceeds meaning and content, spilling over into meaning. They reveal what the priorities the authors had, and what they wanted to draw attention to, subtly coloring the flavor of an argument. From the field of anthropology and art history, Meyer Schapiro describes style as a formal quality of creative work rather than being tied to a particular material, technique, or subject matter.[23]

Notions and norms of style in English cross-pollinate into and influence standards in programming, even though programs are notably distinct from prose (which I discuss in chapter 2). One way of understanding code style mimics Strunk and White in describing conventions that seek to make code understandable and maintainable. Understanding and clarity are emphasized even more strongly in programming style guides like *The Elements of Java Style*, a book for writing in the popular programming language Java, mimicking the prescriptive and inflexible approach to language Strunk and White lay out.[24] Some of these rules are specific to the particular features and syntax of Java, and others are more language-agnostic. The much more limited scope of Java’s semantics compared to natural language means that some of the advice in this book is only applicable when writing this language, not necessarily programming as a whole.

As this style guide reveals, not all code that gets you a “correct” result is the best code. Code style tends toward usability and practicality, concerned with how fast and accurately code solves a problem, or how well readers can understand it. Writing code requires constantly making tradeoffs about how a problem should be expressed such that it can be solved, and that

[22] Jakobson, “Closing Statement.”

[23] Schapiro, “Style.”

[24] Vermeulen et al., *The Elements of Java Style*.

the computer can accurately interpret its instructions. These goals are common in industrial contexts, where code must be used at scale, and wide-ranging user experience and runtime are crucial.

A strict notion of style may be backed up by style guides, but there's still wiggle room in code for personal preference. If we look at pedagogical takes on what makes a piece of writing "good," we'll find inconsistencies. Patrick Sullivan finds that different instructors can give papers wildly different grades depending on their own criteria for "good" writing.[25] In fact, he quotes Pat Belanoff, who goes so far as to call these differing expectations for writing "a sign of strength, of the life and vitality of words and the exchange of words." Similarly, code style is not set in stone, and a programmer might prefer specific styles of organizing ideas and formatting their code that another might hate. The code styles that Lopes lists demand attention in different aspects of the coding process, from memory management to error handling to functional programming styles to succinctness.[26] Each style she chooses makes us perceive the resulting program in a different way. Style, then, offers a way into discussing code beyond "correctness." Code becomes a text that does work even when a program isn't running. To study code style is to observe the way we present our ideas to machines, but just as importantly, each other.

[25] Sullivan, "An Essential Question."

[26] Lopes, *Exercises in Programming Style*.

chapter 2: Code as relation

If Lopes describes programming styles as restraints on code writing, she gives less attention the question of how these styles and restrictions start to arise. I'm interested in this question because stylistic choices have the power to attract and push away certain writers, to align with rhetorical and aesthetic movements. Meyer Schapiro points out that in art history, style is a "criterion of the date and place of origin of works, and as a means of tracing relationships between schools of art."[27] If style is relational, then one way it might matter is through the different movements and understandings of code it stands in for.

As a case study, I look at the Rust programming language, a relatively new systems programming language focused on memory safety and speed—two areas of concern that often seem at odds with each other when coding, a contradiction I'll touch on later. In recent years, Rust has seen wider adoption and gained a higher profile,[28] and for the past seven years, it's consistently maintained the top spot on Stack Overflow's most-loved languages list.[29] This well-loved status comes from its particular priorities, but also the support that the core language team offers to coders. Rust has easy-to-read error messages built into the compiler, detailed documentation, and a wide suite of tools that make writing the language easier.

A programming language is part of a rich, interconnected ecosystem of tools that support writing code. This includes the compilers that do the actual conversion of code to machine instructions, but also other programs used in the code-writing process. Linters and code analyzers detect errors ahead of compilation and suggest idiomatic code, or, code that follows the conventions of the language. We previously touched on how programs may contain *valid* code

[27] Schapiro, "Style."

[28] "Rust Survey 2021 Results."

[29] "Stack Overflow Developer Survey 2022."

that the compiler understands, but that more factors than validity and correctness affect what makes a piece of code "good." Above the rules of syntax lie a fuzzier communion of norms produced by the people who write a particular programming language; many of these norms are supported and distributed through tooling. These tools are built into code editors, distributed on the Internet, and designed and discussed by programmers.

I choose Rust specifically because it's a language with a notoriously high difficulty curve.[30] Rust is well known for its commitment to the memory safety of languages like Java and Python *as well as* the speed of C. C puts the burden of making sure programs are safe on the programmer rather than using techniques under the hood to make sure the program will *always* be safe. By "safety," I mean making sure a computer's memory isn't read and written to where it shouldn't be. If we imagine computational memory as a paper, reading and writing to memory would be analogous to finding a space to put a new sketch, or looking for a specific drawing. The computer on its own has no idea what memory belongs to what data: the paper doesn't know where one drawing ends and the next begins. Safety means keeping each drawing separate, making sure old drawings aren't covered up by new material, and ensuring that we don't consider two separate sketches the same one. In C, where speed is the priority, it's up to the programmer to calculate ahead of time how memory should be allocated and freed—determining what pieces of the page are taken up by what, and sticking to those boundaries. Unfortunately, memory doesn't have a graphical representation as a physical paper does, making it difficult to reason about, and very easy to get wrong. Java and Python, on the other hand, handle memory management in the background, taking on the responsibility of safety. These methods often lag behind C in execution speed because they have to be monitored continuously while the program

[30] It's also one of my personal favorite languages.

still is runnning. C doesn't carry out these checks at runtime, assuming that the programmer's taken care of it—at the cost of needing to be *extremely* careful about what code you write, risking a wide variety of errors that safe languages don't need to worry about at all.

In chapter 1, we saw a comparison of Java and Python, noting that Python handwaves concepts in exchange for user-friendliness. This dichotomy has effects on what materially happens when we run code. Usually at small scales, the effects are negligible, but when scaled up, languages with higher levels of abstraction are often slower and use more energy than other, lower-level languages.[31] Abstraction obscures technical details, often to make that language easier to write, but it also makes it more difficult for a computer to understand, taking longer and more energy to run. Rust, however, wants both safety *and* speed. The safety guarantees Rust provides means that many of its features are unique to the language, and are often difficult to internalize for people just picking it up. For example, all variables in Rust cannot be modified by default, requiring the keyword mut to allow them to be modified. In general-purpose languages, the ability to change your data after you declare them is usually taken for granted. Immutability is a common quality of functional programming languages, which are often more esoteric and the domain of academics and type theorists rather than software engineers who ship software products. Rust beginners also often struggle with the borrow checker, or borrowck, a concept that simply doesn't exist in other programming languages. Other languages wholly lack the vocabulary for some of Rust's most central concepts, which in some ways evens the field for all of its learners: everyone has to wrestle with learning how references and ownership work, regardless of how much experience they have.

[31] Pereira et al., "Energy Efficiency Across Programming Languages."

Rust uses references as fancy pointers that allow memory safety to be reasoned about at compile time. borrowck ensures that in safe Rust, all data has no references pointing to it, any number of immutable references pointing to it, *or* only one mutable reference. It imposes language-level rules to reason that a program is safe before any code is even run. This decision stands in stark contrast from languages like C, Java, or Python, which are all popular beginner languages taught in introductory programming classes.[32] Rust trades familiarity with other languages' semantic norms for an effort to ensure its goals. It doesn't shy away from difficulty of learning, and insists on making programmers aware of work that's abstracted away in higher-level languages, and difficult to manage in lower-level ones. Rather than emphasizing ease of learning or speed of writing, Rust asks its learners to slow down and invites them to engage with what's going on under the hood. In a world focused on constant forward progress, languages that value a slower cycle of editing and reflection are rare. Rust is much stricter than Python, which is a popular beginner language; many of the things Rust considers compiler errors are accepted by Python. Rust simply *won't* accept some programs that are perfectly viable in the latter language. This means writing Python is faster—which is useful in many circumstances—but also liable to host a much wider variety of bugs. In contrast, Rust values correctness and careful consideration over speed. Language design is a commitment to a certain kind of pedagogy and ideology, providing a model of how a language imagines the workings of a computer.

Because of these differences, Rust is often regarded as difficult to learn, with a high learning curve that newcomers often struggle with. Rust doesn't settle for compromises on its goals—instead, it chooses to offer tooling to make it easier to understand. These tools are just as

[32] Becker and Fitzpatrick, "What Do CS1 Syllabi Reveal about Our Expectations of Introductory Programming Students?"

critical as the language's syntax and features itself. The compiler's design is pedagogical as much as technical, with a focus on how compilation errors are presented to those writing in the language. Rust's difficulty means that much of the time, people writing it will be faced with a long list of compiler errors that they need to parse, understand, and fix to make rustc accept their program. Rust makes an effort to make these errors as readable as possible, and its errors are one of the language's most attractive features.[33]

As a quick example, here's some code that won't compile:

```
let x = 3;
println!("x = {x}");
x = 4;
```

Attempting to run it would net you this error:

```
error[E0384]: cannot assign twice to immutable variable `x`
 --> src/main.rs:4:5
  |
2 |     let x = 3;
  |         -
  |         |
  |         first assignment to `x`
  |         help: consider making this binding mutable: `mut x`
3 |     println!("x = {x}");
4 |     x = 6;
  |     ^^^^^ cannot assign twice to immutable variable

For more information about this error, try `rustc --explain E0384`.
```

This rustc output is much more verbose than expected from most programming languages. It also offers help text to the proramer, suggesting the addition of the mut keyword, which resolves the error. Running the suggested "`rustc --explain E0384`" also presents more information about the error. In contrast, a similar error in Java would look like this:

```
Test.java:6: error: cannot assign a value to final variable x
        x = 6;
        ^
1 error
error: compilation failed
```

[33] "Rust Survey 2021 Results."

This is a much shorter error message, showing only one line of code and a terse description of the error. Rust's output takes up space, giving more context and pointing to lines of code where those errors are likely to have occurred. Many programming languages are notorious for giving particularly opaque errors that novices often struggle to understand. The language of compiler errors is another code that must be learned when programming. With experience, one grows to recognize what certain error messages mean, but until then, error messages tend to be left completely unread because of how little light they shed on what might be wrong with a program. While notoriously dense errors are the norm in other languages, Rust's compilation errors strive to be helpful above all else. They seek to not just point out what's wrong, but provide searchable tools and keywords through which a programmer can find further resources.

The core "grokloop," as Kate Compton names this loop of creation and feedback,[34] of programming is one of writing, compiling, reading, and editing. In Rust, this loop is shortened by the compiler itself, which gives quick feedback on errors and *how to resolve them*, facilitating the "editing" piece of that cycle. It attempts to not put responsibility on a novice programmer to stumble across understanding via web searching or other resources. rustc's support is inherently attached to the compilation process—ergo, it's nigh-impossible to write Rust code without encountering its input on your programs. It *expects* a programmer to come across errors as part of the coding process. in the process of writing code, Rust plans for difficulty; it takes failure to communicate as a given. Machine and human languages are difficult to reconcile. Rust doesn't shy away from this fact, but instead offers tools to facilitate that communication, from the compiler errors itself to other external tooling (which I'll discuss further below). The compiler's

[34] Compton, "CASUAL CREATORS."

attitude toward coding signals that it wants the programmer to succeed, working toward a mutual understanding of a program on both sides of the parser.

These errors are a pedagogical tool as much as a marker of what needs to be fixed in a program, facilitating my understanding of the computer's systems. I had minimal systems programming knowledge before learning Rust, so my first time writing unsafe code, where the compiler doesn't check for memory safety errors and anything goes, was nerve wracking. But I'd been writing *safe* Rust and engaging with rustc errors for a good while, so the understanding I had of memory management was enough to tide me over. Working with Rust's definitions of memory safety were sufficient enough to teach me the basics of writing unsafe code without its safety rails. Rust gave me the vocabulary and confidence to broach a previously intimidating subject on my own.[35] Still, it's not something I *want* to do regularly—I'm happy to cede this task to the compiler.

Rust attempts to have the best of both low- and high-level worlds, but it sacrifices language complexity to achieve this. Rust takes a long time to understand, and even with a decent knowledge of the language, it still takes me a while to write it. I rarely write Rust without running into errors with the borrow checker, with mutability, with types, with trait objects, and expect the compiler and my tooling to communicate with me when it can't parse my work. Rust challenges the notion that we should understand how everything works—its errors allow pieces of the language to be digested over time, slowly, as familiarity grows with it. Tooling supports a

[35] Crichton, "The Usability of Ownership" notes that Rust is over-cautious about safety, and won't accept all technically-correct programs. There are many programs that are safe nominally, but cannot be reliably proven by the compiler to be safe. borrowck won't always offer an accurate explanation of soundness, which means it's not a perfect teacher of what memory safety means.

lack of knowing: it expects that a programmer will make mistakes, forget, or simply not understand, and points them toward potential solutions.

It's worth noting that the "complexity" here is not necessarily *natural*. Rust's difficulty comes in part from how distinct its expectations are from other languages, which first introduced this abstraction-performance divide. Rust is only alien because programming languages have not historically prioritized or given language to the concepts that Rust has. Rust's tooling, then, acts as a gateway into the paradigm of coding the language prioritizes. The attention paid to the feedback programmers get on their work means that this initial hurdle of learning the language is a priority for the Rust compiler team. Rust is not only its technical language features. It's also, crucially, concerned with the whole community of those who write it, from people comfortable with C and assembly to those who've barely scratched the surface of programming, and offers its companionship at all levels.

Rust's cautiousness encourages a style of careful, thoughtful programming, paying attention to details and particulars. It's this care that drew me to Rust as a non-systems programmer: its guardrails allowed me to build confidence in the correctness of my programs. The compiler itself felt supportive of my code-writing process, guiding my way through the language as a novice. As part of the RustConf 2020 opening keynote, Ashley Williams details the core values of the project, noting in particular the word "empower" in Rust's slogan:

> …so, as we look at this slogan, "a language empowering everyone to build reliable and efficient software," I don't want you to think that we've thrown this "everyone" out… but there's a core of it that is true-er, which is: "we are a language empowering everyone, but especially folks who didn't think systems programming was for them."[36]

[36] Matsakis et al., "RustConf 2020 - Opening Keynote"

Williams goes on to explain how the term “systems programming” actively keeps people away from a language because of the stereotypes surrounding it—of techy wizards siloed away from the world, who mumble incomprehensible tidbits about computers.[37] The people who were *able* to access this language were often rich, white, male, able to access a university education and the technology that allowed them to work with computers regularly. Rust separates itself from this history of systems programming with its tooling, focusing on empowerment of its writers as much as the power of the language. Choosing a programming language is a stylistic choice as much as a practical one, carrying more connotations than just the technical features of the language. It’s a commitment to a particular set of *values*.

***

While rustc’s errors are fairly comprehensive and well-liked, other tools exist to facilitate code writing that don’t come by default with the compiler. Rust has an autoformatter (rustfmt) and language server protocol implementation (rust-analyzer), which format code and allow error messages to continuously show up while coding, making the grokloop of writing code and seeing errors even faster. Rust also has cargo clippy, a more complex linter that checks for style and errors. These tools serve to help programmers write “better” code that aligns with the priorities of Rust as a whole. clippy comes with more than 550 lints, separated into eight categories, listed as: correctness, suspicious, style, complexity, performance, pedantic, nursery, and cargo.[38] Here I’m focusing on the first three as the categories most-often encountered.

Lints in the “correctness” and “suspicious” categories handle code that’s very likely to be wrong or useless, further extending the checking rustc already does and essentially acting as a proofreader or second bug-catcher. Such lints are relegated to a separate, opt-in tool rather than

[37] Chun, “On ‘Sourcery,’ or Code as Fetish.”

[38] “Clippy Lints.”

the compiler directly—the sheer volume of feedback clippy provides could be overwhelming to a novice writer. While seeking to be useful, rustc's errors don't try to find *every* possible error. The language cares about safety and trust, but doesn't want to turn away programmers because of a constant negative feedback loop, either. Rust's power to analyze a program is also finite—there are plenty of situations it can't reason about when guessing at a programmer's intention. Additionally, putting *too many* restrictions on a programmer's style on the compiler side can wildly limit what's expected or possible to be written. Rust is a general purpose language, meant to facilitate a wide variety of use cases, and encoding too many patterns as erroneous can shut off cases that perhaps wanted to be left open. Therefore, clippy's lints are opt-in: it's well-known that its warnings are annoying, and those who use it actively agree to let it read their code. On the flip side, if needed, clippy also provides a smattering of "restriction" lints, which detect patterns that may not be inherently bad, but sometimes helpful in very particular situations.

Furthering the default compiler's role as a pedagogical tool, some lints are less generally applicable to programming at large, but instead provide suggestions for methods in the standard library that work as shorthands for the desired semantics. option_map_or_none, for example, suggests the method `_.and_then(_)` instead of the pattern `_.map_or(None, _)`. The standard library's documentation is lengthy, making it difficult to search for one specific function. Will Crichton, looking at the usability of Rust's memory model, points out that it's unlikely to know every semantic tool you have at your disposal as a new user, especially when a data structure like Vec can have over a hundred associated functions of wildly-varying appropriateness to your task.[39] Searching through all these functions to see which ones fit your task is time-consuming. Moreover, new users may not even be aware that such functions exist—Crichton notes that Rust

[39] Crichton, "The Usability of Ownership."

novices “didn’t think to go looking for a helper function” when writing. Functional programming, one of the programming styles that Rust sypports and uses, thrives off the abstraction of patterns into higher-order functions, but this comes at the cost of needing familiarity with those patterns. Comparing this experience to that of writing proofs in Lean, Crichton (reasonably) complains that “it’s an excruciating experience to carefully scan the hundreds of theorems in the standard library.” clippy aids this process of searching by automatically detecting patterns that can be simplified, actively pointing attention to the functionality of the language provides *while* coding. These errors support programmers who are still learning what Rust has to offer them, as well as programmers who simply don’t have the bandwidth to remember all these functions.

Meanwhile, style lints are especially concerned with idiomatic code, providing lints like `single_match`, which “checks for matches with a single arm where an if let will usually suffice.” This is done to prevent excess nesting, which is generally considered something to avoid in many languages—too much nesting often signals with high complexity, and may be better served by abstracting out into several smaller, easier-to-digest function calls. clippy adheres to these “best practices,” seeking to make code readable and maintainable. Other style lints introduce style conventions common in the Rust community at large. The lint `new_ret_no_self` would show an error if a method for a type named new doesn’t have a return type that includes itself. For example,

```
impl Example {
    fn new() { /* ... */ }
}
```

would trigger this lint, while `fn new() -> Self { /* ... */ }` would not. This provides consistency between other libraries in the Rust ecosystem. clippy has certain expectations that may not be necessary when coding alone, but are necessary the moment programs turn from the individual to a community. As such, using clippy is to opt-in to norms of the Rust community at large, to interact with a set of values that bloom from but aren't enforced by the compiler. It's to recognize Rust as deeply connected with the people who use it. Style is connected to other writers as much as it is an act of personal expression.

These practices become engrained in the code style Rust deems "good" by borrowing norms from other programming communities, and through continued use. As an official tool, clippy produces and maintains these standards for the larger community of Rust programmers. clippy is able to dictate what code is correct for *every programmer that uses it,* actively pointing out code it deems problematic. Like any technology that attempts to mark out deviance, it holds a great deal of power—it's an automated tool that can be utilized to judge if some code is good or bad, with all the consequences that judgement might carry.

Despite being an obstensibly general-purpose language, the norms of Rust don't necessarily match the needs of other coding communities. When working with a friend to define a formal grammar using a Rust enum, she was frustrated because rustc would warn that enum types should have capitalized names, while lowercase names were the stylistic norm for writing formal grammars. The following code:

```
enum Grammar {
    term,
}
```

gives us this warning:

```
warning: variant `term` should have an upper camel case name
 --> src/main.rs:7:5
  |
7 |     term,
  |     ^^^^ help: convert the identifier to upper camel case: `Term`
  |
  = note: `#[warn(non_camel_case_types)]` on by default
```

This is purely aesthetic. Rust capitalizes types to keep this styling uniform across all code written in the language, so it's easy to tell types and functions apart with a glance. Many other languages don't blink an eye at what capitalization scheme you use for variable and function and type names. Rust considers this a warning, not an error—the code still compiles—but it *is* output you have to see when checking error output. Someone who wrote this code would continuously get feedback that it *isn't quite right*, despite matching up with the expectations of a different coding community. Rust simply didn't have that group in mind when setting those capitalization rules. "Good" code style differs wildly across communities. A single style can't work for *every* case, but Rust and clippy attempt to set a baseline that meets *most* needs, for better or worse.

clippy highlights the norms of the Rust *community* as well as the language itself. To turn on clippy errors for a piece of code means you're not just writing Rust for yourself, but for others to read and understand. By using clippy, we're forced to pay attention to how other programmers of Rust write the language, granting us uniformity between our styles so we can make them legible to each other. This commitment to a particular flavor of legibility means we want to be read by others in this community. We *want* to conform to their styles. Because code is often worked on by teams of people rather than one single writer, having a uniform style makes a project easier to maintain so that other people can continue understanding it into the future. Choosing a programming language requires some consideration toward how the goals of your code align with a language's priorities, however small that alignment might be. Likewise, actively opting into a style via tooling and error reporting means considering other readers of your code, and how easily you want to be understood.

chapter 3: Code as play

If Rust's tools present us with a "proper" or "correct" style for code-writing, we must also be able to write code incorrectly, improperly, against the norm—to reject the path toward comprehensibility that Rust gives us. Code that's "wrong," that ignores what's considered "good" by dominant modes of expression, is *still code*, and still worth examining for its rhetorical moves. Turning back to natural language as guide for an aesthetics of code, we notice that "non-standard" Englishes are often those spoken by marginalized communities: Black writers using AAVE, immigrants who speak English as a second language or outside of America, trans voices and their swishy falsettos. These voices draw attention to their orality, to everything but the *content* of their messages. We might call these "queer" styles, which break outside of normative notions of good writing, that love excess and the unnecessary and the frivolous as much as functionality.

In the prologue to *On Freedom and the Will to Adorn,* Cheryl Wall describes the embellishment and flourishes of Black American essayists, stylistic choices that exceed, overflow, and draw attention to themselves. Wall calls the attention toward essayistic style by these writers "an enactment of the will to adorn, an expression of an attitude toward language."[40] In writing about experiences with freedom (or of fighting for it), Black writers care about style and embellishment and beauty as much as content, carrying with them "an understanding that language did more than convey information." Even with the urgency of Black protest, these writers pay attention to style—a seemingly excessive quality of writing that takes up space for its own sake. The ability to choose to indulge in that stylistic adornment just because a writer *can*, regardless of urgency, is one of freedom. All writing is stylish, but here, I highlight writing that

[40] Wall, "Prologue."

indulges in embellishment and adornment just because it *can*. Style offers us a way into a potential queer aesthetics of code, one that pushes against notions of code as functional above all else, leaving room for play rather than aiming toward optimization and perfection.

This may seem contradictory. Code's binaries and structures impose strict empiricisms on those who might seek to *escape* discrete categorization. In the hands of hegemonic systems, computation is often used to harm. For example, to echo Os Keyes, some health insurance companies might track what food you buy to calculate premiums. However, that data can only be collected in the grocery stores they can integrate their surveillance systems in. People who do their shopping elsewhere would escape that data collection, but at the cost of higher premiums. Alongside the fact that people who shop at the corner bodega are likely poor, immigrants, and/or people of color, this higher price puts more strain on those who are already struggling to survive. The cost of unintelligibility is to be shut out of systems that don't acknowledge or care about marginalized lives. Keyes observes that a reformist approach to their critique would be to "make sure there's a sensor in the bodega too!"[41] Reform and inclusion *expand* the boundaries of who's surveilled and pushed into standardization. Conversely, being rejected by the state has very real, violent consequences, no matter the size of the boxes it shoves people into.

Even when used as a tool to help those in need, computation fails: N. Katherine Hayles describes Ellen Ullman's experience creating software for AIDS patients to access information, and the disconnect she felt when meeting the very real people that software was serving:

> Then, as the independent contractor responsible for the system, she met with the staff whose clients would be using the software. Suddenly the clear logic dissolved into an amorphous mass of half-articulated thoughts, messy needs and desires, fears and hopes of

[41] Keyes, "Counting the Countless."

> desperately ill people. Even as she tried to deal with the cloud of language in which these concerns were expressed, her mind raced to translate the concerns into a list of logical requirements to which her programmers could respond. Acting as the bridge arcing between the floating signifiers of natural language and the rigidity of machine code, she felt acutely the strain of trying to reconcile these two very different views of the world.[42]

On meeting her clients, Ullman's forced to realize that the lived experiences of these people cannot *possibly* be represented by her software, which—while it may be helpful—cannot care for the nuances of their day-to-day lives. The analog struggles to be contained by finite, discrete, cleanly defined options, set by a programmer's whims and ultimately bounded by the finite size of a machine's working memory. Queer existences work against computation and data, exceeding what systems can be coded to encompass. The friction between the two reveals what can't be grasped by computers, what's unaccounted for and unparseable. As José Esteban Muñoz writes, queerness is a horizon we're working toward but can not yet grasp; if our selves can be represented in an easily computable way, we lose some of our opacity. Amber Jamilla Musser, following Édouard Glissant, describes opacity as a mode of "always and insistently thinking with the possibility, however momentary, of illegibility rather than a stabilized notion of resistance"[43]. Musser details a resistance based not on mutual understanding, but on uncomfortably holding space for difference, allowing unknowing to fester.

Édouard Glissant, in "Concerning the Poem's Information," suggests that the limitations Ullman comes across are "useful for suggesting what is stable within the unstable. Therefore, though it does not create poetry, it can 'show the way' to a poetics".[44] Through the stability of the

[42] Hayles, "My Mother Was a Computer."

[43] Musser, "Introduction."

[44] Glissant and Wing, "Concerning the Poem's Information."

machine—the cracks where the unwritable leaks through, refusing to be contained—we might get a peek at possibilities beyond what computers can capture. Already we can see queer and trans lives clashing against (but also finding humor) in the ways our identities are (mis)represented by digital interfaces that expect normative users. We might be indignant that we're not offered an accurate gender option on an online form, but amused that the same form lists "Venezuela" as a proposed third gender.[45] To not take this formal mangling of our epistemes seriously is critical. If we refuse to take computational systems as *real*, our interactions with them become a form of play.

This is not simply a digial problem. Even removed from the digital world, we turn to boundaries to set the differences between each other: terms of identity and categorization, labels and descriptors, some benign, others harmful. Language, ever imperfect, cannot hope to capture all ways of being—but we reach for it anyway, attempting to describe our existences in this world, and jump the gap between each other. Computation exacerbates the systems it was built to sustain. However, perhaps the grace we give language can be extended to machines, approaching them with the knowledge that they expose incomplete models of reality. That inaccuracy is something to be aware of and push against, but there's also potential to lean into and play with the ontologies that code brings with it. This capacity for play and worldmaking is a source of potential for a queer computational style.

This is not to discount the harm that empirical systems cause, nor minimize the way computed systems exacerbate that harm. These systems are violent and oppressive, targetting the most marginalized and vulnerable, who are least recognizable to hegemonic understanding. This is not a call to fold marginalization *into* discrete, easily digestible categories, where it might be

[45] Elden, "Genders.WTF"

included into the normative and lose its teeth. Inclusion into structural norms renders queer and marginalized people legibile, losing our opacity and becoming readable; we want to escape capture, not to be bound to rigid, slightly-more-expansive categories. We need systems that care for marginalized people and make space for different possibilities of living, but we also have to be aware that no system can acknowledge the sheer range of possible worlds that can be built. The nature of our very *language*, let alone merely code, means the structure and containment that comes with speech is near-impossible to escape. Despite this, the way marginalized people push at the edges of what can be expressed in formal logics is worth paying attention to. Non-normative users fuck with the formal. Following them, we catch a glimpse of the assumptions we've made and the structures we're cleaving to in our descriptions of the world. Queer codes arise from the gaps between what the machine can capture and what we know to be true.

micha cárdenas describes an artwork by the Peruvian artist Giuseppe Campuzano, *DNI*—it's a national ID card where, depending on the angle you view it at, "the sex marker shifts from 'M' to 'T,' for *travesti*, and the image of his face changes."[46] This ID card isn't a computational system in that it doesn't directly involve a computer, but it was still *designed* with computers in mind. cárdenas notes "the seemingly random numbers and '<' signs common to passports today that make their data more easily readable by electronic scanners" on the surface of the card. The algorithm—the series of steps that allow this ID card to be *made*, then comprehended by technology that seeks to parse the information on it—is one that seeks to identify people's genders to the state. Keyes notes that even if the state *recognizes* gender changes and nonbinary genders, recognition and recording is still a violence: "it enables control

[46] cárdenas, *Poetic Operations*.

and surveillance, because now, even aside from all the rigid gatekeeping, a load of people have a note somewhere in their official records that you're trans."[47] Queer possibility demands not a *more inclusive* state, but the abolition of its hierarchies and control. Campuzano prods at the stability of that attempt at identification. She presents her body as something of presentation maleable, that shifts when you move around it. His gender is mobile, unable to be captured by the ID card. She reveals the gaps in the algorithm, which allow her to escape. Importantly, he uses the algorithm itself to present alternatives to the rigid categories it presents, but also moves *beyond* those logics to present something about his gender. Campuzano plays with the patterns of the ID card to create this work of art—one that's not meant to be used at the border, but reminds us of it anyway. If used as a legitimate form of identification, the ID card would break the algorithm even as Campuzano mimics it. Her presentation is one that the ID card won't and cannot account for. Trying to represent it using this algorithm disrupts its functionality and necessitates different modes of representation, ones fluid, ever-changing, and impractical for the technology of the state. Queer ways of being long to exist outside of the state's control, and therefore are partially defined by the field of infinite possibility of what the state is *not*.

This dissonance is accentuated by the different epistemologies at work when moving between the computational, the state that uses it, and the marginalized people who interact with both in precarious and messy ways. There's slippage between what the computer understands of a model, what the human understanding of the model is, and reality, which is where queer possibility thrives—at points where the machine's model of the world doesn't match up with a human one. This might be because a programmer only imagined their software to be used in a particular way or didn't realize a corner case existed, or because the way a system was encoded

[47] Keyes, "Counting the Countless."

doesn't actually match up with how the programmer *thinks* it's encoded, or a myriad of other reasons. Computational systems by nature must make decisions about the "ontologies" they choose to represent because they can't account for infinite possibility. However, those systems rarely—if ever—accurately account for *all* ways of interfacing with them. The moment we imagine a "normal" user for our programs, or expect a particular way of interacting with code, we (necessarily) set limits on what's possible for our systems. This lack of universality isn't a bad thing in itself. It's untenable for code to do *everything*, and neither do we want all possibility to be capturable. A coding practice attentive to marginalized people would care for the way our lives are unaccounted for, and how we navigate our inclusion in, exclusion from, or queering of those systems altogether.

Tech is not inherently evil, although it's easily and readily co-opted by pre-existing systems of power to hurt. It orients us toward the future. Focusing code on functionality and optimization further supports normative violence, racing forward without bothering to examine the effects code has on the world. If we leave space for play and failure in our code, we disrupt that incessant forward movement and create potential for something *new*. Following Muñoz, we can't grasp a queer future in the present because we cannot recognize a new world yet,[48] but we can dream: can technology and code help build and imagine something new? The mathematical logic that code's built on allows the construction and theorization of entire worlds, so it might be possible for it to eschew normative modes of thinking and knowing, and create entirely new logics of being in the world. This ontological power has been operationalized in violent ways, but it's also incredibly compelling. When you can define a new reality in your code, there's no

[48] Muñoz, "Queerness as Horizon."

one true way to solve a problem. We can reject that technology is meant to accurately mimic our world, and instead create “alternate ways of being, living, and knowing.”[49]

***

Today, though, the norms put forth by technology fail to consider marginalized people, whether intentionally or not, letting them consistently fall through the cracks. This failure can be something as mundane as writing a regular expression (regex) to validate fields of a form, trying to guess if the inputs of those fields are correct or not. Regular expressions attempt to describe the structure of a string of characters: in a database search, if you’ve used the * symbol to represent a “wildcard” character, that comes from regex. One regex could look like “^[\w\.]+@([\w-]+\.+)+\.\w+$”, which attempts to identify if a string is an email address or not. This particular regex would match strings that met all these conditions:

1. started with more than one alphanumeric character, a period, or an underscore, (“^[\w\.]+”)
2. followed by an @ symbol, (“@”)
3. followed by a sequence of alphanumeric characters with at least one period in the middle somewhere. (“([\w-]+\.+)+\.\w+$”)

Notably, this expression isn’t *perfect*. It won’t accept email addresses that use certain valid symbols: email+alternative@example.com is a valid address, but wouldn’t be allowed by this regex. It accepts some email addresses that are definitely invalid: email@example.invalidtld cannot be a real address, since the domain “example.invalidtld” cannot exist. The Internet Assigned Numbers Authority, which coordinates website names across the entire Internet, simply

[49] Keyes, “Counting the Countless.”

doesn't recognize ".invalidtld" as a top-level domain, meaning it cannot be registered as the ending of a url.

This attempt to describe emails is relatively low stakes, but *widely* used in practically every signup form you might encounter online. In summer 2020, I have a go at writing this regex for a form for an event I'm helping run—and get it wrong. Someone contacts me after the form goes live, informing me that their email isn't included in the pattern I've described. In this case, that person was able to contact me through other means, and cared enough to bring the error to my attention. In a different context, that rejection would force someone to spend limited time and energy dealing with the error that they might not have, and could potentially dissuade them from filling out that form entirely, leaving them unable to engage with this piece of software.

Attempting to describing this pattern might seem straightforward, but it's more difficult than it seems to identify valid and invalid email addresses accurately.[50] We might say at first blush that an email is a username followed by the @ symbol followed by a website, but then we also have to define what a username or a website looks like—can we only use ASCII characters? What about non-Latin scripts, like Chinese or Tamil? What about characters with diacritics, like in French, Spanish, Vietnamese? What about languages written from left-to-right? How would we write this regex, then? And if emails (which were made specifically for use with computers in mind) seem tricky to define, this doesn't bode well for our ability to accurately represent more nebulous, informally defined categories. In fact, whole websites exist to list ways in which programs make assumptions about their users—their names, addresses, how their computers

[50] Without a formal specification, at least, but that's an entirely different story. Specifications are documents that rigidly, precisely define *exactly* how a programming language should be written and what its semantics are. A specification for email addresses exists (see Resnick, "RFC 5322 - Internet Message Format"), but they are technical and jargon-y documents, and likely not the first resource that comes to mind in practice.

keep track of time, and more.[51] A queer code style might simply not bother with validating an email address field, knowing that not everyone has an email or wants to share it, or questions why emails are being collected altogether, wondering if it's necessary to keep track of that data at all. A queer code style acknowledges the limits of computation against the infinite possibility of human existence, embraces complexity, and doesn't mind doing the extra work to accomodate that.

These incongruencies between a programmer's imagination and what actually *happens* in a computer's working memory manifest as glitches and bugs, from the relatively benign to legitimately harmful. Ian Gilling discusses glitch as a site of computers' failure. When computers glitch out, machines become more machine.[52] It's revealed here most cleanly that human logic falters and struggles to match an epistemology inherently different from ours. Gilling notes that the glitch demystifies tech, revealing it as flawed rather than sterile and pristine. Glitches draw attention to tech's physicality/materiality rather than the patterns of pixels on your screen. They break the illusion that anything happening on your computer is concrete or natural; they're a *material* side effect of code. They happen when code is *run* on machines rather than when it's theorized about in proofs and papers. The computer unexpectedly speaks here, pointing to its own agency. Glitches remind us that computers aren't perfect machines, and we can't have complete control over what happens inside our computers. In the act of translation between our language and the computer's, meaning gets lost. Our communication with computers is necessarily incomplete. The abstraction of programming languages smooths out fine detail, but to know *exactly* what's going on under the hood is near impossible. It takes a *great* amount of knowledge and skill to understand what's happening at the lowest levels of computation. People

[51] McKenzie, "Falsehoods Programmers Believe about Names."

[52] Gilling, "Haunted by the Glitch."

program in higher-level (more abstract) languages because it's easier, because it makes "commonplace" tasks simpler to grasp. Pushing around bits and registers is both often unnecessary and beyond the expertise of amateurs, and a waste of time to consider. The inner workings of computers is hidden under layers of silicon and electricity, becoming a black box—an incomprehensible interiority we cannot get much of a glimpse into. Glitches break the illusion that the code we write to our machines is completely transparent to *any* party involved, and remind us not to take our writing *too* seriously: we're all making this up in the end.

Moreover, code *itself* is a form of language not easily understood. The rigid structures of programming language in turn produce algorithms incomprehensible, procedures for reasoning alien even when describing them in natural language. I glimpse here some possibility for queer, unreadable opacity—lack of understanding thrives in poetry. Jakobson writes that "the supremacy of poetic function over referential function does not obliterate the reference but makes it ambiguous" (i don't have the full citation for this).[53] Where other styles of writing ask for clarity (such as Strunk and White's insistence on it),[54] poetics might prefer visual, semantic, or audial pleasure over communicating accurately exactly what you mean. For those learning how to code or who simply have been hacking at a problem for too long and don't know what's going on anymore, just typing *stuff* into a program and hoping it runs isn't outside the realm of possibility. Here, I ask the computer running my code if what I'm doing makes sense, and evaluate the results it gives me: compiler errors, bugs and crashed programs, miraculously working executable files. I don't understand what I'm doing—I'm throwing spaghetti at the wall, wondering if what I'll somehow stumble on a working solution based on vibes alone, on what looks right. That vibes-based approach to a coding practice is by no means sustainable, and

[53] Jakobson, "Closing Statement."

[54] Strunk and White, *The Elements of Style*.

rarely results in well-written, easily understandable code—but it's a style based purely on the visual. This practice uses the vocabulary of a programming language to build something that *looks like code*. I stop writing to convey information, but instead am mimicking the stylistic qualities of code I've already seen.

Here's a code snippet from an old problem set I did two years ago:

```
(** **** Exercise: 2 stars, standard (In_app_iff)  *)
Lemma In_app_iff : forall A l l' (a:A),
  In a (l++l') <-> In a l \/ In a l'.
Proof.
  intros A l1 l2 a. split.
  - induction l1 as [| h1 l1' IHl1].
    + destruct l2 as [| h2 l2'].
      * simpl. intros [].
      * simpl. intros [Hl2 | HIn].
        { right. left. apply Hl2. }
        { right. right. apply HIn. }
    + destruct l2 as [| h2 l2'].
      * simpl. intros [Hh1 | HIn].
        { left. left. apply Hh1. }
        { rewrite app_nil_r in HIn. left. right. apply HIn. }
      * simpl. intros [Hh1 | HIn].
        { left. left. apply Hh1. }
        {
          simpl in IHl1. apply IHl1 in HIn.
          destruct HIn as [HInl1 | Hh2 ].
          - left. right. apply HInl1.
          - right. apply Hh2.
        }
  - induction l1 as [| h l1' IHl'].
    + simpl. intros [[] | HInl2].
      * apply HInl2.
    + simpl. intros [[Hha | HInl1] | HInl2].
      * left. apply Hha.
      * right. apply IHl'. left. apply HInl1.
      * right. apply IHl'. right. apply HInl2.
        (* if the answer to this one is significantly shorter and less
complicated i will cry. thanks *)
Qed.
```

You don't need to understand this. *I* certainly didn't when I wrote it[55]—you can see my comment at the end telling you just how frustrated I was by the end of my writing it. However, this piece

[55] Nor do I understand it any better now.

of code uses the various terms I knew were available to me in this particular programming language (“simpl”, “left” and “right”, “apply”, “induction”, the various bullets and indentations), and tosses them together to make some program-looking soup. You can notice patterns even without knowing what this piece of code does. At the beginning of an indented section we often invoke “simpl”; after applying we add a line break or back out of an indented section like a refrain. Brackets open and close, and bullet points make pointless lists. We’re directed to go left or right without really knowing where we’re wandering. If I stop thinking about what these words mean and simply let the syllables spill out of me like so many sounds, I see structure in the instruction. Style and meaning are interlocked, but style *exceeds* meaning. It’s something we can recognize even if we don’t understand a single line of code. This approach to code doesn’t require us to understand the details of its syntax and semantics: instead, we pay *attention*.

While writing this, I looked for patterns I recognized and wrote down keywords in an attempt to wheedle the compiler to tell me something new, working at the whims of a program that told me if I was right or wrong. I didn’t understand what each line does, but that doesn’t matter—the result is a combination of machine desires and what I thought the code *might* look like based on previous encounters with it. It becomes a practice based solely on style rather than meaning, poetics over sense and reference. In poetry, Jakobson calls the subconscious understanding of a word’s connotation (a “dark” or “light” or “heavy” or “cool” synesthetic quality) “sound symbolism.”[56] A similar phenomenon happens here—not based on the beauty of any particular phrase in code, but a vague feeling of what *might* work, whose correctness is up to the machine, ultimately, to decide.

[56] Jakobson, “Closing Statement.”

***

In part, this program is so unfamiliar because of the programming language it uses. Coq/Gallina, a little-known functional programming language created primarily for the purpose of proof-writing, lacks many of the control flow structures we take for granted in most general-purpose programming languages. More esoteric programming languages ("esolangs," colloquially) exist. These languages were not made to be generally used, created for one specific purpose in mind. Often that purpose is simply for *fun*. Esolangs disregard the functionality of computers to create strange, purposefully opaque codes to write with. Unlike general-purpose programming languages, which generally throw around a similar set of concepts (for loops, if statements, function calls, structs…), esolangs tend to embrace weird syntax and symbols, extended and impractical metaphors, and jokes. They're languages that exist for the sake of existing, for the joy of having been imagined.

One of my favorite esolangs is HOtMEfSPRIbNG, or HOMESPRING, which stands for "Hatchery Oblivion through Marshy Energy from Snowmelt Powers Rapids Insulated but Not Great."[57] HOMESPRING takes the programming concept of "streams" and takes the metaphor—streams of data to literal streams—to its extreme, pushing all of its language features into an analogy of salmon swimming up a river. One theoretical program written in HOMESPRING looks like this:[58]

```
universe bear hatchery Hello,. world!.
 powers   marshy marshy snowmelt
```

This program prints out the words "Hello, world!", then exits. An equivalent program in Python would simply read,

[57] Binder, "Homespring."

[58] An example taken from Binder, ibid.

```
print("Hello, world!")
```

The Python program is short and to the point. HOMESPRING ignores this easy transparency and dives into the joke of its stream-based metaphor. In a "tutorial" for the language, Binder writes another version of the "Hello, world!" program, noting that "this program is functionally equivalent, but it looks like a strange poem. That is considered a Good Thing is [sic] HS terms":

```
Universe of bear hatchery says Hello. World!.
 It   powers     the marshy things;
the power of the snowmelt overrides.
```

Complicating and slowing down programs for aesthetics is *encouraged* by the language, wanting the programs written to look like natural language. This complication actively makes it more difficult to read and write this language, existing purely because it *can*. HOMESPRING is primarily a joke, but it leaves ample room for writerly styles. Outside of a handful of reserved words ("bear," "hatchery," "oblivion," and the like), HOMESPRING allows *any* word to be written in the program. In the snippet above, most of the words aren't accounted for by the HOMESPRING language. "of," "says," "Hello World!", "It," "the," "things," "power," "overrides." only serve as embellishment. Their addition complicates the structure of the program, requiring more thought from the programmer to arrange the actual keywords to make the program work properly. Because of these complexities, and its differences from most widely-used programming languages, HOMESPRING was not meant to be understood or written. It's hopelessly difficult to understand what any HOMESPRING code is doing by only reading it, let alone attempt to write it yourself. Serving little to know practical use, HOMESPRING serves as a rhetorical dig at general-purpose programming languages, satirizing commonly-used language surrounding the advancement of technology.

HOMESPRING's utter uselessness is a stylistic choice as much as the literal aesthetics of its code is. In its standard, the document that defines exactly what each term in the programming language means, HOMESPRING pokes fun at the constant need for technology to move forward and improve on itself.[59] Binder calls HOMESPRING's unyielding commitment to the stream metaphor "metaphor-oriented programming," and demands the reader to "learn it now or be left behind! Your current favorite language stands no chance!" Technology so often forces itself onward into the future, growing bigger with no concerns about how much space is being taken up. HOMESPRING mocks this constant cycle of growth by claiming metaphor-oriented programing as "revolutionary" even as it shows off the impracticality of this commitment.

Metaphor, HOMESPRING claims, is a commonly used tool in programming languages to build up abstractions. Abstractions offer shorthands for commonly encountered patterns in programming. A procedure described in many lines of code can be named—that shorter name stands in for the whole procedure, allowing it to be used more easily, over and over, where that code is needed. If I had some lines of code that put boots onto a cat, abstraction would allow me to argue that the code could also be applied to dogs, since dogs are also four-legged. Abstraction structures programs, threading repetition throughout a piece of code. That repetition has the power to decide that *one thing is like another.* To jump much further back, the example of the regular expression for recognizing email addresses is *also* an abstraction. We assume that every person who fills out the form has an email. We assume that an email is structured in a particular way. This bit of code that applies to cats can also be used for dogs because they all have four feet—does that mean that code that applies to me will apply to you, because we both have names and emails? To critique the dangers of relying too heavily on abstraction, HOMESPRING cheekily leans *into* metaphor. Everything attempts to map back onto objects along a stream: bears, sources of electricity, shallows and rapids, snowmelts. But meaning fails because

[59] Binder, "Homespring-2003 Official Language Standard."

abstraction is incomplete, good at generalizing but bad at handling corner cases. Many things in programming are not like a river at all. HOMESPRING's abstraction makes it actively, *purposefully* more difficult to code in it.

However, when we lean back and don't try to understand what's happening in this code—as a programming language or even as English text—we're still left with some *feeling* of what this language was meant to invoke. The code above sketches out a river, one with a bear and a salmon hatchery and a power generator and a snowmelt and marshes. It captures an image, however strange and disjointed, of a waterway at which these human and animal needs meet. Fish for water, humans for electric power, bears for food, snowmelt for movement. Because HOMESPRING allows arbitrary text along with these keywords, we're able to construct programs with as much of it as we want, letting code take up more space, and flower to tell a more coherent story. HOMESPRING may not be comprehensible, but it is beautiful and attention-catching—its bizarre syntax stands out among so many languages with the same handful of keywords.

This particular aesthetic comes at a practical cost even beyond reader and writer overhead. HOMESPRING runs slowly, taking much longer than most other languages just to do single-digit addition. Code that goes against the norms of programming often gives up optimization for its poetics. It takes up space, works slowly, makes your machine's fans spin up and him. Kate Compton's zine, "Opulent Artificial Intelligence," pokes fun at the invisibility we demand from computational systems:

> "But do so UNOBTRUSIVELY. Do so UNSEEN. Never, never, make yourself a spectacle. They do not wish to see you or talk to you. You are just here to do work."
>
> "….Oh, and can you do it faster and with less resources?"

HOW DREADFUL

lets make her FABULOUS![60]

Compton offers a computation that refuses to hide in the background, that's aware of the space it's taking up and loves it. This computation is materially tangible, drawing attention to its own medium. It works against needs for speed and easy digestibility, valuing aesthetics and poetics just as much as, or more than, functionality. The material effects of this code are *effective* just as glitches are, take up more space than necessary, drawing attention to itself as software. Code that's excessively complex often churns out "bad" software when compiled because the machine has trouble understanding, stuttering as it tries to parse what's been written. By making software, and the code behind it, visible, we draw attention to what would seek to become naturalized and static. This shift in attention forces us to look at the aesthetics of computation and computational writing, drawing us once again to style. Much of this code is impractical and useless to capitalist logics that seek conformity and speed—logics deeply intertwined with white, cishetero, colonialist normativity.

***

If Lopes defines programming style as the particular restraints set on a program by a problem, programming language, and/or hardware, then a queer code practice grabs at the edges of those restraints, questioning the boundaries of what's possible with code. It wriggles into gaps between what we intended and how we imagined something to work, and what can actually be done. Both code and poetry by nature further restrict the language we can use—so we arrive back at the epitaph of the first chapter, finding that "decoding a poem is not always easy, its meaning often remaining elusive. Poem-ing code to the contrary goes automatically, like magic."[61]

[60] Compton, "Opulent Artificial Intelligence."

[61] Weille, "This Code = this code."

We poem our code, embracing queer poetics that care less for functionality and more for flair and nonsense. Jakobson defines code (in a linguistics context) as the *shared* language between the speaker and the addressee, one that needs “metalingual” commentary to check if both conversants are on the same page.[62] In code, that metalanguage is disturbingly stable because of the formal and fixed semantics of a programming language. But those semantics are also alarmingly *unstable*, counterintuitively, because of their precision. Human language is flexible, carrying varying connotations, context, and meanings; our understandings bubble over, tugging at those neat definitions. We hold onto multiple meanings at once, try to keep them from mixing, ultimately fail. We were not meant to be that kind of precise—so code becomes both painfully accurate and absolutely incomprehensible. Holding onto both sides of that paradox, queer code grapples for multivalence and excess, for stuff you thought you could wrap your mind around but ultimately don’t. And that’s fine. We sit in that unknowing (with machines, with each other, with ourselves) together.

[62] Jakobson, “Closing Statement.”

bibliography

Arawjo, Ian. “To Write Code: The Cultural Fabrication of Programming Notation and Practice.” In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, 1–15. CHI ’20. New York, NY, USA: Association for Computing Machinery, 2020. https://doi.org/10.1145/3313831.3376731.

Becker, Brett A., and Thomas Fitzpatrick. “What Do CS1 Syllabi Reveal about Our Expectations of Introductory Programming Students?” In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 1011–17. SIGCSE ’19. New York, NY, USA: Association for Computing Machinery, 2019. https://doi.org/10.1145/3287324.3287485.

Binder, Jeff. “Homespring.” Webpage, 2003. http://jeffreymbinder.net/misc/hs/hs.html.

———. “Homespring-2003 Official Language Standard,” 2003. http://jeffreymbinder.net/misc/hs/hs.pdf.

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

Camp, Tracy, W. Richards Adrion, Betsy Bizot, Susan Davidson, Mary Hall, Susanne Hambrusch, Ellen Walker, and Stuart Zweben. “Generation CS: The Growth of Computer Science.” *ACM Inroads* 8, no. 2 (May 2017): 44–50. https://doi.org/10.1145/3084362.

cárdenas, micha. *Poetic Operations: Trans of Color Art in Digital Media*. Duke University Press, 2022.

Chun, Wendy Hui Kyong. “On ‘Sourcery,’ or Code as Fetish.” *Configurations* 16, no. 3 (2008): pp 299–324. https://doi.org/10.1353/con.0.0064.

“Clippy Lints.” The Rust Foundation, n.d. https://rust-lang.github.io/rust-clippy/master/index.html.

Compton, Kate. “CASUAL CREATORS: DEFINING a GENRE OF AUTOTELIC CREATIVITY SUPPORT SYSTEMS.” PhD thesis, University of California, Santa Cruz, 2019.

———. “Opulent Artificial Intelligence: A Manifesto.” Digital zine, 2017. http://galaxykate.com/pdfs/galaxykate-zine-opulentai.pdf.

Crichton, Will. “The Usability of Ownership.” arXiv, 2020. https://doi.org/10.48550/ARXIV.2011.06171.

Elden, Effy. “Genders.WTF.” Webpage, n.d. https://web.archive.org/web/20230225160200/https://genders.wtf/.

Gass, William. “The Aesthetic Structure of the Sentence.” In *Life Sentences*. Random House, 2012.

Gilling, Joseph. “Haunted by the Glitch: Technological Malfunction - Critiquing the Media of Innovation.” In *10th International Conference on Digital and Interactive Arts*. ARTECH 2021. New York, NY, USA: Association for Computing Machinery, 2021. https://doi.org/10.1145/3483529.3483667.

Glissant, Édouard, and Betsy Wing. “Concerning the Poem’s Information.” In *Poetics of Relation*. University of Michigan Press, 1997.

Hayles, N. Katherine. “My Mother Was a Computer.” University of Chicago Press, 2005.

Jakobson, Roman. “Closing Statement: Linguistics and Poetics.” In *Style in Language*, edited by Thomas A Sebeok. The Technology Press of Massachusetts Institute of Technology; John Wiley & Sons, Inc., 1960.

Keyes, Os. “Counting the Countless.” Real Life, April 2019. https://reallifemag.com/counting-the-countless/.

Lopes, Cristina Vidiera. *Exercises in Programming Style*. 2nd ed. Routledge, 2020.

Matsakis, Niko, Mark Rousskov, Aidan Hobson Sayers, Ashley Williams, and Nick Cameron. “RustConf 2020 - Opening Keynote,” 2020. https://www.youtube.com/watch?v=IwPRu5FhfIQ.

McKenzie, Patrick. “Falsehoods Programmers Believe about Names.” Blog post, 2010. https://www.kalzumeus.com/2010/06/17/falsehoods-programmers-believe-about-names/.

Muñoz, José Esteban. “Queerness as Horizon: Utopian Hermeneutics in the Face of Gay Pragmatism.” In *Cruising Utopias: The Then and There of Queer Futurity*. NYU Press, 2009.

Musser, Amber Jamilla. “Introduction: Brown Jouissance and Inhabitations of the Pornotrope.” In *Sensual Excess: Queer Femininity and Brown Jouissance*. NYU Press, 2018.

Parrish, Allison. “Language Models Can Only Write Poetry,” 2021. https://posts.decontextualize.com/language-models-poetry/.

Pereira, Rui, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. “Energy Efficiency Across Programming Languages: How Do Energy, Time, and Memory Relate?” In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, 256–67. SLE 2017. New York, NY, USA: Association for Computing Machinery, 2017. https://doi.org/10.1145/3136014.3136031.

Resnick, Pete. “RFC 5322 - Internet Message Format.” Website; Internet Engineering Task Force, Network Working Group, n.d. https://datatracker.ietf.org/doc/html/rfc5322#section-3.4.1.

“Rust Survey 2021 Results.” Webpage; Rust Survey Team, 2022. https://blog.rust-lang.org/2022/02/15/Rust-Survey-2021.html.

Schapiro, Meyer. “Style.” *Anthropology Today*, 1953.

"Stack Overflow Developer Survey 2022." Webpage, 2022. https://survey.stackoverflow.co/2022.

Strunk, William, Jr, and E. B. White. *The Elements of Style*. Fourth. Allyn & Bacon, 2000.

Sullivan, Patrick. "An Essential Question: What Is "College-Level" Writing?" Edited by Patrick Sullivan and Howard Tinberg, 2006.

Vee, Annette. "Introduction: Computer Programming as Literacy." In *Coding Literacy*. MIT Press, 2017.

Vermeulen, Allan, Scott W. Ambler, Greg Bumgardner, Eldon Metz, Trevor Misfeldt, Jim Shur, and Patrick Thompson. *The Elements of Java Style*. Cambridge University Press, 2000.

Wall, Cheryl A. "Prologue: Moving from the Margins." In *On Freedom and the Will to Adorn: The Art of the African American Essay*. University of North Carolina Press, 2018. http://www.jstor.org/stable/10.5149/9781469646923_wall.5.

Watson, Cecelia. "Introduction: Love, Hate, and Semicolons." In *Semicolon: The Past, Present, and Future of a Misunderstood Mark*. ECCO, 2019.

Weille, Jan de. "This Code = this code." Artist's statement. In *Taper*. 7. Bad Quarto, 2021.