Diagram, Debug, Fold: Programming as a Form of Critique

Olivia Kan-Sperling
Honors Thesis in Modern Culture & Media
Brown University, April 2020

Diagram, Debug, Fold:
Computer Programming as a Form of Critique

Olivia Kan-Sperling
Honors Thesis in Modern Culture and Media (A.B., Track II)
Brown University, April 2020

First reader: Péter Szendy
Second reader: Joan Copjec

TABLE OF CONTENTS

INTRODUCTION

> *The author has never, in any sense, photographed Japan. Rather, he has done the opposite: Japan has starred him with any number of 'flashes'; or, better still, Japan has afforded him a situation of writing. This situation is the very one in which a certain disturbance of the person occurs…*
>
> - Roland Barthes, *The Empire of Signs.*[1]

If, in the new-ish fields of new media or software studies, much has been done to apply various semiotic frameworks to "digital objects," less attention has been paid to the process of *making* such texts (code, media, architectures).[2] Few have asked, *what it is like to program?* One aim of this thesis is to offer such a description: of the practice or "experience" of computer programming. Although I hope this description might resonate with programmers, and my analysis is founded both on my own coding experience, and accounts of computer scientists, my primary goal is not to provide a "photograph" or an ethnography. Just as Barthes does in his beautiful study of a "Japan" that, he emphasizes, he has partially invented, I write on computer science by isolating "a certain number of features (a term employed in linguistics), and out of these features deliberately [forming] a system."[3] Like Barthes's description of the "Japanese" signifying system composed of tempura or chopsticks, this project takes a semiotic perspective—though its object is less the linguistic system of code itself, and more what it is like to interact with it. So we will ask, more specifically, after something like the "semiotic experience" of programming: *In what way can we call programming reading or writing?*

Programs share many obvious and superficial characteristics with printed texts (for example: being composed of discrete, alphanumeric characters; like books in the West, read from left to right and top to bottom; like in the English language, containing symbols such as "print," "and," "list"). But the more specific concepts of reading and writing to which I compare

---

[1] Roland Barthes, *Empire of Signs* (New York: Hill and Wang, 1982), 4.
[2] Particularly noteworthy among these theorists, and those which I will both draw upon and argue against here, are: Lev Manovich, Wendy Hui Kyong Chun, N. Katherine Hayles, Alexander Galloway. Their work is often on the *object*, and sometimes on the *user*, but rarely on the *writer* of new media.
[3] Barthes, *Empire,* 3.

programming are patterned on the work of several theorists whose own reading and writing practices are most "isomorphic" with the features of programming in which I am interested. In the language of computer science, a "high-level overview" of these features: Programming will be reading/writing as 1. Gilles Deleuze's acrobatic invention of shapes; 2. Jacques Derrida's delicate and tricky figures of speech or plays on the materiality of signs; 3. Michel Foucault's discursive, addictive technologies of self. Throughout all three chapters, this programming is reading-writing as the creation and manipulation of *forms*. Form, for us, has two primary senses: 1. dynamic shapes or moving figures, topology, morphology, diagram; 2. language signs and syntax, the support (two aspects which will be discussed in the first and second chapters, respectively). In summary, programming, here, emerges as a figurative, diagrammatic activity (Deleuze, Althusser), infected by the materiality of its semiotic system, and irremediably given to error (Derrida). Its processual workflow hooks the programmer into a conversation with their machine, creating an addictive discursive feedback loop. Creating this cybernetic system is a mode of subjectivation in the sense of a technology of the self (Foucault), or fold (Deleuze).

It is less that I am applying or mapping, say, Derrida or Foucault's theories *of* writing in general onto the program, and more that I compare programming to *their own* writing and reading. We are speaking, therefore, more specifically, on the reading of the critic or the writing of the philosopher. I am interested in tracing the isomorphisms between these two activities—programming and philosophy or critique—despite and because they are often assigned such opposite polarities. This is perhaps most easily identifiable in Heidegger's technophobia, but also for Foucault, Derrida, and Deleuze, the program or code is often a figure for that which, because mechanical, is antithetical to the openness, otherness, or outside-ness of Thinking. So I write from slightly outside both computer science and critique, introducing these foreign discourses to each other in the vague interest of opening out towards "the possibility of a difference, of a mutation, of a revolution in the propriety of symbolic systems" of one or the other, as Barthes does by inventing a semiotics of the Orient for the Occidental reader.[4] Computer code affords me a "situation of writing" through which to think thinking and the technical together, to once again rearrange these terms in relation to one another, in a way that causes "a certain disturbance of the

[4] Barthes, *Empire,* 3-4.

person" to occur.[5] It is such a disturbance via the "flashes" of technology that constitutes the cybernetic mode of subjectivation of the programmer, which, in turn is what makes programming and philosophy disturbingly similar. In a sense, this comparison is just another way to write in the spirit of originary technicity or prostheticity, a tradition that animates every chapter but to which we will return to explicitly only in the conclusion.

A specification of scale: What is programming?

Digital media comprise a heterogeneous multiplicity of technical and semiotic systems. This is a function of the extraordinary technical complexity of the digital computer as a "writing machine" and the ensuing diversity of its practical applications. The attempt to provide a total account of an actually existing practice—programming—on this "universal" machine is impossible. I will attempt to specify what sort of "programming" will be discussed here, first by a partial list of exclusions.

I will *not* emphasize machine learning, the rapidly developing branch of computer science concerned with enabling computers to learn. The increased economic and disciplinary relevance of this sub-field of artificial intelligence has prompted widespread discussions and anxieties over its ethical and philosophical implications, the nature of intelligence, and what it means to be alive at all. The field of software studies has, similarly, seen a turn away from the critical concerns of traditional Anglophone media theory (representation, ideology) and towards more metaphysical considerations (the ontological status and "experience" of the computing machine itself). Although I will draw on some of these authors, including Beatrice Fazi, Wolfgang Ernst, and Luciana Parisi, my analysis engages primarily with (older) texts by authors interested in the signifying strategies of new media: Lev Manovich, N. Katherine Hayles, and Wendy Hui Kyong Chun. Given my interest in programming as a creative practice, the impenetrable deep neural nets of machine learning, and related problems of cellular automata and emergent synthetic life-forms, are less relevant in that their specificity lies in that which is *not* programmed by the human. Nor will I address programming at the scale of distributed systems—the task of coordinating the communication of a network of machines crucial to the

[5] Barthes, *Empire*, 4.

globalized economy. Alexander Galloway's *Protocol,* on network transfer protocols, is one example of a work which engages with the technologies that make the Web "world wide." Nor am I studying the labor practice of software engineering. Former engineer Federica Frabetti, for instance, has written on the complex discursive networks that structure the contemporary software industry in her book *Software Theory*.

This is programming, therefore, at a slight remove from both practical applications and the theoretical discipline of computer science. My examples are fairly academic exercises: the implementation of classic data structures and algorithms. The programs I examine can be run on a single, non-networked machine. This narrowing of scope or limiting of scale, which still allows for an infinitude of kinds of programs with different ends, is necessary because of the complexity of the technical processes involved. It is also a function of my mostly academic personal background in programming. What I hope to address is the lowest common denominator of all applications of modern computer science: the writing of alphanumeric symbols which will be interpreted by a machine.

In 2020, this "writing" is typically done in high-level programming languages: the recognizably Anglophone languages in which the vast majority of code written by humans is produced. As case studies, I will take examples of programs written in the common high-level languages Java and Python.[6] These are "high-level" in that they require many translations into other layers of language before being interpretable by the machine itself. These "compilations" and "interpretations" are automated processes; running any high-level program, therefore, initiates multiple invisible readings and writings of yet other programs, in other languages, by programs themselves. Although, today, this code is rarely written by humans, I will also discuss such lower levels of technicity because the account I am working towards is that effective programming requires understanding and (indirect) figuration of multiple scales of technicity simultaneously. Programming writes far more than just the Java or Python file itself.

[6] Those familiar with different programming paradigms will notice that I emphasize object-oriented design and languages. This is primarily because OOP's concepts seem more easily graspable by a lay audience. I believe the same analysis could be constructed if one were to privilege, say, functional programming.

A note on terms and font: This has been written with the intention that a reader with no background in computer science could understand the crucial points of the argument, if not *all* of their details. For the sake of clarity, in order to call attention to their status as technical nomenclature, general computer science terms are printed in **gray bold face** the first time they appear in a section of text. Some terms may not be completely explained when first introduced, but will be elaborated further on when doing so becomes relevant. References to *specific* programming language constructs, or blocks of code, are printed in `monospace`.

CHAPTER 1. SCALE AND DIAGRAM

> *Our method … aims to produce a system of thoughts that bridges different orders of magnitude through developing a theory of relations. Philosophical concepts can be seen as inventions that try to overcome the incompatibilities or even indifferences between two orders. Hence philosophy remains technical in this project.*
> - Yuk Hui, *On the Existence of Digital Objects*.[7]

The computer writes, and reads, in many different virtual and material locations, and in many different "languages," almost instantaneously. The coexistence of these semiotic systems, this heterogeneity of digital and analog signifiers that includes graphical images, alphanumeric symbols, and voltage differences, perhaps best distinguishes the new media object from older technologies. Arguments like Friedrich Kittler's oft-cited claim that "there is no software" because, ultimately, all code functions only as voltage differences in the hardware circuits of the machine, are, for us, beside the point.[8] We will see, rather, that the specificity of programming lies in the activity of creating relations or mediations between these different systems, which, like Yuk Hui, I conceptualize as operating at different *levels* or *scales*. This should not be understood exclusively or explicitly in the spatial sense of object *x* being "smaller," "lower," or "inside" another object *y* (although all three of these metaphors will be used), but to indicate the coexistence of multiple "orders" of what Hui (following Gilbert Simondon and Gaston Bachelard) calls "technical reality." Scalar levels are therefore understood loosely as different "dimensions." Hence, "depth" and "inside" ought to be seen with quotation marks. When I have recourse to language such as "high-level" (to describe "visible" semiotic systems with which human programmers interface directly) and "low-level" (technical, semiotic operations carried out by the machine without direct human intervention), it is because these are the metaphors through which computer science spatializes and hierarchizes the operations of its technical object.

---

[7] Yuk Hui, *On the Existence of Digital Objects*, (Minneapolis: University of Minnesota Press, 2016), 30-31.
[8] His argument is akin to claiming that "there are no apples" because, *in the final analysis,* apples can be reduced to atoms. Friedrich Kittler, "There is No Software," *CTheory* (1995), accessed March 31, 2020, www.ctheory.net/articles.aspx?id=74.

The aim of this chapter is not to provide a definitive description of the set of forms a program takes or operates on. Rather, by engaging the work of several new media theorists as well as the figurative language and visual metaphors computer scientists use to do their work, I wish to illustrate the heterogeneity of forms that structure computer science and the discourses around it in order to highlight the importance of formal thinking to programming. I take Lev Manovich's figures of the "fractal" and the "binary" as a starting point, expanding and complicating his argument throughout the chapter. Here, I understand mediations between abstract/concrete and time/space—which Hui might characterize as *relations* that bridge different orders of technicity—as problems of *figuration*. Although Hui's work does not bear on reading and writing, I cite his description of his own method above in order to gesture towards another possible path that one might take in order to compare programming and philosophy: in terms of relationality, and Simondon and Bachelard's work on orders of magnitude. Instead, at the end of the chapter, we will begin our comparison of programming to reading, writing, and critique via a discussion of the figural operations Althusser performs in the opening essay of *Reading Capital*, and Deleuze's description of Foucault's discursive diagrams.

Programming as figuration

LAYERS, PARTS AND WHOLES

In accordance with the high-level/low-level vocabulary of computer science, most attempts at a definition of new media objects by media theorists gesture towards some idea of depth, and the fact of mediation, or translation, between layers. As N. Katherine Hayles wrote in 2004, "print is flat, code is deep."[9] This tendency is exemplified by Benjamin Bratton's 2015 book *The Stack*, in which the eponymous computational data structure is the privileged model not only of network technologies, but our geophysical and political reality, which Bratton organizes vertically in the stratified yet "interdependent layers" of Earth, Cloud, City, Address, Interface, User.[10]

---

[9] Whether this mischaracterizes the complex practices of citation and context that might give print a certain "depth" is another matter. N. Katherine Hayles, "Print is Flat, Code is Deep: On the Importance of Media-Specific Analysis," *Poetics Today* (vol. 25:1): 67-90.
[10] Benjamin Bratton, *The Stack: On Software and Sovereignty,* (Cambridge: MIT Press, 2015), 11.

This particular formal quality is also emphasized in one of the earliest, canonical texts of software studies: Manovich's *The Language of New Media.* In what he calls a "textbook" for the study of "new media objects" (which I will abbreviate here as "NMOs"), Manovich hopes to identify their "emergent conventions, recurrent design patterns, and key forms."[11] His media are new in that they are described in terms of their similarities to and differences from older media, primarily cinema—differences which derive from their technical functioning. While Manovich privileges objects more traditionally recognizable as "media," like digital images or video, his definition allows room for software or even code; the five general principles he derives ought to "hold true across all media types, all forms of organization, and all scales."[12] These principles build on each other, as in axiomatic logic. Manovich's structural, "bottom-up" analysis therefore mirrors the hierarchical composition of NMOs themselves: Just as a program "undergoes a series of translations" from high-level programming language to executable code to binary code, *The Language of New Media* progresses from binary code to computer program in order to arrive at a theory of the organizing logic of NMOs.[13]

Manovich gives this hierarchy or verticality two distinct, even contradictory, shapes. The first is that of the *fractal*: "Just as a fractal has the same structure on different scales, a new media object has the same modular structure throughout."[14] NMOs are discrete objects which can be combined into "larger-scale" objects, without the individual elements losing their separate identities. In a video-editing software, for example, one might collate a series of shorter video clips to form a longer video of the same file format. If NMOs comprise a multiplicity of layers, this is in the strict sense of "comprise"—a strangely bidirectional verb because it has two meanings which are, if not opposites, then inverses: both "to constitute, to make up" and "to consist *of*, to be made up *of*." This quality perfectly captures the recurrent, synecdochic relations of parts to wholes Manovich expresses with the fractal.

---

[11] It is in this sense that he emphasizes a structural "language," rather than describing their "aesthetics" or "poetics." Lev Manovich, *The Language of New Media* (Cambridge: MIT Press, 2001), 11-12.
[12] "Software" is typically understood as a consumer-facing graphical user interface, whereas "code" is a text written by programmers (although, today, the line between consumer and producer is increasingly blurred, and, as we will see, all code is also an interface). Ibid., 14.
[13] Manovich, *Language,* 11. For Manovich, "logic" is quasi-synonymous with "language."
[14] Ibid., 30.

Fractalization is produced by the programming technique of **modularization**, in which a problem is divided into a set of discrete sub-problems. This technique is to Silicon Valley what the assembly line was to Ford; its innovation enabled the emergence of the software industry by allowing multiple engineers to work on the same project without requiring interaction with the implementation details of their colleagues' work. As Wendy Chun emphasizes, the method is a way of hiding (or, as she argues, mystifying) the interior workings of different parts of the program.[15] Modularization is perhaps best illustrated by **object-oriented programming** (OOP), the widely popular programming style in which a program's functionality is distributed across discrete **objects** which are put into communication with one another. One object can contain many other objects; each individual element is thus isomorphic with the whole. While many programs—particularly at the higher levels Manovich discusses—may exhibit such a fractal-like isomorphism, a closer examination of OOP, as well as lower levels of technicity, will allow us to show that heteromorphism between parts and wholes is an equally salient feature of programming.

ASYMMETRIC BINARIES, HETEROMORPHISM

The other figure that Manovich emphasizes might be called the "asymmetric binary." Rather than a multiplicity of synecdochic or fractalized layers, this is a *dual opposition* between contrasting terms in which one precedes the other. For Manovich, these two terms might be, for example: culture/computer, surface/depth, immersion/information, form/content, narrative/database, or action/representation.[16] The two terms are related by "transcoding": a type of transformation that refers both to the technical process of converting numerical representations into media (image, video, etc) within the machine, and the higher-level process in which technology "reformats," or influences, human culture. In the logic of transcoding, one form is transposed into its "opposite" on a higher scale. This dynamic of transcoding between asymmetric binaries recurs under many names and on many different levels of Manovich's analysis.

---

[15] Wendy Hui Kyong Chun, "On Software, Or the Persistence of Visual Knowledge," *Grey Room* (vol. 18: Winter 2004): 38.

[16] One might note that these oppositions follow a Marxist figure of base –> structure, in which the "deep" technical term determines the "superficial" cultural layer. Manovich, *Language*., 229, 65, 216.

A closer examination of the dualism **data/algorithm**, which Manovich mentions only briefly, will allow us to highlight what Manovich's theory of transcoding addresses, if not entirely develops: that part of the specificity of computers is their folding together—though not dissolving—of such binary categories. Manovich defines data and algorithm as follows:

> Computer programming encapsulates the world according to its own logic. The world is reduced to two kinds of software objects that are complementary to each other—data structures and algorithms. Any process or task is reduced to an algorithm, a final sequence of simple operations that a computer can execute to accomplish a given task. And any object in the world—be it a population of a city, or the weather over the course of a century, or a chair, or a human brain—is modeled as a data structure, that is, data organized in a particular way for efficient search and retrieval. Examples of data structures are arrays, linked lists, and graphs. Algorithms and data structures have a symbiotic relationship. The more complex the data structure of a computer program, the simpler the algorithm needs to be, and vice versa.[17]

The categories of data and algorithm are particularly interesting because they play out other divisions with recognizable philosophical resonances: such as between object and process, form and content. Following the pattern of transcoding, Manovich often describes the data structure as the underlying figure that "supports" the algorithm. But, as we will see, when considering the totality of technical scales within the digital object, there is not merely one, but many thresholds of such reversals; rather than trying to establish one of the two as causally or temporally prior, we will instead emphasize the process of transformation of forms as that which defines programming. The figuration of one form as its opposite on another scale is also a metamorphosis, a trans*form*ation that introduces discontinuities and heteromorphisms into Manovich's figure of the fractal which is identical with itself on every scale.

In the definition above, "algorithm" appears rather vaguely as "process"—any dynamic in-between that transforms data inputs into data outputs. Manovich also seems to use "data" (typically considered the "raw material" on which the program operates) and "data structure" (typically understood as the manner in which the former is represented) synonymously. This distinction is thus one of *process-in-time* vs. *shape-in-space*. On certain

[17] Manovich, *Language*, 223.

levels, programs do indeed differentiate between processes and objects. But the "symbiotic relationship" Manovich describes, in which the complexity of a data structure and an algorithm are inversely proportional, exists precisely because the distinction between the two is purely one of convenience—or, rather, a function of scale.

For example, a programmer might want to use a **priority queue**—essentially a list sorted in accordance with some comparative function—to structure his data. The high-level language Java offers a **class**, called `PriorityQueue`, in its **library**.[18] The `PriorityQueue` class allows the programmer to add items to a list which are sorted automatically, without him needing to invoke the command `list.sort()`, or even implementing the sorting function from scratch. Although Java would describe an instance of a `PriorityQueue` as an *object,* the organization of this data structure is facilitated by an algorithm that is built into it (a particular sorting function, for example). What are, in a normal list, two separate concepts (the object of list, and the process of sorting it), are, in `PriorityQueue`, combined as a single object. In an object-oriented language like Java, this is the basic pattern by which all programming proceeds: when the programmer writes a program, she does so by creating classes of objects to which correspond both noun-like **attributes** (for example, `name`), and verb-like **methods** (like `sort`). Within the scope of a function definition, moreover, she might define other objects as well as other functions. In this sense, it is not only that object becomes process, but that process also becomes object, which might become process again, and so on.

The distinction between object and process is, therefore, merely a conceptual tool that creates different levels of abstraction. Java's `PriorityQueue` data structure is a shorthand for what, in another language, would need to be programmed by hand as a complex amalgamation of functions and other data structures. In this sense a data structure is itself a program at another scale. But this relationship is not that of the isomorphic fractal; the parts (in this case, methods and attributes) that make up the whole of a Java object are heteromorphic with each other and thus with the object they compose.

---

[18] A class can be imagined as a generic template for making specific data objects. A library is a kind of interface of ready-made data structures and functions that a programmer can use—in other words, a collection of classes. These might be included as standard linguistic features of a language, or require being "imported" or added from an external source.

The mediations through which the semiotic figures—lists, integers, functions—involved in programming pass are therefore not only "translations," but, literally, trans*form*ations across scales. This is also true in the case of analog and digital signs. As N. Katherine Hayles emphasizes in "Print is Flat, Code is Deep: The Importance of Media-Specific Analysis," what we call digital computers in fact function via both analog and digital representations. "At the most basic level of the computer are electronic polarities, which are related to the bit stream," the sequence of discrete binary digits, "through the analogue correspondence of morphological resemblance."[19] Analog communication occurs via a similarity of continuous shapes, whereas digital communication relies on an arbitrary system of discrete signs. Human language, from the perspective of classic linguistics, falls into the latter category. In a computer, the zeros and ones of binary data are transcoded into increasingly complex discrete signs, including hexadecimal code, assembly commands and high-level keywords. The highest level of the **graphic user interface**, however, functions once more via analog resemblance in the form of interface icons that imitate the morphology of items like folders or film cameras. Hayles's computer is a layer-cake, an "Oreo cookie-like structure with an analogue bottom, a frothy digital middle, and an analogue top."[20] Not only are the *forms* of the computer variously discrete (digital) or continuous (analog) at different scales, but the manner in which mediation between these layers occurs, the process by which information is translated, itself varies between linguistic transcoding and morphological resemblance.

I draw on Hayles's observations to demonstrate the heterogeneity of a computer's semiotic systems, as well as to give another example of a layer-model of new media that takes a different form from Manovich's either bi-partite or self-identical one. While high-level programming does not typically necessitate awareness of either analog end of the physical computer, the transfiguration of an abstract, figurative data structure or algorithm into a discrete sequence of Java commands during implementation is also a shift from information conveyed via analog resemblance to digital encoding. In other words, one transforms the morphological, analog figure of a data structure, like a list, into language, which is a digital representational

[19] Hayles, "Print," 75.
[20] Ibid.

system. As we will see, this shift from image to implementation is also one from abstract to concrete.

ABSTRACTION (GENUS/SPECIES)

We have described the manner in which a program's functionality is split and distributed across the axis of space versus time. Another dimension we might consider is *abstraction,* and the oppositions ideal versus material, or form versus content. There is a distinction between data structures and algorithms as abstract concepts—"linked list," "merge-sort"—and their **implementation** in a particular language, as well as between their implementation and their **instantiation** with particular data. For example, the priority queue is a generic or abstract data structure whose defining feature is that it is sorted (or prioritized), but Java and Python's priority queue classes differ both syntactically, functionally, and in their implementation "under the hood."[21] For example, on the most superficial level: Python's more primitive version of the Java `PriorityQueue` class is called `heapq`. Not only do the signs or operations through which one manipulates a priority queue differ between Java and Python (they are, after all, different "languages"), but the algorithm through which the class is realized (or implemented) in the lower-level code that "underlies" Java or Python might be different as well. Instantiation, conversely, designates the process through which a given language's `PriorityQueue` class is used to contain actual data. For example, within a Java program, one concrete **instance** of the general class `PriorityQueue` is created as, say, `myAlphabetizedList = ["Amelia", "Bella"]`. The higher levels of programming, especially in object-oriented languages, involve a potentially infinite layering of such abstractions.

In object-oriented programming (OOP), the classes of objects you create in order to encapsulate functionality are essentially templates or patterns that are used to create multiple instances of one object. One class can therefore be re-used within the same program, as well as across different projects. A **concrete class**, moreover, can **inherit** certain features from a potentially endless chain of other concrete classes, as well **abstract classes**—classes which can

[21] They may, for instance, use different sorting algorithms. We will return to differences between programming languages in Chapter 2.

not be instantiated. These relationships of abstract/concrete, mediated by the concept of inheritance, essentially follow the model of genus/species taxonomy inaugurated by Linnaeus. For this reason, OOP is often taught via zoologically-themed exercises: `Mammal` is an abstract class for the concrete class `Deer`, which can be instantiated as `bambi`. Classes can also implement **interfaces**, which are different from abstract classes in that they merely specify certain methods which must be present in a class, without providing an implementation for them.

Figure 1, on the following page, shows a diagram of Java's built-in `PriorityQueue` class. This is a type of schema through which a programmer might conceptualize their own `Mammal` or `Deer` classes. The table in Figure 2 shows the methods (the functionality) that `PriorityQueue` inherits from its superclasses, among them `java.lang.Object`, the highest-level class from which all Java classes inherit. The `<E>` with which `PriorityQueue` is annotated indicates that a particular instantiation of the class must be given a certain **type**, such as `PriorityQueue<String>` or `PriorityQueue<Deer>`, or even `PriorityQueue<PriorityQueue<String>>`, specifying the type of object that the queue will contain. The place-holder parameter `E` (element) is thus another notation of abstraction. As in our previous discussion of objects and processes, there are multiple stratifications on which what is "abstract" on one level becomes "concrete" on another, as well as lateral relations between forms of abstraction.

We can also identify the importance of such layers of abstraction beyond the scope of OOP — in the implementation, for example, of algorithms or data structures themselves. Figure 3 shows high-level **pseudocode** for the minimax algorithm, an algorithm used in game theory to determine the optimal move for a player in a two-player zero-sum game such as tic-tac-toe. An algorithm is usually considered independently of the particular data structure one might use to implement it. In this case, the diagram in Figure 4 visualizes an example minimax game implemented as a tree-search problem.[22]

[22] Minimax is a fairly simple algorithm which dictates that you assume that your opponent, also playing via the rules of minimax, will take the move which is *least* advantageous to you. "Games," Massey University of New Zealand, accessed March 31, 2020, https://www.massey.ac.nz/~mjjohnso/notes/59302/l05.html.

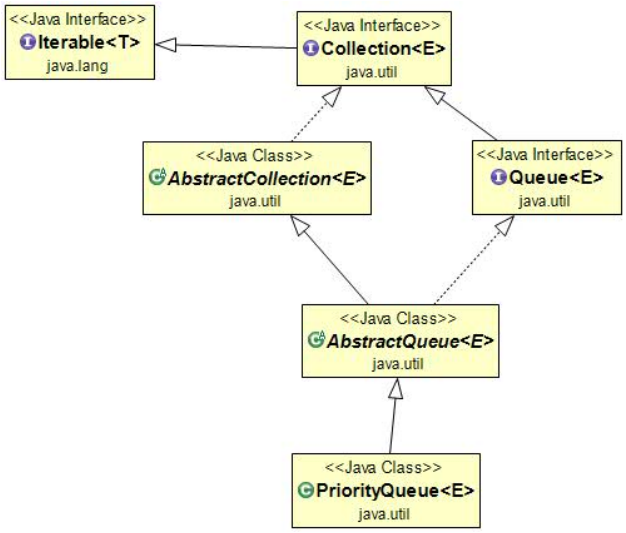

FIGURE 1

Java PriorityQueue class diagram. ("Priority Queue Java," JournalDev, accessed April 17, 2020, https://www.journaldev.com/16254/priority-queue-java.)

Methods inherited from class java.util.AbstractQueue

`addAll, element, remove`

Methods inherited from class java.util.AbstractCollection

`containsAll, isEmpty, removeAll, retainAll, toString`

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait`

Methods inherited from interface java.util.Collection

`containsAll, equals, hashCode, isEmpty, removeAll, retainAll`

FIGURE 2

Java PriorityQueue class inheritances. (Screenshot of "Priority Queue (Java Platform SE 7)." Oracle. 2018, accessed March 31, 2020, https://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html.)

```
function MINIMAX-VALUE(state, game) returns a utility value

  if TERMINAL-TEST[game](state) then
      return UTILITY[game](state)
  else if MAX is to move in state then
      return the highest MINIMAX-VALUE of SUCCESSORS(state)
  else
      return the lowest MINIMAX-VALUE of SUCCESSORS(state)
```

FIGURE 3

Minimax pseudocode. ("Games," Massey University of New Zealand, accessed March 31, 2020, https://www.massey.ac.nz/~mjjohnso/notes/59302/l05.html.)

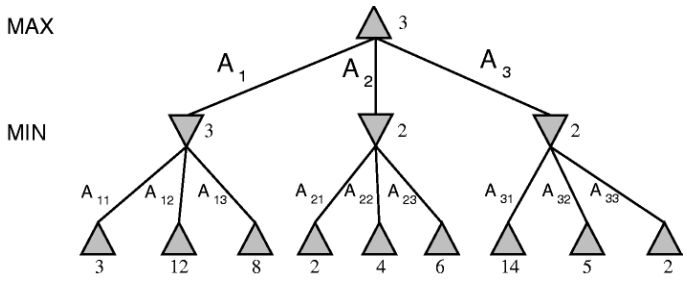

FIGURE 4

Minimax tree diagram. (Ibid.)

In implementing the minimax algorithm, a programmer might rely on either or both of these representational tools. Pseudocode is an informal schema, expressed in some more or less formalized version of human language, for how one might structure one's implementation, independently of any specific programming language—though different pseudocode for a specific algorithm might be more or less suited to your language of choice. The diagram in Figure 4, on the other hand, combines morphological representation with concrete values to illustrate both the *form,* or data structure, an implementation might take—here, a "tree"—and the manner in which values will propagate up the tree in accordance with the minimax algorithm. These two schemas are representations "above" the level of written code that nevertheless constitute part of the activity that I call programming. Implementation of minimax requires both "translating" pseudocode into, say, Python, as well as transcoding the continuous shape of the tree into discrete language.

While one might be tempted to characterize programming's layers as increasing in abstraction the further they move from the material reality of the machine, the tree diagram complicates this slightly. As an *example*, it is a kind of abstract template that allows the programmer to write a general implementation, which is able to function as such precisely because it is *more* specific or concrete than an actual implementation. The diagram, among other things, is therefore an abstract figure which nevertheless combines ideal and material, morphological and digital codes.

The navigation of the categories of abstract and concrete, form and content, is thus a key activity of computer programming. In his study of markup languages in *On the Existence of Digital Objects,*[23] Yuk Hui, similarly, observes the importance of the form/matter distinction, arguing that computing is predicated on classically Aristotelian hylomorphism. He opposes this to a Simondonian or Heideggerian understanding of technology in which matter gives rise to form; a good sculptor, for example, allows form to arise from the material at hand. Hui claims that this does *not* apply to computing, for "in the age of mass production… it is no longer a

[23] Markup languages like HTML and XML, are static, purely "descriptive" metadata schemes. They merely dictate the form of a digital object like a webpage; unlike Java, they can not manipulate objects or initiate processes.

question of human skill, but rather of the machine standards that create such forms."[24] Form, therefore, is superior to matter (data) in the digital object. Hui's analysis operates on the level of metadata schemes as the creators of digital objects, and for the most part elides their human origin. But programming, too, is an artisanal skill in the manner Hui describes; a good programmer chooses the data structure and/or algorithm most suited to the problem or data set at hand. As we will see, material constraints such as memory and processing power, which change depending on the hardware in use, are also a crucial consideration in any non-trivial program.

THE WORK OF SCALING

This brings us to a closer examination of the relation of the *programmer* to the figures and transformations across levels we have traced. I would argue, contra Hui, that the program is not incompatible with Simondon's critique of hylomorphism and his theory of technicity as "taking of form." In *On the Mode of Existence of Technical Objects,* Simondon relates the Aristotelian division between form and matter to the worker's alienation from both his labor and the technical world. In the typical situation of labor, in which an overseer who is removed from the material object commands the worker's manipulation of material, "the worker must have his eyes fixed on these two terms [form and matter], which he must bring closer together." Therefore:

> the attention is given to form and matter, not to *the process of taking form as operation*. The hylomorphic schema is thus a couple in which the two terms are clear and the relation obscure. Under this particular aspect the hylomorphic schema represents the transposition into philosophical thought of the technical operation reduced to work, and taken as the genesis of beings.[25]

"Work" entails a lack of understanding of the middle between form and matter, the process by which they are unified, the *mediation* which constitutes "the active center of the technical operation that remains veiled" in alienation. One might, therefore, ask: Is programming true technical knowledge, in which man "[represents] to himself the way of functioning that coincides with the technical operation" that is taking-of-form, or is it mystified work?[26]

---

[24] Hui, *Existence*, 61.
[25] Gilbert Simondon, *On the Mode of Existence of Technical Objects,* trans. Cecile Malaspina and John Rogove (Minneapolis: University of Minnesota Press, 2016), 248-249.
[26] Ibid., 249.

Here we return to our previous discussion of the division between process and object, again taking the PriorityQueue as our example. While the PriorityQueue appears as a static structure to which elements are added, it has a built-in sorting algorithm that maintains the order that gives the data structure its specificity. Thus, on a lower level, this organization or form is the result of a process that is permanently active. An instance of a data structure is less the static output of a one-time call to a function, an architecture that, once erected, is stable, than a continual taking-of-shape: each time an element is added to a queue, linked list, or graph, the algorithms which define its structure are called upon to maintain its organization.

Wendy Chun's critique of software as ideological rests on a similar argument that links the question of object and process to systems of visibility and a logocentric understanding of the performative power of language. The popular notion of digital technologies as transparent is possible only by making invisible the *process* of computation: the fact that computers "*generate* text and images rather than merely represent or reproduce what exists elsewhere."[27] When a user opens a file, they ignore that the image they see has been programmatically generated, and that what appears as a static form is actually being continually produced by light pulses within the screen. Both high-level languages and software more generally[28] allow the programmer or user to forget the material functioning of the computer, and imagine that their commands magically or fetishistically produce effects without mediation through the machine. Chun identifies this ideology, in which word is transparently converted into deed and human intentionality is transmitted seamlessly via language, with Derrida's concept of logocentrism. Here, code is a law that is executed automatically.

This move is coextensive with the mode of abstraction enabled by high-level languages, which spatialize (i.e., object-ify) what *are really*, on a lower level, linear, time-based machine processes: "Software as logos turns program into noun—it turns process in time into process in

---

[27] Chun, "Software," 27. Whether or not it is true that computers are commonly viewed as "transparent" is debatable but irrelevant here. Arguably, in 2019, anxieties over the "black box" of machine learning equal those over surveillance and systems of hypervisibility in popular discourse.
[28] Again, "software" or "interface" typically refers to a product for use by the non-coding public. But high-level languages (as well as any "layer" in our model of programming) are also interfaces in that they are mediations between human user and a lower level of machine technicity.

(text) space."[29] The level of technicity Chun privileges in her analysis is assembly code: programs written in a low-level language that can be run only on a specific processor architecture (unlike Java programs, which can be run on any machine). Assembly code, which consists of a series of commands to move data into different memory addresses, was once the only way for computer scientists to program their machines. Today, very little programming is done in assembly languages; high-level languages automatically compile programs into assembly and then into machine code, without intervention by the programmer.[30] In comparison with the recognizably Anglophone high-level languages, assembly code is only barely legible as derived from human language, or intended for human use.

The high-level language Python is distinguished from the assembly language x86-64 by a formal syntax that allows a non-linear structuring of the **flow of control** in the program. The top line of code in Figure 5 (following page) initiates a **while loop**: a series of steps that is performed again and again until the starting conditions are no longer met. The lines below `if` are executed only on the condition that the `start_list` queue is empty; otherwise, the lines below `else` are executed. The `for` keyword initiates a process in which a series of operations are performed for each element in the `list_of_items`. A **for-loop** is not only a handy automation for what one could write out for each element in the list (given that the programmer knows its size), it is an abstraction of a linear sequence into something like a concept of a movement: In this case, "add *each* `item` to a list." It is a linguistic shortcut (what Chun calls a metonymic "explosion of instructions"[31]), as well as a concept or figure tracing a movement or process of transformation.

---

[29] Wendy Hui Kyong Chun, *Programmed Visions: Software and Memory* (Cambridge, MA: MIT Press, 2011), 19.
[30] Though these compilers, or "interpreters," are, of course, themselves programs written by a programming language designer.
[31] Chun, *Programmed,* 41.

```
while (not start_list.empty()) and (not goal_list.empty()):
    if (not start_list.empty()):
        current_state = start_list.get()
        for item in list_of_items:
            if (item not in start_list):
                start_list.add(item)
```

FIGURE 5

Flows of control: example of *while*, *if*, and *for* in Python.

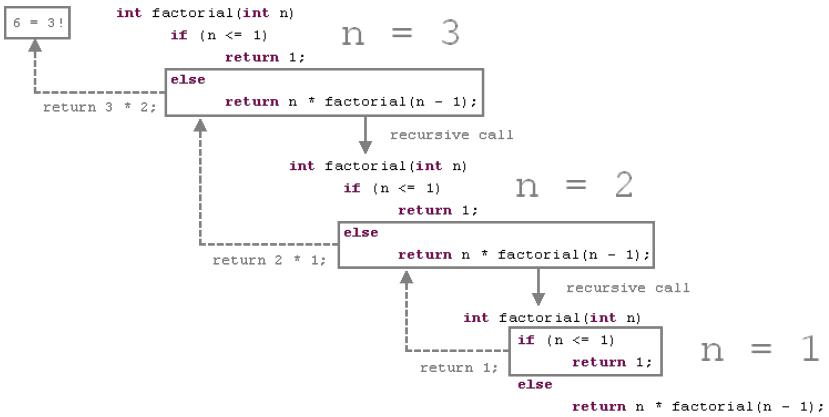

FIGURE 6

Diagram showing flow of control in a Java function to compute the factorial of a number *n.* ("How to perform recursion operation in Java," CodingSec, accessed March 20, 2020, https://codingsec.net/2016/09/perform-recursion-operation-java/.)

In Wolfgang Ernst's terms, programming on a high level is *chronopoetical* in that it involves the creation and manipulation of complex figures of time and causality. For- and while-loops are the two most elementary examples of time figures instantiated by the programmer: **iterative** structures that perform the same sequence a certain number of times or until a specified condition is met. **Recursive** functions take a similar but more complex shape, in which the **return** value of a function is a call to the same function. Figure 6 (previous page) shows a short, recursive Java program that computes the factorial of a number: the mathematical function `n = n*(n-1)*(n-2)*…*1`. The diagram shows the flow of control in a call to the function with the input `n=3`. The diagram's gray arrows, which go both up and down on the same line of code, demonstrate the unique temporality of recursion, in which one must represent to oneself the program's *linear progression* in time, which enables a gradual transformation of the input data, *as* a *cyclical repetition* of the same function. In this way, the temporality of programming is inextricable from the transformation of data. The minimax algorithm implemented as a search tree considered above is another tempo-spatial figure: an oscillation between two symmetrical functions, `max` and `min`, across multiple levels of a branching tree. Algorithms are temporal figures for the taking-of-form of structured data. As written documents, programs are linear, discretized representations of continuous, tempo-spatial morphologies.

While any of these programs could be written in x86-64, the strictly linear control flow of assembly language would make this exceptionally tedious. Moreover, the programmer's engagement with the data and operations at hand would be on a micro-scale of materiality—adding bits together, moving values from one slot in memory to another—to which it is difficult for most programmers to adapt. It is this loss of material closeness to the machine that Chun mourns. The abstractions that a language like Java enables increase the power of programmers at the expense of their knowledge of the functioning of the machine.

In the case of high-level versus assembly code, I would complicate Chun's argument that one is purely spatial, and the other purely processual. Assembly code relies on a division between object-like data and memory addresses, and the processual commands used to move them. Similarly, the abstractions enabled by high-level languages are not only spatial, but also temporal figures. But the question is less whether or not I agree with Chun's assertion that, from

the perspective of their technical reality, programs are processes rather than objects, and that, to a certain extent, high-level languages obscure this fact as a function of their representational systems. It is clear that the textual representation of a program in Java is necessarily static, whereas programs in action are dynamic linear processes not immediately visible to a human observer. But for us, the question is not whether any specific level of technicity is "more spatial" or "more temporal," nor whether certain features grant that level of technicity critical primacy in terms of theorizing digital technology as a whole. It is rather a question of whether one conceives of programming as restricted purely to one level, or as something that, in fact, traverses multiple levels. If it is truly "process" or movement that ought to be privileged in a discussion of programming, it is not because any given lower level of technicity is more properly described as dynamic rather than static. Rather, the "process" of programming is the movement or mediation between such layers of representation: in Manovich's terms, *transcoding*, in Simondon's, the process of taking form as operation.

While Chun develops an insightful analysis of the gendered history of computing and the human labor relations that would become technically inscribed in computer architecture, it is difficult to understand the ultimate import of her criticism of high-level languages as abstract. The "paradox" she identifies, in which high-level languages reduce "knowledge" (of detail) but increase power, is common to thinking. It seems trivially true that theoretical or conceptual work involves abstraction and hence reduction, generalization, or forgetting of certain material specificities, nuances, or implementation details.[32] The ideological nature of such thinking lies not in the (necessary) use of such abstractions, but in using them as if they were not abstractions.[33] This is the crucial point which Chun does not develop sufficiently. In order to argue that programming not only can, but *must*, as part of its specificity, mediate between abstractions and visualize the hidden materiality of the machine, we will turn to Wolfgang Ernst's theory of technology as creating temporalities.

---

[32] Kieran Healy, for example, observes that appeals to greater "nuance" in sociology prohibit the discipline's analytical power. Kieran Healy, "Fuck Nuance," *Sociological Theory* (vol. 35:2), 118-127.

[33] This is Chun's own definition of ideology, which she takes from Žižek's understanding of commodity fetishism as deriving its power from its use in practice, rather than its mystification in theory. Chun, *Programmed,* 52.

ECONOMIES OF TIME AND SPACE

Even in the age of highly abstract programming languages, effective, efficient programming is characterized by consideration of lower-level machine processes. Wolfgang Ernst illustrates this with his concept of electronic media as *time-critical* machines, technologies "in which minimal time processes represent a critical and thus decisive criterion for medial operativity."[34] Ernst's *time critique* operates at the sub-microscopic level of technicity, examining the time-based events of binary circuits and the clock pulses of the system clock to demonstrate their criticality to the functioning of the individual computer, as well as to its networking with other machines.[35] Digital media are not only semiotic machines, but time machines—indeed, on the microscopic level, even signals are time functions—that are *chronopoetical* in that they "generate original figures of temporal processuality" outside of the typical human understanding of time, as we have already examined in our discussion of recursion and other algorithms.[36]

While the high-level programming in which we are interested does not necessitate representation—in the sense of representing-to-oneself, imagining, or the German *vorstellen, vor sich stellen*—of the operation of logical circuits or other micro-temporal processes, most non-trivial programming tasks engage with the time-critical nature of computation. Programs are static representations of processes that, once written, are *run*. When optimizing for speed, the run-time or *time-complexity* of a program is a critical consideration, just as, when optimizing for space, the *space-complexity* is critical. Time and space are the resources that must be optimized in the economy of programming. The relationship between the two is often inversely proportional; thus, optimization techniques must be tailored to the problem at hand. A truism of software engineering is that "programmer time is more valuable than processor time": in the computing economy, the labor-time it would take for a programmer to fully optimize a problem is more valuable than a company's technical resources, especially given the rapid hardware advancements in processor power and memory. But if we are to theorize the specificity of programming as writing in relation to the *machine*, time-space complexity must form a key part of our analysis. While a programmer *could* solve a problem any number of ways, and there is not

---

[34] Wolfgang Ernst, *Chronopoetics,* trans. Anthony Enns (London: Rowman and Littlefield, 2016), 10.
[35]Ibid., 63.
[36]Ibid., vii.

always a "best-practice" solution, ignoring the question of efficiency erases the entire disciplinary thrust of computer science—it makes programming a purely aesthetic rather than a technical exercise.

Time- and space-complexity are usually expressed in **Big-O notation**, which describes the behavior of a function when the argument tends towards a particular value. An algorithm's time- and space-complexity is given a Big-O category as a function of the size of the data given to it as an input. Note that the fact that an algorithm's time complexity, arguably its most important feature, is defined as a function of its input, provides another example for the co-articulation of temporal and spatial figures in programming. For sorting algorithms, it makes sense to consider time-complexity from the perspective of best-case and worst-case input scenarios—the worst-case scenario being, for example, that the elements of the original input array are ordered in reverse order. An algorithm has a space-complexity as well as a time-complexity because, in the case of array-sorting, it must store the array data structure as well as any intermediate arrays or other values used during the process of sorting.

In order to sort an **array** of objects in Java, for example, one might write at least ten different sorting algorithms, all of which have different best-case, worst-case, and average-case time and space complexities. In many introductory computer science classes, students implement several if not all of these algorithms themselves in order to gain an intuition for time and space complexity, even though most high-level languages provide sorting as a built-in function. Even if most programmers may not do a formal complexity analysis for each program they write, having a sense of the time-space complexities of the built-in procedures of different high-level languages is imperative to writing efficient code. It is equally important to visualize certain aspects of machine architecture. For example, depending on the language, a recursive solution to a problem may cause a **stack overflow**—running out of space, or memory—far more quickly than an iterative solution.

As we will see in Chapter 2, these lower levels can and often are made visible to the programmer, for example by examination of stack traces or run-time analysis software. Programming in an "intermediate" language like C++, moreover, offers both high-level functionality and low-level manipulation of memory. But the majority of these lower-level

semiotic systems remain "hidden" beyond the "surface" of the Java text. High-level programming, therefore, is an activity in which multiple representational layers must be traversed simultaneously in the service of optimization or efficiency, one of the key technical — and, as we will see, political and economic—imperatives that structure modern programming.

* * *

To summarize, so far, programming:

* Traverses **layers or scales** of a representational system, often imagined as having hidden depths and visible surface,
* which is necessitated by the **material constraints** of computer hardware
* and the technical imperative of **optimization.**
* Divides and re-distributes across **heteromorphic** scales, rather than dissolves, **oppositional forms** such as form/content or object/process.
* Structures **flows-of-control** and creates diverse **chrono-spatial figures** or shapes in a **discrete, linear, static text.**
* Instantiates complex chains of **abstractions.**

We will continue to refine and add to this (both partial and overlapping) list of qualities that describe this figurative aspect of programming—an aspect which we will here call *diagramming.* While other software theorists remark on some of the same qualities, they subsume them under concepts or within configurations that are misleading here. By way of contrast: the eponymous "interfaces" of Alexander Galloway's *Interface Effect,* for example, are similar but not isomorphic to our diagram-programs. The preceding description of programming resonates with Galloway's emphasis on media as processes of mediation rather than static entities: viscous *middles,* in-betweens, or "thresholds, those mysterious zones of interaction that mediate between different realities."[37] But the "interface" seems ill-suited to express this: interfaces are the thin sheet or surface between two geometries. If it is a "middle," then paradoxically one without substance, and certainly without movement. Equally, I would argue that, as a study of new

[37] Alexander Galloway, *The Interface Effect* (Cambridge, UK: Polity, 2012), 8.

media, *Interface Effect* ultimately, in Simondonian terms, "leaves the relation obscure" between the relevant inter-facing realities (or, in my words, scales). This is symptomatic of Galloway's privileging of *users* over programmers; despite his insistence on the non-visual nature of new media and the importance of properly technological considerations of the digital,[38] his orientation towards the traditionally mediatic is betrayed by his choice of a word typically synonymous with "graphical user interface" to express his concepts.

Diagrams, like programs, capture dynamic processes in static, abstract representations. They usually illustrate structural relations between concepts, which may be related via, for example: "cause-and-effect" (time), "part-to-whole" (scale), or "communication" (space/ network).[39] For us, the the tree data structure—which we have relied on to illustrate various concepts above—is a particularly exemplary diagram because it can express both temporal flow-of-control or "decision structure" (minimax algorithm) as well as genealogies, inheritances, or levels of abstraction (OOP class hierarchy). Additionally, when diagrams function as *examples* with specific values, they play at the boundary of concrete and universal, or material and ideal. They thus encapsulate multiple scales at once, acting as the medium through which programmers transcode between different layers of abstraction and semiotic systems. Finally, I have chosen this word not only because diagrams are a common tool for practicing programmers, but "diagram" is a concept for several thinkers—chief among them Deleuze—whose writing on the concept will be useful in the coming pages.[40]

---

[38] Galloway, *Interface*, 17.

[39] I have neglected the concept of horizontality here, choosing instead to emphasize depth/layers/ scale. The figures described here are by no means an exhaustive account. One might also, for example, emphasize the lateral/horizontal relations of "communication" between the objects of object-oriented programming, or global distributed systems of computing.

[40] There exist also other, related concepts of "diagram" that I will not be able to incorporate here: Félix Guattari (see: *Chaosmosis*) and Paolo Virno (see: "Natural-Historical Diagrams: The 'New Global' Movement and the Biological Invariant") both use the word. There is also an entire field of research relevant to cognitive science and human-computer interaction known as "diagrammatic reasoning" (see *Diagrammatic Reasoning: Cognitive and Computational Perspectives,* ed. Janice Glasgow*).*

Topologies and terrain

In order to begin to reflect on the implications of describing programming as figuration, I'd like to point to several works that make clear the political stakes of programming's forms. The first is Luciana Parisi's *Contagious Architectures,* which takes up similar problems—the creation of novel chrono-spatial figures, parts and wholes, the category of abstraction, and the nature of thought itself—in the framework of Alfred North Whitehead's metaphysics. Via a reading of computational ideas of infinity, Parisi argues that algorithms create digital spatiotemporalities that "do not represent physical space, but are instead new spatiotemporal actualities."[41] Emphasizing the shift, in architecture, from notions of Euclidean distance and distribution of points on a Cartesian plane to distance as relational and probabilistic, Parisi describes "blob architectures," "surfaces of continuous variations, in which the physical distance between points has been transformed into a temporal variation."[42] The architectural trend she discusses—parametricism—designs structures using algorithms that incorporate environmental contingencies as input "parameters." This spatialization of temporal terms corresponds to an aesthetic form of "folds, morphologies, smooth surfaces, and real-time evolving structures" as well as a mode of control that "anticipates (and does not repress) change before it is actualized, and rather uses change to program new actualities."[43] In digitalized urban planning, methods of control and topologies are thus products of the same algorithm. Unlike Ernst, who highlights the chronopoetical digital movements that occur below human scale, Parisi describes, via examples taken from recent architectural practice and theory, the creation of spatiotemporal topologies *within which* humans live.[44] This helps us to recognize the implications of programming in terms of power and human subjects. As she emphasizes, architecture is only one application of the kind of algorithms she discusses; "other examples might include the relational architecture of

---

[41] Luciana Parisi, *Contagious Architectures: Computation, Aesthetics, and Space* (Cambridge: MIT Press, 2013), xiii.
[42] Ibid., 83.
[43] Ibid., 88; 85.
[44] Actually Parisi's primary interest is not humans at all—neither the programmer (or architect) who creates such flows-of-control, nor the citizens who inhabit them—but algorithms as constituting a nonhuman mode of thought via the incursion of random data into computer processing.

databases, the cultural, political and economic statements of search culture, the connectedness of social media, and the immediacy of data communication."[45] [46]

Justin Joque's *Deconstruction Machines: Writing in the Age of Cyberwar* thinks digital topologies on an even larger scale: the global communications networks that construct new temporal and spatial relations that overlay geographic and geopolitical space. Joque emphasizes that these are not *only* spaces of frictionless communication and instantaneity; "the spatial arrangement that arises is not a uniform closeness of all spaces on earth." Rather, the World Wide Web is defined as much by the new borders it produces: "blockages and slownesses" as well as Parisi's smoothness and speed.[47] Joque's analysis, to which we will return, examines these topologies in terms of cyberwar and sovereignty. Both Parisi and Joque's digital objects are larger-scale products of programming as we have described it. They illustrate that programming's diagrams are not merely aesthetic forms, but shapes that structure thought and behavior, flows of control that regulate capital and political investments.

Another variation on the theme of topology more directly relevant to our interest in reading/writing is Althusser's description of writing as constructing *fields* or *terrains* of knowledge. Although he emphasizes that this figurative language is merely metaphorical, here, Parisi's topologies offer a striking parallel.[48] For Althusser, one of Marx's major philosophical contributions was the insight that science occupies a terrain, or problematic, "within the horizon of a definite theoretical structure" that "constitutes its absolute and definite condition of possibility, and hence the absolute determination of *the forms in which all problems must be posed*."[49] Reading critically, as with writing, is the creation of such terrains—terrains which make themselves felt in the same manner as programming's topologies, by, however indirectly,

---

[45] Parisi, *Contagious*, xxiii.
[46] I would be remiss here not to point to Peter Eisenman's theory of the architectural diagram, which he derives from a reading of Derrida. The relationship of diagram to architecture Eisenman develops bears similarity to our theory of programming-as-diagramming, in which the diagram is both a practical tool and a figure that expresses the specificity of the discipline. Eisenman also used an initially pre-digital method of diagramming to conceive of new buildings themselves; in which case the diagram "is a mediation between a palpable object, a real building, and what can be called architecture's interiority." Eisenman, *Diagram Diaries,* (New York: Universe, 1999), 27.
[47] Justin Joque, *Deconstruction Machines: Writing in the Age of Cyberwar* (Minneapolis: University of Minnesota Press, 2018), 35.
[48] Louis Althusser, *Reading Capital* (London: New Left Books, 1970), 26.
[49] Ibid., 25.

structuring our political and cultural reality. As Ellen Rooney emphasizes in "Live Free or Describe," Althusser is a thinker of forms, of reading and writing as problems of form.[50] For example: the manner in which he contorts concepts of space in his description of a—from the perspective of Euclidean geometry, impossible—topology in which "all its limits are internal, it carries its outside inside it," that is "*infinite* because *definite,*" having "no external frontiers separating it from nothing, precisely because it is *defined* and limited within itself"[51] is reminiscent of the complex algorithmic forms Parisi describes, and that programmers instantiate in text. In subsequent chapters, we will examine further points of similarity, as well as difference, between programming and symptomatic reading.

The Diagram

Gilles Deleuze is another philosopher who might be called a "thinker of forms"; his lovingly described figures—the rhizome, for example, or the plane of immanence—do much of the conceptual work of his philosophy. We will focus on the two primary forms elaborated in his book on Foucault: here, the diagram, and, in Chapter 3, the fold.[52] Foucault himself uses the term "diagram" in *Discipline and Punish*: the Panopticon is a concrete assemblage animated by a more general diagram, a "mechanism of power reduced to its ideal form" and "detached from any specific use."[53] The diagram, Deleuze continues, is "a cartography that is coextensive with the whole social field. It is an abstract machine."[54] These diagrams operate at the macroscopic level: They express one of multiple configurations of power relations of a society at a given moment in time. For this reason, they do not have a specific instantiation or use in the way a

---

[50] Ellen Rooney, "Live Free or Describe: The Reading Effect and the Persistence of Form," *differences* (vol: 21:3), 132. My reading of Althusser here, and my project in general, is highly indebted to Rooney's work on form and critique.
[51] Althusser, *Reading,* 27.
[52] Deleuze also describes a somewhat different "diagram" in the context of Francis Bacon's painting in *Logic of Sensation*. The diagram of *Foucault* seems nearly identical to the concept of *dispositif* or apparatus that Deleuze discusses in his essay on Foucault: "What is a Dispositif?" in *Two Regimes of Madness,* trans. Ames Hodges and Taormina, Mike, ed. David Lapoujade (Cambridge: MIT Press, 2006), 338-348.
[53] Michel Foucault, *Discipline and Punish,* trans. A. Sheridan (New York: Pantheon, 1995), 205.
[54] Gilles Deleuze, *Foucault* trans. Séan Hand (Minneapolis: University of Minnesota Press, 1988), 34.

program does. At the same time, however, Deleuze leaves open the possibility of the diagram operating at multiple scales.

> The diagram acts as a non-unifying immanent cause that is coextensive with the whole social field: the abstract machine is like the cause of the concrete assemblages that execute its relations; and these relations between forces take place 'not above' but within the very tissue of the assemblages they produce.[55]

The diagram can be apprehended both as an abstract form, and as that internal engine which powers the concrete technologies of power. In this sense, Deleuze argues, Foucault saw the Panopticon not only as a technology specific to prison and systems of surveillance, but also as having an "abstract formula": namely, no longer merely "'to see without being seen,' but to impose a particular conduct on a particular human multiplicity." The diagram moves on a larger, higher, or more abstract level than the concrete assemblage, and is their cause. But these relations are not to be understood in the simplistic scalar senses of whole/part or large/small nor linear cause —> effect; rather, the "larger" cause is *immanent* to the "smaller" concrete assemblages.

To generalize this multi-scalar characteristic of the diagram, we might say that, as a figure, it is inherently one which cuts across layers—a dia-gram, a diagonal. Deleuze also identifies diagonality as a key feature (one might say a movement, or a method) of Foucault's archaeology. The conceptual architecture constructed in *The Archaeology of Knowledge*, as Deleuze tells it, is a stratification of semiotic systems into heteromorphic layers and thresholds, in which the statement is characterized by "the shape of the whole curve to which [its elements] are related,"[56] governed by rules or patterns of formation that operate transversally. Deleuze emphasizes not only figures and forms, but the movement between layers: Manovich's transcoding. Programs have a different *content* than statements or discursive formations: For instance, the problematics of the position of speaking subject and repetition in the Foucauldian sense are foreign to code (a point to which we will return in the conclusion). But they are strikingly isomorphic. The archaeologist, like the programmer, "must pursue the different series, travel along the different levels, and cross all thresholds; instead of simply displaying

[55] Deleuze, *Foucault*, 37.
[56] Ibid., 4.

phenomena or statements in their vertical or horizontal dimensions, one must form a transversal or mobile diagonal line." Like the archaeologist, the programmer tracks heteromorphisms: discontinuities and lines of fracture between layers.

Beyond this formal similarity, the Deleuzian diagram allows us to tie programming more tightly to the forces that animate the whole range of Foucault's work: knowledge, power, subjectivity. For the Foucault of *Discipline and Punish*, Deleuze argues, there are two fundamental forms: the visible and the articulable, system of light and system of language, form of content and form of expression.[57] The rather idiosyncratic meanings Deleuze gives to the concepts of form, matter, and functions, are not particularly relevant here, nor is the lengthy and acrobatic paraphrase required to trace the exact contours of the highly complex shape Deleuze calls "diagram." But, broadly speaking, the diagram is "a display of the relations between forces which constitute power."[58] In the passage from diagram to concrete assemblage, the two forms discursive/non-discursive or articulable/visible are differentiated. This bifurcation is the operation of *knowledge*, of which *power* is the immanent cause.[59] The diagram represents the historically specific situation of power, knowledge, and discursive systems in relation to one another.

Programming is the process of constituting exactly these relations: the capture of the non-discursive (data) in specific discursive or representational forms (a database, an algorithm), the instantiation of many concrete assemblages patterned by the same diagram—and all of these operations guided or animated by a particular expression of power. This force that diagonally traverses every layer of programming—the meta-diagram of both programming and capitalism—is *optimization*.[60] The imperative to optimize time and space on the lowest level of technicity is the same "relation between forces that constitute power" that patterns the lives of individuals.[61] As the technical mandate of programming, the abstract formula of optimization is actualized in each program, just as it structures production and society on a larger scale. Although we are

---

[57] Deleuze, *Foucault,* 33.
[58] Ibid., 36.
[59] Ibid., 38.
[60] Beyond optimization, another technical imperative one might emphasize, as Joque does in *Deconstruction Machines,* is that of "security." This might also be easily tied to a diagram structuring both programming and political or social reality.
[61] Deleuze, *Foucault,* 36.

limited in our scope to consideration of the former, microscopic scale, the theme of "optimization" might be identified in both economic and cultural phenomena from labor management practices at Amazon's fulfillment centers to the spiritual and physical exercises aimed at feminine self-actualization Gwyneth Paltrow prescribes on her TV show *The Goop Lab*. If knowledge is the concrete assemblages through which the diagram operates, then, in the case of programming, the diagram is the movement of power that connects all the layers of a program at various points, the force that animates them, the *running* itself which unifies forms in an expression of force. It is, literally, *flow-of-control*. Deleuze's diagram allows us, therefore, to emphasize that, in programming, taking-of-form is always a question of power.

By way of comparison, one might easily point to the—sometimes obviously, sometimes insidiously—violent structures of domination computer technologies instantiate and uphold across the world. But Deleuze also opens up another possibility, one that will bring us back to the question of writing:

> There is no diagram that does not also include, besides the points which it connects up, certain relatively free or unbound points, points of creativity, change and resistance…For each diagram testifies to the twisting line of the outside spoken of by Melville, without beginning or end, an oceanic line that passes through all points of resistance…From this we can get the triple definition of writing: to write is to struggle and resist; to write is to become; to write is to draw a map: 'I am a cartographer.'[62]

I want to dwell on this final comparison between writing and cartography. If Foucault is Deleuze's "new cartographer," the writer, the drawer of maps, then he is also the drawer—or is it the creator?—of diagrams themselves. Here, as with Althusser, the activity of writing is that of constructing new topologies or "dimensions": in the case of Foucault, a "diagonal dimension, a sort of distribution of points, groups, or figures that no longer act simply as an abstract framework but *actually exist* in space."[63] Here, we might note a slippage between the representation, or recording, of the diagram and its creation as an "actual" entity—the diagram, we recall, is the *display* of the relations between forces, but also the operation of these forces themselves—that maps well onto the dual semiotic and functional nature of computer programs.

---

[62] Deleuze, *Foucault,* 44.
[63] Pierre Boulez, *Relevés d'apprenti* (Paris: Seuil, 1966), **quoted in** Deleuze, *Foucault,* 22. Emphasis mine.

In the same movement through which Deleuze opens up the possibilities for "creativity, change, and resistance" in the diagram, he relates it closely to the activity of writing. The diagram that is *created*—rather than merely drawn or recorded, as in archaeology—through writing is a diagram of a particular kind: a *fold*, or what Foucault calls a technology of the self. This passage points us towards a comparison between programming and writing, in fact, between programming and Foucault's own writing (which we might place under the loose category of "critique"), which we will explore in the following chapters.

CHAPTER 2. BUGS AND MATERIALITY

It's not a bug, it's a feature.
- Programming proverb

A key strategy of Derrida's deconstruction of Western metaphysics is his foregrounding of the "organization of the signifier" rather than the "semantic, that is, thematic, *content* of a text."[64] His attention to, for example, the *espacement* of written words (which might also be called their discreteness or digitality), emphasizes the *forms* of representation: that which, in logocentrism, is deemed accidental or marginal to the true function of language: namely, the fluid transmission or communication of meaning. Here, "form" is thus co-extensive with the *materiality* of the signifier. This interest in materiality gives rise, in Derrida's writing, to a multiplicity of conceptual figures deriving from, and devoted to, technologies of inscription: "I have always written, and even spoken, *on* paper—at once about paper, on the surface of paper, and with an eye to publishing a paper. Support, subject, surface, mark, trace, gramme, inscription, fold…"[65] His style is marked by a strategic rhetorical deployment of these formal qualities: In "Limited Inc a b c…," for example, he puns with the sensible, sonic qualities of words, as well as their spelling, to deliver a scathing response to Searle's defense of traditional linguistics.[66] This play with the materiality of signs is also often done via programmatic processes. In the "Envois" of *The Post Card*, he omits bits of text of indeterminate length and replaces them with lines of 52 spaces, determined via "a cipher that [he] had wanted to be symbolic and secret—a clever cryptogram, that is, a very naïve one, that had cost [him] long calculations."[67] In this way, Derrida foregrounds the formal qualities of language forcibly; mediation is made sensible via mathematical or mechanical means. Computer code, however, does not require such de-

---

[64] Jacques Derrida, *The Post Card: From Socrates to Freud and Beyond,* trans. Alan Bass (Chicago: University of Chicago Press, 1987), 424. Emphasis mine.
[65] Jacques Derrida, "Paper or Myself, You Know… (new speculations on a luxury of the poor)," trans. Keith Reader, *Paragraph* (vol. 21:1), 1.
[66] Jacques Derrida, "Limited Inc a, b, c…", *Limited Inc*, trans. Samuel Weber and Jeffrey Mehlman (Evanston: Northwestern University Press, 1988).
[67] Derrida, *Post*, 5.

naturalizing interventions in order to make felt the intertwined categories of materiality (physical substrate) and form (structure or syntax) that enable the semantic "content" of computational inscriptions. Rather, programming necessitates the manipulation of these qualities.

This chapter leaves aside Deleuze's diagram in order to describe programming-as-writing at a more granular level, on the scale of the operation of individual signs themselves. I draw primarily on Derrida's critique of logocentric philosophies of language in "Signature, Event, Context" and "White Mythology." Programming is a writing in which the materiality of the technology of inscription cannot be forgotten, in a semiotic system in which meaning is variable and unstable, and where failure of communication is a constant reality. If logocentrism entails a kind of suppression or malignment of mediation, programming materializes and cannot escape its interferences. Via this observation, a secondary argument will thread through this chapter: that programming, in its "thinking within" mediation, bears a formal similarity to certain aspects of deconstruction itself. The importance of diagrammatic thought (thinking across scales of technicity and figurations of time and space), will also find further illustration here via technical examples in C and Python.

Supplement, support, software

In comparison with the fairly modest technologies of pen and paper that have supported centuries of written language, the software tools that enable programming are elaborate assemblages. As Wendy Chun demonstrates, high-level languages themselves are a kind of—ideological—software, or mediation, without which the growth of the tech industry is unimaginable.[68] Her aversion to the semiotic abstractions of high-level languages over the less human-readable, more "material" systems languages veers close to a nostalgia for a pre-prosthetic origin within a technology itself. As discussed in Chapter 1, systems languages are already an abstraction from the electronic signals that, following Chun's logic, might be said to "really" constitute the technical reality of computers. Oddly, an underlying strand of Chun's argument seems to be a complaint that software (or high-level languages) make computers more *legible* and therefore

[68] Chun, "Software," 38. This is also the argument she develops in *Programmed Visions.*

usable, when such usability is arguably central to all technology. Not only that, but language itself, as Derrida emphasizes, is itself technological.[69] Programming dramatically foregrounds this technological character of writing, as well as the infinite proliferation of prostheses of all kinds.

In a word-processing software like Word, certain combinations of signs may conjure up a thin, squiggly line, prompting the spell-checker to ask: "Did you mean..?" (Derrida describes submitting his writing of *Circonfession* to such injunctions—"The paragraph is going to be too long; you should press the Return button"—and thereby ending every paragraph after roughly 25 lines.[70]) The text editors used for code offer many such "supports" to writing. On the most rudimentary level, syntax highlighting allows words to appear in specific colors depending on their function: Identifiers might be pink, user-defined functions green, functions provided by the programming language blue, and punctuation, like brackets and semicolons, white. The margins of the text window are scored by line numbers, allowing the programmer to locate specific lines of code in the event of an error message, which typically returns the line at which the program prematurely terminated execution. More sophisticated **integrated development environments** might offer a range of tools designed for a specific language, from **debuggers** (testing software that allows the programmer to step through the execution of a program line by line) to convenient depictions of class hierarchies.[71]

Many of these tools are, in a sense, visualizations: They not only supplement the purely linguistic digitality of language with sensorial cues (such as color), or analog representations (such as class diagrams), but visualize what is invisible or opaque (the inner workings of the machine).[72] A debugger, for example, allows the programmer to set **breakpoints** within the temporal execution of the program, at which they can inspect the values to which different variables correspond. Figure 7**,** a screenshot of the integrated development environment Eclipse's debugger, illustrates the rich semiotic machinery with which professional software development

---

[69] Jacques Derrida, *Of Grammatology,* trans. Gayatri Chakravorty Spivak (Baltimore: Johns Hopkins University Press, 2016), 8.
[70] Jacques Derrida, "The Word Processor," *Paper Machine,* trans. Rachel Bowlby, 22.
[71] Such as the class hierarchy diagram examined in the previous chapter.
[72] The "visuality" of software is precisely what Chun takes issue with. See Chun, "Software."

is supplemented.[73] The four quadrants are different representations of "where" or "when" you currently are in the program's execution. The bottom-left window is the actual Java code written by a programmer, whereas the window to its right is Eclipse's schematic rendering of the program's functions. The top-left window shows the list of function calls performed by the machine: the kind of linear or time-based representation of the computer's technicity for which Chun advocates. The top-right shows the values of different user-defined identifiers at this point in the program's execution: For example, at this temporal-spatial "moment" in the code, radius is set to 15.

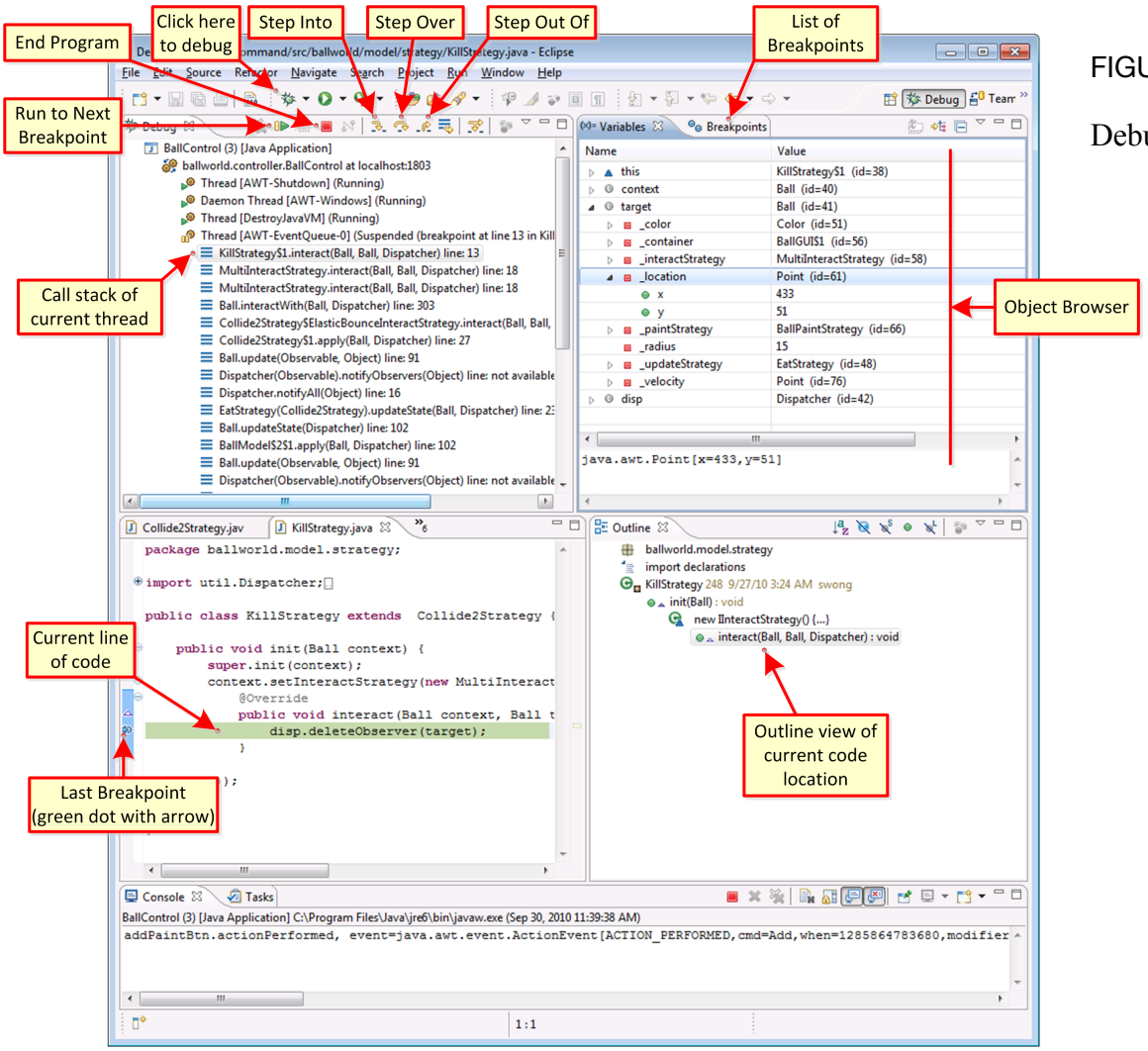

FIGURE 7

Debugging in Eclipse.

[73] Stephen Wong, "Eclipse Debug Perspective Screen Shot," COMP310, Rice University, 2017, accessed April 10, 2020, https://www.clear.rice.edu/comp310/Eclipse/debugging.html.

Profilers, on the other hand, are programs used for time- and space-optimization that *quantify*. Profiling returns a measurement of, for example, the time spent in each part of the program, thereby alerting the programmer to functions that most require optimization. With this quasi-empirical instrument, one approaches the program in the same way a natural scientist would her object of study. Unlike the mathematical formulas to which they are often compared, programs take on a life of their own, requiring data analysis in addition to diagramming. Although the information provided by debuggers or profilers *could* be logically deduced from the code itself—for example, by performing a theoretical time-complexity proof—their widespread usage exhibits the extent to which programs are often opaque to their own makers.

As Derrida points out with concern, spellcheck may harden certain semiotic structures, giving Word's conventions the appearance of concrete constraints. But in Word, unlike a code editor, corrections are ultimately only suggestions. Spelling and grammar "mistakes" are rarely absolutely prohibitive to communication. In code, however, the slightest deviation from the prescribed syntax will cause the intended message to miss its recipient, the machine; the interventions made by code editing software—for instance, the automatic insertion of a missing parenthesis—are therefore more than mere suggestions. Moreover, software like Word supplements a presumed natural fluency with language. But writing even trivial syntactically correct programs without the aid of software—say, on paper, or on a whiteboard—is difficult, and, for large-scale projects, virtually unimaginable. For these reasons, the "paper" of programming—the interface through which inscription is carried out—is more prominently present than the more easily marginalized physical supports of writing (whether parchment, or Pages). Programming is a writing of which the prosthesis is conspicuously visible—a reality produced, as we will explore later in this chapter, by the inescapable incursions of error.

Variables and polyvocality

By providing language-specific features like syntax highlighting and methods of tracking variable assignment, these software supplements might be characterized as offering a visualization of the semantic *context* in which the words of a high-level program are run. Unlike

in human writing, this context is, as Derrida would say, "fully saturated."[74] The *situation* in which high-level code is written fully determines its meaning. This total contextual positioning is produced across a variety of scales: Code is written in a certain language, stored in a specific file location, and run on certain hardware. It is embedded in a network of countless other programs (compiler, interpreter, operating system), and can be supplemented by external resources (like imported libraries). These technologies, together, create the conditions of its functioning and enable complete specification of its meaning. There thus exists the structural possibility (and desirability) of deterministically produced correct or expected behavior. Here we find the crucial difference between code and Derridean writing: given that a context *were* known completely by the human programmer, the behavior, or meaning, of a certain line of code *would* be unambiguously determinable in every instance.[75] Complete comprehension of these different spatial and temporal contexts, levels, or scales, however, is practically impossible. This section describes several aspects of variable assignment in order to illustrate the scalar or diagrammatic agility required to create "meaning" in programming and the manner in which problems of materiality manifest, and to give a sense of the thick semiotic webs that, as we will see, inevitably trip up even the most expert programmer.

MEMORY ADDRESS: MEANING AND MATERIALITY

In "White Mythology: Metaphor in the Text of Philosophy," Derrida links the philosophy of language inaugurated by Aristotle to Western metaphysics via the concept of metaphor. Axiomatic to logocentrism is the original, sensible image to which corresponds exactly one signifier: "to be univocal is the essence, or rather the *telos* of language."[76] Metaphor is therefore dangerous because it "may set off an errant semantics. The sense of a noun, instead of designating the thing which the noun should normally designate, goes elsewhere." Metaphor

---

[74] Jacques Derrida, "Signature, Event, Context" in *Limited Inc,* 3.
[75] This is the aspect of code most frequently commented on by scholars in comparing programming to writing. Hayles, for example, opposes the materiality of code to Derrida's grammatological theory of writing because of this "clarity" and "unforgivingness." See N. Katherine Hayles, *Writing Machines* (Cambridge: MIT Press, 2002), 48. We will complicate this argument further towards the end of this chapter.
[76] Jacques Derrida, "White Mythology: Metaphor in the Text of Philosophy" trans. F.C.T Moore, *New Literary History* (vol. 6:1), 8, 48.

both produces an excess of signs—one sign being substituted for another, and thus pointing to the same referent—and exposes the existence of concepts or referents which do not have a signifier proper to them.[77] In a vocabulary more proper to computer science: The logocentric model of language assumes a bijection, a one-to-one relationship, between signs and referents.

The idea of a sensible, pre-linguistic entity which is *expressed* by a sign is foreign to computer science; although the linguistic sign in code is inextricably linked to a material operation, programming languages encapsulate a formal system which is *non-representational*—it bears no relation to an external non-semiotic reality. It is not a model, but a world unto itself. This is only one sense in which many of the problematics of metaphor Derrida links to philosophy in "White Mythology" are inapplicable here. But the concept of metaphor, in confounding an "easy" or self-present system of meaning through the introduction of polyvocality, is a way of foregrounding the semiotic system itself, in a manner that holds certain similarities to code. Programming is predicated on the substitutability and proliferation of signs, rather than their stability. The relationship—which we will provisionally call "meaning"—between the surface-level sign (the word typed by the programmer) and its interpretation by the machine is not only inflected by the various contexts at play, but, in concrete instances, actively created by the programmer through the process of variable assignment or name-binding.

We will first examine name binding—the assignment of data to identifiers—in the low-level language C. **Identifiers** are the lexical tokens by which a programmer names various components of the program, the most basic of which is a variable. One might write, for example, `x=1`, thereby assigning the identifier `x` to the variable value of 1. Each identifier corresponds to a specific location in memory, a **memory address**, where the value is stored. Technically, the "meaning" of an identifier is this location, the "contents" of which can be modified. This is particularly visible in C, in which variables can also be manipulated via **pointers**, which hold a memory address accessible to the programmer. Figure 8 illustrates the process by which a pointer, `p`, is assigned a value. First, the pointer variable `*p` is instantiated without a value: It is **uninitialized**. It is then assigned to the memory address of the reference variable `a`, which has a value of 4. One can access both the memory address and the values of `*p` and `a`. Semantically,

[77] Derrida, "White," 32.

they are identical, but depending on their interaction with other parts of the program, it might be preferable to use one or the other. One can create multiple pointer variables that point to the same address or value, for example via pointers which point to other pointers. There are many reasons why identifiers that refer to the same value might be made to proliferate in this way, including to allow multiple programmers to work on the same project, and the necessity of reassigning identifiers to new values in different parts of the program.

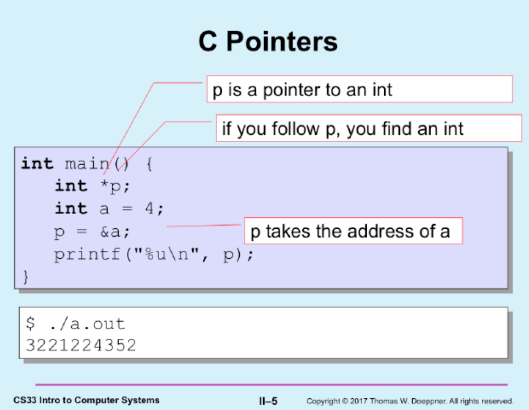

FIGURE 8

Assigning pointer values in C. (Thomas W. Doeppner, lecture slides for CSCI0330: Introduction to Computer Systems, Fall 2017.)

We are considering high-level, rather than systems programming. But, as argued in Chapter 1, just as a high-level program relies on transcoding into other languages, effective high-level programming requires an understanding of lower-level system behavior. I have provided this technical example in C to emphasize the material reality of programming's signs. Unlike in the purely symbolic language of mathematics, the numbers 1 and 1.0 are not equivalent in computer science: They take up different amounts of space in memory. Although the specifics of memory management are handled "under the hood" in high-level languages, programming is always the creation of representational forms from ultimately material transformations of data.

MEANING IN SPACE AND TIME

The materiality of the memory address in comparison with the pointer or identifier highlights the different status of "meaning" across scales of technicity. But one can also track varying contexts of meaning within the dimension of the high-level program. If one identifier can point to or hold

multiple values, this is because it traverses a multiplicity of different *spatiotemporal coordinates* that establish context and determine its meaning.

Spatially, a name binding (variable) is located within a certain semantic *scope:* the area of code in which an identifier is bound to a certain value. Outside of this scope, the identifier x might hold a different value—say, a list rather than an integer—or no value at all. In the Python example shown in Figure 9, the variable y has a scope that is **global**: Its value can be accessed from any point in the program (the matter of whether, where, and how its value can be modified depends on the programming language). The identifier x is defined twice: In my_function_1, it is a **local** variable with an initial value of 0 that is modified within the function. In my_function_2, x is the name of the **argument parameter** of the function: Its value is dependent on the argument **passed in** when the function is called. For example, calling my_function_2(3) would temporarily assign x to 3, and return the value 103. Outside of these two functions, the identifier x is **out of scope**—it holds no meaning, and writing the line y=y+x would throw an error message. A variable is also given meaning by its positioning in the temporal or linear execution of a program. my_function_1 consists of an iterative loop in which 1 is added to the integer x ten times (resulting in a return value of 10). It is this dynamic transformation of meaning that is often most difficult for programmers to track. The debugger in Figure 7, which materializes certain aspects of the program's context at a frozen moment in time, is meant to provide insight into these operations by providing four different ways of representing the contextual positioning of the program at a single breakpoint.

```
y=100

def my_function_1():
    x=0
    for i in range(10):
        x = x+1
    return x

def my_function_2(x):
    return x+y
```

FIGURE 9

Two Python functions in which x and y operate as argument parameters, local variables, and global variables in different contexts.

Python uses a concept of lexical or static scope. Older programming languages operate with dynamic scope, in which the binding of an identifier is interpreted not by its location in the written program, but purely by the point in execution in which the program finds itself.[78] Because the latter is no longer used (in an instance of the historic trend in programming languages away from linear, time-based representation and towards spatialization that Chun describes), I elide its discussion here, but scope is an interesting case study for the infinite ways in which programming languages can differ in their implementation. The manner in which "meaning" varies under the hood across languages also contributes to the "denaturalized" semiotic sensibility necessary for programming I argue for here. Unlike most humans, most programmers achieve fluency in many languages (Brown's introductory Computer Science sequence teaches four): Not only must programmers think within the constraints of mediation, but they bear no illusion that, within the discipline of computer science itself, there is any one "natural" language or method of representing technical systems.

This cursory summary of identifiers, variables, and memory ought to sketch some of the complexity of computer science's semiotics, which far exceeds a simple, stable bijection between signs and referents, i.e., between variables and values, or objects in memory. Many variables can refer to the same object, and a variable's meaning can pass through many mediations, via assignment to a string of many other variables, before "arriving" at its actual value. (Note that, while this chain of "deferral" is potentially very lengthy, it is not indefinite or infinite.) The value of a variable can change countless times throughout a program, as well as, as in the case of uninitialized pointers, hold no value at all. In this way, programming languages are hardly univocal. Nor is meaning natural or fully "self-present" to the programmer, rather, it is constantly mediated, channeled through a multiplicity of signs, passed by copy, passed by reference, modified, nullified, made anew. Variables are always being assigned new values, swapped with each other, and even declared void of meaning by assignment to the value NULL. Programming

[78] Shriram Krishnamurthi, "Scope," *Programming and Programming Languages*, accessed March 31, 2020, https://papl.cs.brown.edu/2018/Interpreting_Functions.html#%28part._.Scope%29.

is characterized by polysemy, excesses and vacuums of meanings which are constantly duplicated, copied, and rearranged with regard to their signs.[79]

Variables in programming and metaphors as described in "White Mythology" are not equivalent—most importantly, variables do not hold the same status of both grounding and destabilizing the function of their language—but both are linguistic operations through which the thickness of language as a medium becomes apparent in different ways. If logocentrism involves the illusion of meaning as self-evident, self-present, and self-identical, ideal and therefore divorceable from its expression in a certain medium, then the vagaries of variable assignment, like metaphors, confirm the difficulty with which representations are made: the material supports from which they are inextricable, their shifting value based on context, and therefore the unthinkability of meaning outside of language. Another name Derrida has for logocentrism is phonocentrism: the "absolute proximity of … voice and the ideality of meaning" that presupposes the possibility of wholly pre-figured concepts that are transmitted via a clean, "loss-less" communication.[80] There is, of course, no speaking in computer science; equally, as we will explore in the next section, the persistence of failures of communication in coding via bugs introduces a rupture between the intentions or "voice" of the coder and the effects of code, making impossible a phonocentric understanding of programming.

Failure: Bugs and edge cases

In code, even the smallest typo in language syntax (analogous to a spelling or grammar mistake) can lead to failure during the program's execution: what one might call the "non-arrival" of intended meaning at the program's destination. In "Signature, Event, Context," Derrida locates such mis-firings as not only a possibility, but a structural feature of writing. Because the permanence of the written sign "carries with it a force that breaks with its context, that is, the collectivity of presences organizing the moment of its inscription," allowing also its iterability or citationality, the "infelicities" of speech that J. L. Austin relegates to the margins of his theory of

[79] In passing, Manovich links this variability of signs within the computer to broader social shifts in which every cultural "constant" has become substituted with a "variable." *Language,* 43.
[80] Derrida, *Grammatology*, 12.

language are actually an "essential risk" of communication.[81] This possibility for failure or existence of indeterminacy is, for many theorists of software, what is impossible to locate in code.[82] As Justin Joque notes, in order to establish a (misleadingly) firm division between computer and natural languages, such readings must caricature the former as fully understandable and self-present (or vice versa).[83] Because the inevitability of failure in code is crucial to my description of programming, I will briefly outline two interesting, different approaches to this theme to clarify my own argument.

Joque's *Deconstruction Machines* considers several instances of attacks on computer systems in which user input is designed in order to exploit flaws or inconsistencies in the code.[84] Because the hacker's modus operandi is to "read and write systems against themselves," Joque compares the semiotic strategy of cyberwar to Derridean deconstruction.[85] In code's vulnerability to attacks, Joque identifies the same kind of failure that Derrida does in written language:

> The possibility that the execution of a program could produce an unexpected result, either accidentally or as the result of malicious code, injects into computation the possibility that it differs from itself. The exact same code run at a different time or in a different place bears the possibility of different results. This potential difference simultaneously indicates a necessary deferral: we can never know what a program does or means until it is executed.[86]

This is also the spirit of my argument. Joque and I approach the same issue from opposite, though symmetrical, positions: Whereas I will deal with errors that occur during debugging (i.e., before a program is made public), Joque considers errors that may be willfully introduced by a second programmer, who reads the potential flaws in the first program in order to conduct a cyberattack. Here, however, I would make a finer distinction than Joque does: Although "we" may *rarely* ("never" is an overstatement) know exactly what a program will do until it is executed, this is not to say that the program itself behaves inconsistently. As explained earlier in

---

[81] Derrida, "Signature," 9, 15.
[82] See, for example, Hayles's comparison of code to Derridean différance in *My Mother Was a Computer* (Chicago: University of Chicago, 2005), 47.
[83] Joque, *Deconstruction,* 20-21.
[84] For example, if a form expects only certain number or certain types of characters to be entered as login information, and an attacker enters additional or abnormal characters, they might be able to overwrite the executable code of the form itself, replacing it with malicious code.
[85] Joque, *Deconstruction*, 75.
[86] Ibid., 76.

this chapter, unlike writing, the "context" of every program is fully saturated in that it is completely deterministic. The fact that human computer users can never fully anticipate a program's behavior due to their insufficient grasp of its technical functioning is not the same feature of writing Derrida describes in "Signature, Event, Context." For Derrida, the possibility of the non-arrival of its meaning is constitutive of writing's structure: This can not be said of either bugs or cyberattacks.

One approach to conceptualizing such an interior condition of computational failure can be found in Beatrice Fazi's *Contingent Computation: Abstraction, Experience, and Indeterminacy in Computational Aesthetics,* via her re-reading of several fundamental texts in computational theory: Kurt Gödel's incompleteness theorem and Alfred Turing's proof of the existence of incomputable functions. These proofs are usually interpreted as demarcating the limits, and thereby limitations, of formal axiomatic systems. Fazi, however, uses Turing's proof, which addresses the issue of defining the infinite in finite terms, to locate an "opening up" to infinity *within* (rather than at the limits of) computation. Turing's proof shows that no general algorithm can exist to decide whether another program will run infinitely or terminate.[87] In short, "computation is made up of quantities, yet these quantities cannot be fully counted."[88] Although Fazi does not put forward this argument herself, one might compare this insight to Derrida's exposure of indeterminacy of meaning in writing. For Derrida, this destabilizing of traditional philosophies of language is also a critique of a Western metaphysics of presence. Similarly, for

---

[87] For example, a **while** loop whose condition to continue is **true** will run forever, which is not necessarily undesirable. Distinguishing such infinite loops from statements that will terminate is known as the *halting problem*.

[88]Note that "computation" is to be understood here less in terms of specific digital technologies, and more as any method of systematizing reality through an abstract, formal structure composed of discrete quantities. (Fazi, *Contingent Computation* (London: Rowman and Littlefield, 2018), 47). Fazi's argument, which is similar to Parisi's in *Contagious Architectures,* relies on a Whiteheadian reading of already-complex computational theory, for which reason I elide in-depth discussion of it here. She argues that this ingression of infinity, and hence indeterminacy, into computation establishes the basis for a computational "aesthetics" that is divorced from the empirical or sensible. In this sense her book might be seen as one answer to the call with which Derrida opens his essay "Typewriter Ribbon:" to think the incompatible concepts *event* and *machine* together, a feat that would produce "a new logic, an unheard-of conceptual form" through "a thinking [that] could belong only to the future—and even … makes the future possible." Derrida,"Typewriter Ribbon: Limited Ink (2) ("within such limits")," trans. Peggy Kamuf, in *Material Events: Paul de Man and the Afterlife of Theory,* ed. Tom Cohen, et al. (Minneapolis: University of Minnesota Press, 2001): 277-278.

Fazi, the contingency inherent to computation also serves as a refutation of metacomputation or *mathesis universalis*, the rationalist dream of "a comprehensive science of calculation through which one would be able to grasp the fundamentals of reality itself."[89]

This highlights a crucial difference between her analysis, Derrida's, and mine. Whereas Fazi and Derrida draw metaphysical critiques from an analysis of a formal representational system (computation or writing), I am interested in developing a phenomenological account of the process of operating on and within this system itself. Crucially, the Derrida of "Signature, Event, Context" *rejects* any analysis of the intentions of the writer or speaker. *Contingent Computation*, similarly, is not interested in the "scene" of programming, but the experience or aesthetics of the program itself. My comparison of programming's bugs to Derridean "failure" is, therefore, a deliberate shifting or displacement of his argument that will, I hope, nevertheless prove useful; as we will see, the formal analogy I would like to pursue is not between the program and the written page, but, in a spirit similar to Joque's, between programming and deconstruction itself.

For Derrida, "to write is to produce a mark that will constitute a sort of machine which is productive in turn."[90] Although this description is meant to highlight the radical rupture that the formal system of writing introduces between the author's intention and its received meaning, it obscures the crucial third position (in addition to 1. the writer and 2. the writing) of the reader. It is only through a final, human *interpreter* through whom Derrida's meaning-producing assemblage becomes complete. In programming, conversely, sign and interpretation are combined in machinic execution. Although most code ultimately meets a reader-like user, this user performs no real meaning-creating function, as the response to their input is already pre-programmed. But, from the position of the programmer, the mark he has produced often produces unintended behavior. In this sense, Derrida's formulation might be even better applied

[89] Fazi, *Contingent,* 87.
[90] Derrida, "Signature," 8.

to code than normal writing. As Mark Fisher writes in *Flatline Constructs,* programs are "radically indifferent to any intention that is not already inscribed into them."[91]

Because human intentions are usually confused to begin with, and often become further muddled upon inscription, all programs are riddled by errors, or bugs. The practical persistence of bugs is not the same as the structural insistence of Turing's incomputability, although, as Fazi notes, incomputability implies the impossibility of algorithmically determining the bug-free-ness of a program. Because testing software can only check a program by executing it in finite increments, line-by-line, it is impossible for a testing program to verify the absence of infinite loops in another program without falling into such a loop itself, and thus failing to terminate.[92] In practice, however, a software engineer may very well test his program until, having squashed all of his bugs, he is reasonably satisfied of its correctness, or at least its viability as a product. Incomputability, while a crucial concept for computational theory and research, is not of practical concern to programmers. Bugs may be instances of non-arrival, of miscommunication, but they are not *structurally* interior to the program, because they can be removed.

But although any single bug can be rooted out, it is practically impossible to catch them all. Even after a software has passed through multiple stages of testing and is made available to the public, the cropping up of unexpected and undesirable behaviors creates the need for constant updates. In the iPhone App Store, these are the "bug fixes and performance improvements" that necessitate version 87, version 87.1, version 87.1.1. In these cases, the large-scale deployment of an app to millions of users, all of them doing different things, on different operating systems, on different hardware, combine to an un-testably high (though finite) number of configurations of usage situations—some number of which will, without fail, give rise to undesirable program behavior. In the cases Joque examines, this behavior is the result of intentionally malicious usage of the program. Although a significant aspect of programming is learning to anticipate, and specify the behavior for, such **edge cases**, the empirical variability of real-world applications makes total prediction and control over such situations impossible, and debugging a Sisyphean task.

---

[91] Mark Fisher, *Flatline Constructs: Gothic Materialism and Cybernetic Theory-Fiction* (Brooklyn: Exmilitary Press, 2018), 109.
[92] Fazi, *Contingent,* 122.

But even when working on a small program, in which verification of its correctness is practically possible because of its restricted scale, the process of programming is hampered by the ease with which errors are introduced, not only on the level of syntax, but on the structural or logical level of the program's execution. Even in programming languages in which one has a high degree of competency, syntactic rules are easily broken, whether through typos or forgetfulness. This is especially true when switching between languages: where in Python one can end a line with a return key, Java requires a semi-colon. But the impossibility of reaching absolute fluency in programming exceeds the difficulty of remembering the more-or-less fussy grammatical structures of different languages; more significantly, bugs persist because of the difficulty of grasping the complex choreographies of input/output transformations the computer performs during execution, and the complete foreignness of the computer's "mind." As Joseph Weizenbaum writes, it is impossible for the programmer to "know the path of decision-making within his own program, let alone what intermediate or final results it will produce."[93] This is due to the complex multiplicity of technical processes triggered by hitting *run* on a Java program, especially the lower-level chains of interpretation and transcription mechanisms that are usually invisible at a high level. Many bugs are structural flaws in thinking: forgetting, for instance, that one cannot iterate over a list while simultaneously adding elements to it, or accidentally writing a recursive structure that never terminates. These errors are caused by the difficulty of fully immersing oneself in a technical space governed by what is less a "logic" than a baroque, never-fully-knowable system of meaning. Bugs are annoying reminders of the trickiness of formal operations.

As we have seen, the process of diagramming is enabled by an array of prostheses to the alphanumeric character-writing of programming itself. Such software products are ultimately prophylactics against bugs, those (supposedly) external incursions of various kinds of materiality into the (supposedly) immaterial, logocentric writing of programs. Etymologically, these bugs are truly entomological: The term was popularized in computer science when US Navy officer Grace Hopper found a moth between the relays of a faulty Harvard Mark II computer she was

---

[93] Joseph Weizenbaum, *Computer Power and Human Reason: From Judgment to Calculation* (New York: W.H. Freeman, 1976), 234.

working with.[94] Just as insects are eradicated from human habitation, the computer bug is, figuratively and literally, the lacuna in the programmer's knowledge and control of the program, a failure in the transmission of the programmer's meaning, an accident in syntax or logic to be rooted out and squashed. The simplest kinds of bugs—syntax errors, or misnaming of variables—are a function of "languages" whose ultimate technical, material implementation require a rigidity of representation that is often difficult for programmers to adhere to. The bugs caused in software by the edge cases of widespread usage might be called "material" in a looser sense: their unexpectedness arises from the embedding of programs in the real world—introduction onto the market, into the hands of different consumers, onto different systems, and out of the sterile test-space of the computer lab. The last kind of bug is created by the multi- and other-dimensionality of program-diagrams themselves: by the trickiness of tracking, or scaling (in the sense in which one scales a wall or mountain), the relationship between the invisible inscriptions that occur *in* the machine, under the hood, and the unnatural contortions of time, space, and "meaning" that are suspended in a more abstract space of algorithms.

In this chapter, we have examined aspects of programming that, in different ways, materialize aspects of writing or mediation in general that are usually marginalized in logocentrism: the software supplements that enable high-level coding (itself a software in relation to low-level code); the variability of meaning; and the persistence of errors produced by a lack of complete mastery of the technical medium—the thought that is not entirely present to itself and therefore gives rise to unexpected sign behavior. Although the program itself differs from the printed page, most notably in its deterministic "meaning" or behavior, I would argue that the operations of programming as described in this chapter resemble Derrida's own writing—that is, the "strategic device" of deconstruction itself.[95] Derrida objected to any description of deconstruction that would reduce it to a recipe or program, explicitly warning against any

---

[94] This was 1945, when the very-large computers emitted enough heat to draw good-size flying insects into their interiors, which would regularly cause shortened circuits and thus machinic malfunctions. Boris Veldhuijsen van Zanten, "The very first recorded computer bug," *TNW*, Sep 18, 2013. https://thenextweb.com/shareables/2013/09/18/the-very-first-computer-bug/.

[95] Jacques Derrida, "The Time of a Thesis: Punctuations," trans. Kathleen McLaughlin, in *Philosophy in France Today,* ed. Alan Montefiore (Cambridge, UK: Cambridge University Press, 1983), 40, **quoted in** Joque, *Deconstruction*, 80.

"technical or procedural significations" that might insinuate themselves, liable to "seduce or lead astray" the unwary reader.[96] But I do not compare deconstruction to the program, but to programming. Both are kinds of writing that involve a thinking not only *through* writing (as if it were a frictionless "vehicle" of "telecommunication"), but *of* and *with* writing.[97] The similarities are visible in the formal and/or programmatic methods by which Derrida disrupts his prose, allowing the syntactic structures and technologies of inscription at work to materialize. More generally, both deconstruction and programming are a kind of writing-made-difficult, testaments to the fact that meaning neither *wholly* precedes language nor can pass into it without suffering various transformations and surprises. In this sense, writing and programming share formal qualities in that both operate on form itself.

---

[96] Jacques Derrida, "Letter to a Japanese Friend," in *Derrida and Différance*, ed. David Wood and Robert Bernasconi (Evanston: Northwestern University Press, 1988), 3. Joque insightfully summarizes Derrida's seeming indecision over the nature of the proximity between deconstruction and technicality; see Joque, *Deconstruction,* 80-81.
[97] Derrida, "Signature," 3-4.

CHAPTER 3. THE USER: TECHNOLOGIES OF SELF

The mode of "thinking within mediation" described in the previous chapter is an experience of encounter. This chapter begins from the technical event of runtime, describing the mode of subjectivity engendered by interminable, "addictive" conversation with the machine. Programming is a double activity of reading and writing that is simultaneously a relation to oneself and a relation to an other—a cybernetic instantiation of Deleuze's fold of subjectivation. To delineate the features of this cyberfold, we examine Foucault's technologies of the self. In contrast to the discursive practices of self-examination of classical Antiquity, programming leads one *out* of oneself, and into a cybernetic system.

Runtime and recursion; the *pharmakon*

We have discussed meaning in terms of the values different noun-like variables *hold*—in effect, in terms of their "content." But the program, ultimately, is writing that *does*. It is for this reason that it is often characterized as the consummate Austinian performative utterance.[98] For the programmer, this performativity is immediately sensible: pressing "run" prints a result to the terminal window, refreshes the color of a web-page, or updates a database. But this executability alone does not explain why, phenomenologically, running is such a salient feature of

---

[98] Hayles and Galloway agree that code is performative "in a much *stronger* sense than that attributed to language" (Hayles, *Mother,* 50, emphasis mine), and that "code is the only language that is *executable*" (Galloway, *Interface*, 70). The changes that occur upon the pronouncement of an "I do" in a wedding ceremony are "weaker" because they happen primarily in the human mind, and only effect external changes "through complex chains of mediation." (Hayles, *Mother,* 50). I would argue that this concept is part of a larger trend in software theory in which the digital is presented as the *literal* instantiation of what in writing is merely *metaphorical* (see: Espen Aarseth, *Cybertext,* Baltimore: Johns Hopkins, 1997*,* 2)—a stance that elides the technicity of language itself. Chun argues for the *strong* performativity of "code-as-law" by drawing on Derrida's "Force of Law: The Mystical Foundation of Authority" to emphasize code's collapsing of the democratic separation of executive, judiciary, and legislative into the ultimate police power (Chun, *Updating to Remain the Same*, Cambridge, MA: MIT Press, 2016, 82-85). But this comparison actually undermines any particular strongness of code performativity: If programming is a feeling of power, surely this is no departure from that felt by the policeman who shouts "Hey, you!" or the priest who sanctions an "I do."

programming. The significance of performativity to the *process* of programming itself arises, again, through bugs and edge cases; what is deemed the culmination or *telos* of the program as a technology only rises to level of experiential relevance through failures and other marginal events. Without debugging or testing, "run" would be a rare event—merely the final step of an otherwise static programming process. But, because of bugs, programmers run their code over and over again, changing their text file with each iteration in response to the feedback returned by each run.

In debugging, the programmer thus enters into a recursive feedback loop with her machine. Derrida is not the only one to characterize this relationship as addiction. As one tech blog (the first of many results when googling "programming addiction") writes:

> You might imagine that a person would not want to spend hours on end staring at a computer screen, skipping meals, losing track of time, only using a text editor, making small changes to a text file, observing small results, over and over again—in other words, the experience of computer programming. Those who enjoy, or, dare I say, are addicted to computer programming—they spend their time in a trance, going through the motions, waiting for the moment when they have solved a problem and their code does what it was intended to do. … A degree of nervous anticipation builds up before each verification and when the puzzle is finally solved, there is a mild or often intense feeling of pleasure. Which then shortly subsides as the programmer then repeats the cycle, onto the next puzzle, onto the next fix.[99]

Writing in the mid-1970's, Joseph Weizenbaum, a pioneer in artificial intelligence research who would later become one of the field's earliest and most vocal critics, provides a strikingly similar description of what he calls "hackers":

> Wherever computer centers have become established, … bright young men of disheveled appearance, often with sunken glowing eyes, can be seen sitting at computer consoles, their arms tensed and waiting to fire their fingers, already poised to strike, at the buttons and keys on which their attention seems to be as riveted as a gambler's on the rolling dice. When not so transfixed, they often sit at tables strewn with computer printouts over which they pore like possessed students of a cabalistic text. They work until they nearly drop, twenty, thirty hours at a time. Their food, if they arrange it, is brought to them: coffee, Cokes, sandwiches. If possible, they sleep on cots near the computer. But only for a few hours—then back to the console or the printouts.[100]

---

[99] "The Effects of Computer Programming on the Brain," *Grindd*, July 5, 2013, accessed March 31, 2020, http://www.grindd.com/blog/2013/07/the-effects-of-computer-programming-on-the-brain/.
[100] Weizenbaum, *Computer,* 116.

In these accounts, programming is more than merely a labor or a discursive practice, but a way of life—a behavioral pattern that inscribes itself not only on the minds, but on the bodies, of programmers. The iterative nature of programming entwines the human in a technical apparatus driven by a (vicious) feedback cycle. Programmer-code form an information-processing system: Our analysis is therefore *cybernetic* in Norbert Wiener's sense of the term—the study of feedback in self-regulatory systems, of control and communication in animal and machine.[101] This system is a species of the machine-human hybrid Donna Haraway calls the "cyborg," a figure for a condition of subjectivity she, writing in 1985, extended to all living in the late twentieth century.[102]

The descriptions above emphasize that this relationship to the machine induces a neurological and physical pathology in the programmer that implies a degeneration from human to android. This is the programmer as gambling or drug addict, the kind of postmodern subject, "jacked into late capitalism's network of cybernetic communications," that Mark Fisher, in his 1999 dissertation, traces through what he calls the "gothic materialist" media and theory of the late twentieth century.[103] It is reminiscent of David Cronenberg's *Videodrome*, in which a male television producer's "body literally opens up—his stomach develops a massive, vaginal slit—to accommodate a new videocassette 'programme.' Image addiction and image virus reduce the subject to the status of videotape player/recorder; the human body mutates to become part of the massive system of reproductive technology."[104] This is a prosthetized subject, emptied of both interiority and organs, hooked up to the material and economic flows of global capitalism, castrated and penetrated, traversed by the media and technology it consumes intravenously.

---

[101] See Norbert Wiener, *Cybernetics, or Control and Communication in the Animal and the Machine* (Cambridge: MIT Press, 1948).
[102] See Donna Haraway, "A Cyborg Manifesto: Science, Technology, and Socialist-Feminism in the Late Twentieth Century," *Manifestly Haraway* (Minneapolis: University of Minnesota Press, 2016).
[103] Fisher draws on "cybernetic theory-fictions" including J.G. Ballard's *Crash,* Jean Baudrillard's "Ecstasy of Communication," Philip K. Dick's *Bladerunner,* and Jean-François Lyotard's *The Postmodern Condition.* See Fisher, *Flatline.*
[104] Steve Bukatman, "Who Programs You: The Science Fiction of the Spectacle?," in Annette Kuhn, ed., *Alien Zone: Cultural Theory and Contemporary Science Fiction Cinema,* London: Verso, 1990, 206, **cited in** Fisher, *Flatline,* 18.

Weizenbaum distinguishes this condition of "hacking" from the methodical "work" of the engineer: The satisfaction of the latter "comes from solving a substantive problem, not from having bent a computer to his will."[105] Hackers, unlike engineers, "play" programming like a gambler, in what is a struggle for *power*, rather than an exercise of *knowledge*. Chun draws on Weizenbaum's distinction to establish her own division between (systems) programmer and the mere *user* of modern languages, who fetishizes his high-level source code by ascribing a false causal power to it (when the real causality lies in machine code).[106] As discussed in Chapter 1, for Chun, this fetishization of source code constitutes its ideological nature, and the degeneration of programmers into users is, therefore, highly concerning. The high-level language programmer, like the user of graphic software or player of a video game, is duped by an illusionistic interface that obscures reality. In fact, as Fisher points out, such uneasiness over *non*-knowledge (of bugs, of the machine itself) embedded in programming is already present in the earliest work on cybernetics. Wiener's 1964 *God & Golem, Inc,* published decades before the advent of the high-level languages Chun condemns, casts computer code as dangerous sorcery accomplished via "black spells."[107] Fisher summarizes:

> What, according to Wiener, magic spells have in common with code is that the power any user accrues by running them depends upon their giving up 'control' to sequenced programs which may have a very different effect than the user imagines, or anticipates… Sorcery is 'two-edged' because, like cybernetic machines, it awards power—or control—only to the degree that it demands control be given up by the individual subject; the circuit, the cybernetic loop, takes over.[108]

For Weizenbaum, Chun, and Wiener, the unknowability of the written code makes programming about a struggle for power with the machine, in which seeking to blindly dominate or "hack" code also means ceding control to its structuring of the programmer's subjectivity into a cybernetic input/output feed.

---

[105] Weizenbaum, *Computer,* 117.
[106] Chun, *Programmed,* 48-53. See also: "On 'Sourcery,' or Code as Fetish," *Configurations* (vol. 18:3): 299-324.
[107] Norbert Wiener, *God & Golem, Inc.* (Cambridge, MA: MIT Press, 1964), 65. **cited in** Fisher, *Flatline,* 109.
[108] Fisher, *Flatline,* 107-108.

These writers explicitly cast the problem of programming in terms of knowledge and power—which accords with the Foucauldian concept of subjectivity we will pursue in this chapter. But I disagree with their characterization of these two forces in relation to programming. For them, non-knowledge of the machine makes pressing "run" a throw of the dice.[109] But whereas the results of a gambler's game are unknowable because they are determined by pure chance, the functioning of a program is totally deterministic. Bugs arise because the programmer is frequently unable to fully comprehend this functioning or its effects on certain edge cases, but his task is to grasp the mechanisms of causality to whatever level is adequate to the problem he would like to solve. Failure to correctly trace a causal relation is not equivalent to ignorance of its existence: Programming is a reckoning with this former kind of non-knowledge.

Equally, I see no reason why the computer-coder relationship ought to be dramatized as an agonistic struggle for domination, rather than a reciprocal cooperation, a mutually constituted flow-of-control in a heterogeneous system. This is not to evacuate programming of power relations, or political implications, but to preserve a point that Fisher clarifies in relation to Wiener's cybernetics: "control is distinguished from domination, since it is *immanent to the system*—the machine corrects itself."[110] Programming is neither a master-slave relationship nor, like chess, a war between opposing sides; the concept of domination implies a *linear border* between self and machine that precludes the possibility that programmers can craft a more imaginative geometry of subjectivity: a shape as torsional as the diagrams of computation itself. This incorporation is enabled precisely by the asymptotic movement towards understanding the machine by which programming is animated. What interests me is neither the elevation of the programmer to a God or Father whose words are transposed immediately into action, nor his demotion to slavery to machine (Wiener) or false consciousness (Chun), but a careful consideration of the ways in which the materiality and opacity that insist in coding can allow the

---

[109] Weizenbaum devotes several pages to this metaphor. Although I disagree with his diagnosis of hacking, I find his observations on the incomprehensibility of programs elsewhere in the book more convincing. In particular, he points to the problem of programming large systems that endure over time: In the absence of the project's original engineers, each of which only contributed to a small portion of the program, one is left with a nearly incomprehensible system (*Computer*, 232-235). Here we see the differences in analysis that would be necessitated by a broadening of the scale at which we are considering programming and its applications.

[110] Fisher, *Flatline,* 22. Emphasis mine.

programmer to intentionally and intelligently—rather than as an "out-of-control addict"—incorporate an external discourse into their own.

My contention with Chun's argument is not that there is no difference between computer programming and the casual use of software like Word. It is true that digital technologies are, for many, a kind of mystified (phonocentric) "speech-to-text" software. Derrida illustrates this via a description of his own perplexity in the face of word-processing software (again, with an invocation of magic or the occult): "I know how to make it work (more or less) but I don't know *how* it works. So I don't know, I know less than ever 'who it is' who goes there…[people] rarely know, intuitively and without thinking—at any rate, *I* don't know—*how* the internal demon of the apparatus operates."[111] Nor do I deny that the history of computing has introduced more and more layers of mediation between programmer and hardware. My issue is with Chun's implicit eliding of hardware with technicity itself, and her related claim that such mediations induce passivity, stupidity, or a de-skilling of programming. (The bad work of "hacking" is possible no matter the medium, that is to say, even in a systems language.) Although my arguments are illustrated via case studies in C or Python—languages that will, no doubt, eventually become obsolete[112]—my argument has been that programming, independent of any specific language, is about encountering the fact, and thinking within the limits, of mediation itself. The programmer, in and through their task, tries to make out *who it is who goes there,* in their machine—in Simondon's terms, technical knowledge, rather than mystified work. It is an operation in which the limits of one's knowledge, the difficulty of language, materialize in the very attempt to "execute" the written.

This linking of non-knowledge with loss of bodily autonomy and invasion by or submission to a machinic Other ultimately expresses an anxiety over mediation, technology, and writing itself. The programmer-as-addict recalls a discourse as old as the *Phaedrus,* in which Socrates compares writing to a drug. This *pharmakon,* as Derrida writes, is a "philter, which acts as both remedy and poison," a "charm" or "power of fascination" that "introduces itself into the

---

[111] Derrida, "Word," 23.
[112] See, for example, former Apple UI designer Bret Victor's Dynamicland: a research project dedicated to removing computer programming entirely from language, and into the physical world. "Dynamicland," *Dynamicland,* accessed March 31, 2020, https://dynamicland.org/.

body of [Plato's] discourse with all its ambivalence."[113] Chun's condemnation of the gradual demotion of programmers into users echoes Plato's censure of the Sophists. Writers are already (drug) users.[114] Algorithms are often compared to recipes; here, they are a recipe of the same kind as writing. Any manipulation of forms and representations is therefore a kind of witchcraft or potion-making, and the occult operations of the *pharmakon*, which functions "like a cosmetic concealing the dead under the appearance of the living," is, for Plato, a body-horror story as grotesque as Cronenberg's *Videodrome*: Unlike the living organism of *logos*, of speech, writing is an undead perversion that calls into question the authority of the Father and natural reproduction.[115]

I reference Derrida's reading of Plato partially in order to divorce the problem of programming from postmodern anxieties over digitality or the gothic pallor of cybernetic aesthetics, and to situate it in relation to the more general question of technicity and writing. Moreover, Plato's *pharmakon*—of Egyptian origin—indexes a certain foreignness or unknowability that we find also in the opacity of the machine's operation. This unknowability, as we will see, leads to a distancing from self-identity via the denaturalization of one's thoughts and representations. "The *pharmakon* makes one stray from one's general, natural, habitual paths and laws."[116] The unknowability of code, in this way, allows programming to surpass the merely teleological. Hacking (as opposed to engineering) is, in Chun's words, "a technique, a game without a goal and thus without an end" that allows programming to be "not simply the production of a commercial (or contained) product" but rather, like a drug, "something endless that always leads us pleasurably, as well as anxiously, astray,"[117]

Reading and surprise: Bugs as symptoms

The *pharmakon* leads us back to the question of writing and reading, and to examine more closely the particular constellation of relations at play in the "use" of programming. As we have

---

[113] Jacques Derrida, *Dissemination*, trans. Barbara Johnson (Chicago: University of Chicago, 1983), 70.
[114] Chun, *Programmed,* 46.
[115] Derrida, *Dissemination,* 142; 79-80.
[116] Ibid.*,* 70.
[117] Chun, *Programmed,* 49. Chun interprets this "surplus pleasure" of code through the Marxist framework of commodity fetishism.

seen, although programming has an end-goal, the nature of problem-solving cannot be collapsed into the linear execution of a command. The persistence of bugs gives rise to a recursive writing process: Programming's workflow is structured by a cyclical relationship of writing-and-running that gradually (sometimes, asymptotically) reaches a correct result via the programmer's incorporation of feedback from her machine. This feedback—whether in the form of error messages, values printed to a terminal window, or the running of a simulation—is the result of the machine's reading of the programmer's text. The programmer must then interpret the product of this reading, and assess whether it corresponds to her desired outcome. Since it almost always does not, she must then return to her own text, re-reading her code in order to identify the responsible syntactic or logical flaw, before correcting this error and re-running the program. Programming is therefore a reading just as much as it is a writing, on the part of both machine and human.

This is a point on which Joseph Weizenbaum writes particularly lucidly: It is impossible for the programmer to completely "know the path of decision-making within his own program, let alone what intermediate or final results it will produce."[118] This is due to the multiplicity of complex technical processes triggered by hitting *run* on a high-level program, especially the lower-level chains of interpretation and transcription mechanisms that are usually invisible at a high level. But good programming is not produced by a *total* understanding of the problem one desires to solve, or the solution one has planned to implement: "Understanding something always means understanding it at a certain level," rather than "to its ultimate depth."[119] This understanding, moreover, is reached "experimentally" through programming itself, rather than prior to it.

> Programming is rather a test of understanding. In this respect it is like writing; often when we think we understand something and attempt to write about it, our very act of composition reveals our lack of understanding even to ourselves. Our pen writes the word 'because' and suddenly stops. We thought we understood the 'why' of something, but discover that we don't. We begin a sentence with 'obviously,' and then we see that what we meant to write is not obvious at all […] Programming is like that. It is, after all,

[118] Weizenbaum, *Computer*, 234. Weizenbaum's own insistence on this point of the ultimate unknowability of code undoes the distinction he makes elsewhere in the book between "hacker" and "engineer."
[119] Ibid., 107-108.

> writing too. But in ordinary writing we sometimes obscure our lack of understanding, our failures in logic, by unwittingly appealing to the immense flexibility of natural language and to its inherent ambiguity.[120]

Here, Weizenbaum provides a description of writing that, in simple language, accomplishes a great deal of the comparison we began in the previous chapter. The manner in which programming, unlike normal writing, forces us more explicitly to confront our own thinking is through the "reading," or interpretation, that must take place on the part of the machine. "A computer," Weizenbaum continues, "is a merciless critic."[121] Derrida might add that this occurs in his writing as well (albeit perhaps in a less literal manner): "as I write, it is I who am being read first of all by what I claim to write."[122] We remain agnostic to the question of artificial intelligence; without anthropomorphizing the computer as "criticizing" or "thinking" at all, however, one might characterize programming as a type of writing that must be made to encounter an information-processing entity that is radically "other" than human. Besides the fairly trivial fact of the rigid syntax of programming languages, Weizenbaum makes clear, programming is difficult because the computer knows nothing of the programmer's reality.[123] The programmer must mold (or "model") her problem into a figure recognizable by a machine that is like an alien from elsewhere. If the inevitability of misunderstanding plagues inter-human communication, it is exacerbated in programming.[124]

Once she has crafted an adequate representation for her problem, it must be continually revised. Perhaps she took a wrong turn when calculating the path of causality or flow-of-control she has laid out, or perhaps she has failed to specify certain aspects of her model. The programmer submits her thinking to rigorous self-examination. As Weizenbaum says, "one of the most cogent reasons for using computers is to expose holes in our thinking."[125] To use Althusser's words, each bug is a "symptom" of a hidden assumption on the part of the

---

[120] Weizenbaum, *Computer,* 108.
[121] Moreover, "the computer's criticism is very sharp and cannot be ignored; a program doesn't work at all, or delivers obviously wrong results." Weizenbaum, *Computer,* 108-109.
[122] Jacques Derrida, *Points…Interviews, 1974-1994*, trans. Peggy Kamuf, ed. Elisabeth Weber (Stanford: Stanford University Press, 1995), 66.
[123] Ibid., 109.
[124] Note that this is a different configuration of the problems of misunderstanding and context Derrida grapples with in "Signature, Event, Context."
[125] Weizenbaum, *Computer,* 65.

programmer, an unwitting "play on words" that causes the terrain of the program to shift under the unwitting programmer's feet, opening up holes.[126] If, once compiled and run, each program is a crystallization of the programmer's thought in a technical terrain, then debugging is a series of minute seismic shifts through reading. This reading is always a double reading: The program is never self-identical. It exists on one level as that which the programmer *wishes* to do, as the functionality that he *thinks* he has inscribed, and on another level as the machine's technical interpretation. The divergences in this hidden double meaning are often exposed through the event of clicking *run,* after which, in debugging, the programmer must read one text *with* the other. As Ellen Rooney emphasizes, any symptomatic reading addresses not only "'the text itself' but always also the other reading or what Althusser calls the problematic."[127] Debugging is a reckoning with the other reading that is machine interpretation, and hence a "play" with the other reader who is the machine. Rooney's description of the Althusserian mode of critique as constantly generating *surprise* is the aspect of symptomatic reading that perhaps most resonates with programming. In "Symptomatic Reading is a Problem of Form," citing Barbara Johnson, she writes:

> "The surprise of otherness is that moment when a new form of ignorance is suddenly activated as an imperative." Johnson urges us toward the "surprise encounter with otherness" that will "lay bare some hint of an ignorance one never knew one had," that is, pose a question that reorganizes reading on another terrain and changes "the very nature of what [we] think [we] know." No reader commands this process; she is its symptom, herself a surprising "reading effect" in an encounter with the other reader, one who cannot stay on script.[128]

This passage shows the relationship between the reading of an other and one's own self-transformation that is at work in debugging. Here, the surprise of a bug is similarly the exposure of the programmer's ignorance, serving as an impulse to rethink, re-read, and revise one's text. Although the machine may have its own script, the dialog established with the programmer is

---

[126] Althusser, *Reading,* 24.
[127] Barbara Johnson, "Nothing Fails like Success," in *A World of Difference* (Baltimore: Johns Hopkins University Press, 1987), 11; 15, **quoted in** Ellen Rooney, "Symptomatic Reading is a Problem of Form," in *Critique and Postcritique,* ed. Elizabeth Anker and Rita Felski (Durham: Duke University Press, 2017), 131.
[128] Ibid., 134.

never predictable. We will now track the effects the "other reading" has on the programmer's self by following the Althusserian topology, again, through Deleuze, Foucault, and the diagram.[129]

The Superfold

Towards the end of *Foucault*, Deleuze describes a particular kind of diagram he calls the "fold." The fold is the manner through which subjectivation is carried out: "the inside as an operation of the outside," in which "the relations of the outside [are] folded back to create a doubling, [allowing] a relation to oneself to emerge, and constitute an inside which is hollowed out and develops its own unique dimension."[130] For Deleuze, "Foucault's fundamental idea is that of *a dimension of subjectivity derived from power and knowledge without being dependent on them.*"[131] This figure for a non-innocent subjectivity inextricable from the operations of power, neither self-identical, pre-technological, nor whole, looks like our character of the cyborg or pharmakon-imbibing programmer. Deleuze makes clear that folding, once again, is the work of creating a new kind of shape that is unimaginable from the perspective of Euclidean space, in which:

> every inside-space is topologically in contact with the outside-space… this carnal or vital topology, far from showing up in space, frees a sense of time that fits the past into the inside, brings about the future in the outside, and brings the two in confrontation at the limit of the living present.[132]

Here, time and space, as with the programmer's chronopoetic diagrams or Luciana Parisi's algorithmic topologies, are co-articulated into a non-linear, non-Cartesian figure. One must note that, in the context of Foucault's archaeology, the "past" Deleuze references seems best understood as a macroscopic dimension, referring to the manner in which a subject enfolds the historically specific diagram in which he moves. The "future" is a time-place in which one becomes Other than what one is.[133] Our analysis of programming in Chapter 1 understood "time" as a material resource given a sort of figural plasticity and subjected to molding by the

---

[129] And we will open up the divergences between programming and symptomatic reading in the conclusion.
[130] Deleuze, *Foucault,* 97; 100.
[131] Ibid., 101.
[132] Ibid., 119.
[133] Deleuze, "Dispositif," 345-46.

programmer. Our leveling of these two time-scales allows us to create an isomorphism between them, but, later on, we will ask after what is lost in such a substitution of the technical for the human-historical.

The particular features of this fold diagram must be continually made anew. In the conclusion to *Foucault*, Deleuze wonders about the form of subjectivity of our time: it might be "something like the Superfold, as borne out by the foldings proper to the chain of the genetic code, and the potential of silicon in third-generation machines."[134] We note Deleuze's reference to cybernetics, and, without hewing to the particulars of Deleuze's analysis of Nietzsche's superman, will take this provocation of the superfold as a jumping-off point to imagine programming as one method of constructing such a twenty-first century figure of subjectivity. For Deleuze, folding is also a figure for what it means to "think"; through this analysis, therefore, we will also once more arrive at a comparison between programming and the writing and reading of the critic.[135]

Technologies of the Self

Foucault's own concept for subjectivation is the "technology of the self," a type of "technique" with which he became increasingly preoccupied in his later work. As he explained at a lecture at Dartmouth College in 1980, while "analyzing the experience of sexuality, I became more and more aware that there is in all societies" another type of technique, other than the techniques of production, signification, and domination, that "permit individuals to effect, by their own means, a certain number of operations on their own bodies, on their own souls, on their own thoughts, on their own conduct, and this in a manner so as to transform themselves, modify themselves, or to attain a certain state of perfection, of happiness, of purity, of supernatural power, and so on."[136] He would primarily explore this idea through a study of practices of "care of the self" (*epimeleia heautou*) in late Antiquity and early Christianity. This definition confronts us with several

---

[134] Deleuze, *Foucault*, 131.
[135] Ibid., 118.
[136] Michel Foucault, *About the Beginning of the Hermeneutics of the Self: Lectures at Dartmouth College, 1980* (Chicago: University of Chicago, 2015), 25.

differences between programming and the practices Foucault examines that, for us, structure rather than preclude their comparison.

First: Foucault's techniques are more or less explicitly constructed as methods of self-transformation by their practitioners. If programming involves such a process of subjectivation, it is only as a by-product or surplus effect of its stated goal (of building a software tool, regulating communication of distributed systems, etc). But here we might recall that our account of programming is, in a sense, that of a deviant pleasure: of "a technique, a game without a goal and thus without an end" that exceeds "the production of a commercial (or contained) product."[137] So while programming may not be discursively constructed as means of self-transformation, the qualities we have emphasized here, including a certain aesthetic craft of diagramming and, more importantly, "dialogue" with the machine, establish that the ends of programming exceed either the execution of a command or the realization of a goal.

Moreover, it is initially unclear to what extent the programmer operates on their "self" at all. In another lecture at Dartmouth several weeks later, Foucault explains that by self he means "the kind of relation that the human being as a subject can have and entertain with him*self*. *For instance*, the human being can be, in the city, a political subject. Political subject means he can vote, or he can be exploited by others, and so on. The self would be the kind of relation that this human being as subject in a political relation has to him*self*."[138] This is a recursive statement: The world "self" is used in its definition. But this definition works well enough here; like Foucault, we are not interested in un-boxing these nested pairs of self-subject that form each "self" in order to arrive at the deepest inside. (Programming, also, employs recursive function definitions to great effect.) Understanding programming as a technology of the self means considering the text of the program to be a kind of externalization of the programmer's thoughts, of himself, that constitutes a subject-like dimension on which the programmer then operates. Like all technologies of the self, programming passes through an intermediate in the form of its inscription in a secondary, material-discursive technology. But, unlike the Greek forms of *epimeleia heautou*, programming's "subject" is the human as suspended in a semiotic-

[137] Chun, *Programmed,* 49.
[138] Foucault, *About,* 116. Emphasis mine.

technological medium, rather than in a politics or society. Indeed, Foucault does not explicitly constrain subjecthood to the political arena. The citizen is only one example ("for instance"); the political is like the content of an example from which we extract the formal structure. We will return to the stakes of performing such a substitution of content—of the semiotic-technical for a political or ethical dimension—later on.

In emphasizing subjectivity as a discursive dimension of the self, we actually bring the superfold closer to Foucault's own work. In Deleuze's reading of Foucault, which is in many ways an abstraction from Foucault's own historical method, it is sometimes easy to lose sight of this fact: Across Foucault's work, diagrams are figured, at least in part, through discourse; this is true of the disciplinary techniques of *Discipline and Punish,* the incitements to confession of the first volume of *History of Sexuality,* as well as the practices of virtue privileged in his later work on the Greeks. *Parrhēsia,* for example, is a mode of truth-telling that Foucault traces through several different authors and epochs. Although initially a political act—the discursive genre through which a subordinate might address his superior, a form for "speaking truth to power"—in Socrates's teachings *parrhēsia* is generalized to a holistic "care of the self connected to the relation to the gods, the relation to truth, and the relation to others."[139] In either case, *parrhēsia* is a discursive act that retroactively induces a self-transformation: It is "a way of binding oneself to oneself in the statement of truth, of freely binding oneself to oneself, and in the form of a courageous act" in which "the event of the utterance affects the subject's mode of being."[140] In the second and third volumes of *History of Sexuality,* Foucault studies *enkrateia*, "an active form of self-mastery, which enables one to resist and struggle, and achieve domination in the area of desires and pleasures."[141] This species of care of the self encompassed, in addition to a dietetics or regimen of the body, a system of reading and writing embedded in social relations:

> There are the meditations, the readings, the notes that one takes on books or on the conversations one has heard….There are also the talks that one has with a confidant, with friends, with a guide or director. Add to this the correspondence in which one

---

[139] Michel Foucault, *The Courage of the Truth (The Government of Self and Others II): Lectures at the Collège de France, 1982-84,* trans. Graham Burchell, ed. Fréderic Gros (New York: Picador, 2008), 91.
[140] Michel Foucault, *The Government of Self and Others: Lectures at the Collège de France, 1982-83,* trans. Graham Burchell, ed. Fréderic Gros (New York: Picador, 2008) 66, 68.
[141] Michel Foucault, *The Use of Pleasure* (New York: Pantheon, 1984), 64.

> reveals the state of one's soul, solicits advice, gives advice to anyone who needs it—which for that matter constitutes a beneficial exercise for the giver, who is called the preceptor, because he thereby reactualizes it for himself. Around the care of the self, there developed an entire activity of speaking and writing in which the work of oneself on oneself and communication of others were linked together.[142]

In these related practices of *parrhēsia* and *enkrateia*, we find, therefore, that technologies of self, for Foucault, always involve both reading and writing. Like programming, they involve an *askēsis*, a training or set of practical exercises, embedded in discursive technologies.

SELF-EXAMINATION

We are not interested in establishing an exact correspondence between programming and any one of the historical practices Foucault discusses. But the aspects of programming we have described might be compared to greatest effect to the "art of self-knowledge" specific to late Antiquity. This instantiation of the older Delphic injunction to "know thyself" involved, in addition to testing procedures (of abstinence), a nightly "self-examination" in which one was to measure one's actions against certain rules of conduct. This daily accounting, described in exacting detail in Seneca's *De ira*, was less a judicial process and more like "an act of inspection in which the inspector aims to evaluate a piece of work, an accomplished task."[143] One might say that, in debugging, the programmer measures the performance of the program against the goals that he had set for it. But this method of self-evaluation—in terms of ends met—does not capture the non-teleological thinking we have ascribed to programming. A more interesting comparison might be made to another mode of self-examination, elaborated by Epictetus. This exercise of thinking is, like programming, a "labor of thought with itself as object."[144] It is

> an examination that deals with representations, that aims to 'test' them, to 'distinguish' (*diakrinein*) one from another and thus to prevent one from accepting the 'first arrival.' 'We ought not to accept a mental representation unsubjected to examination, but should say, 'Wait, allow me to see who you are and whence you

[142] Michel Foucault, *The Care of the Self* (New York: Pantheon, 1986), 51.
[143] Ibid., 62.
[144] Ibid.

> came' (just as the night-watch say, 'Show me your tokens'). 'Do you have your token from nature, the one which every representation which is to be accepted must have?'[145]

The task of discrimination or *diakrisis* is a kind of border control: stopping every representation that comes into the mind, either allowing it to pass or rejecting it, and thereby ensuring one's autonomy through the mechanism of rational choice; it is both "a test of power and a guarantee of freedom: a way of always making sure that one will not become attached to that which does not come under our control."[146]

Debugging requires a similar kind of second-guessing or re-reading of representations: going through code line by line, checking whether one truly understands the manner in which the machine will interpret what one has written. But this also shows us the primary difference between the cyborg subjectivity and that of the Greek free man. *Diakrisis* involves a relation of self to self in which one identifies and banishes the illegitimate, foreign thought—that which does not have the appropriate "token from nature" and therefore eludes control. The cyberfold, as we have argued, performs no such policing of borders. Nor does it involve a weighing of coins to affirm their worth, sifting real from artificial currency—another metaphor Epictetus employs. Neither the categories internal/external, nor natural/artificial, are relevant here. Rather, programming is a tracing of one's thought in a strange shape and foreign language, which one then re-writes and re-shapes in a continuous process through which the externalization-of-self that is the program shifts and mutates. This gradual metamorphosis therefore expresses a different movement of self-transformation than Epictetus's *diakrisis,* which merely "accepts" or "rejects" representations (rather than operating on them), and which is ultimately aimed at a preservation of or a return to self-identity.

PHARMACOLOGICAL DIALOGUE

In this sense, although neither is straightforward or linear, the Greek fold and the cybernetic fold are really figures with different contours and trajectories of movement. The Hellenistic "conversion to oneself" is a "circle," "loop," or "falling back": "the subject must advance

---

[145] Epictetus, *Discourses*, French ed. and trans. J. Souilhé, Collection des universités de France, I, 4, 18; III, 16, 15; III, 22, 39 **quoted in** Foucault, *Care,* 63-64.
[146] Foucault, *Care*, 64.

towards something that is himself" in a journey that is simultaneously a return.[147] It is an Odyssey bringing man back to his homeland or origin, the successful weathering of which enables self-mastery or self-ownership: "The soul stands on unassailable grounds, if it has abandoned external things; it is independent in its own fortress; and every weapon that is hurled falls short of its mark."[148] The subjectivation of programming involves a different kind of looping: rather than a return to oneself, an externalization of oneself into an alien form which is further transformed by a series of modifications. The manner in which self/other are related in the Greek practices of *epimeleia heautou* and the cybernetic fold are, in fact, kinds of inverse forms.

The Greek technologies of the self always involve the discourse of an other. Practiced within a social network, they are strengthened by conversations and correspondences with one's friends and peers—peers being, of course, free men of a certain political and social standing. But the most important social bond in the care of the self is the relationship of student to teacher, through whom the skill or *tekhnē* of this *epimeleia heautou* is transmitted. This relation, between "technician" and pupil, was especially important in Plato, where "contemplation of self and care of self are related dialectically through dialogue."[149] Later, in the Hellenistic era, dialogue would be replaced by the two discursive activities of listening (to the teacher), and self-examination through writing. This discourse received from outside is always given and incorporated with the aim of autonomy and self-mastery. Socrates takes care of men not in order to sustain their dependence on him, but "so that they learn to take care of themselves."[150]

The inter-human discursive relations that enable programming are especially pertinent to professional software engineering, in which the code one writes must be legible to human colleagues as much as to the machine. But within the limited scope of this analysis, the discourse of the "other" in programming is not of another human. I must note here that I use that word only imprecisely, in a manner reminiscent of, but more general than, Derrida's Other or Deleuze's

---

[147] Foucault, *The Hermeneutics of the Subject (Lectures at the Collège de France, 1981-82)*, trans. Graham Burchell, ed. Fréderic Gros (New York: Picador, 2001), 248-249.
[148] Seneca, *Letter to Lucilius,* French ed. and trans. F. Préchac and H. Noblot, Collection des universités de France, 5, **cited in** Foucault, *Care,* 65.
[149] Foucault, *Technologies of the Self: Lectures at University of Vermont Oct. 1982* (Amherst: University of Massachusetts Press, 1988)*,* 32-33.
[150] Foucault, *Courage,* 110.

Outside. Our lower-case other is merely that which is not oneself—a vague definition that I intentionally allow to bear the traces of more specific usages (the human foreigner who comes from elsewhere, for example). Later, we will ask after the political and ethical implications of failing to specify this concept more closely. For now, I hope that this definition might, like Foucault's recursive definition of "self," be functional despite its poverty.

The manner in which the specific nature of this other/outside does or does not figure in our argument on "foreign discourse" can be illustrated by an anecdote from computer science history: Joseph Weizenbaum, the computer scientist on whose accounts of programming we have drawn, became famous in the mid-1960's for his work on one of the first natural language processing systems. This early chatbot, ELIZA, was designed to respond to human input in the conversational style of a Rogerian psychotherapist, who interacts with his patient by reflecting their language back to them. The dialogue might go something like:

> USER: I haven't been feeling well.
> ELIZA: Why do you think you haven't been feeling well?
> USER: I've been thinking about my father.
> ELIZA: Tell me more about your father.

And so on. This is a fairly easy program to write; Brown computer science students implement a version of it in their first semester. Weizenbaum was horrified by the program's reception by certain sectors of the public, including a paper published in a psychiatric journal suggesting that such a program might be actually used in therapy, and his secretary's emotional request that he give her some privacy while she was talking to the program, despite knowing very well that it was only a computer. This experience would contribute to Weizenbaum's turn to become an early, vocal critic of artificial intelligence.[151] Weizenbaum was concerned that naïve users might believe ELIZA to be actually thinking and feeling—a problem to which we choose to remain agnostic, instead emphasizing the fact that a user might experience the effects of psychotherapy regardless of what kind of other the "therapist" might be. The formal structure of this dialogue with a foreign interlocutor is what induces self-transformation: It is ultimately a technical exercise, or *askēsis.*

---

[151] Weizenbaum, *Computer,* 2-6.

Programming involves a similar kind of dialogue with the other; the program output returned during debugging is like ELIZA's pre-programmed responses. But this exercise differs from both Greek *epimeleia heautou* and computer-assisted therapy in that it is oriented outwards, rather than inwards. The subjectivation of the superfold is not a return-to-self or self-mastery, but an externalization of one's thoughts. Rather than being guided back to oneself by a foreign interlocutor, this cybernetic dialogue leads one to fall deeper and deeper outside of one's "natural" language and into the foreign language and forms of the machine. The programmer's mind is not a fortress: Becoming good at programming means adapting one's thoughts to the patterns and rhythms of an alien technical-semiotic system, in which, in the form of the program, they are suspended and become an object of external manipulation. In the "self-examination" of one's code to which one is led through this dialogue, what one encounters and operates on is oneself, in altered form. It is therefore simultaneously a folding of the outside into the inside, as well as an externalization.

For the Greeks, care of the self always had a medical dimension, especially during the Hellenistic period.[152] Both medicine and *epimeleia heautou* seek to cure *pathos*, an illness of the body and soul that "takes the form of a movement capable of carrying [the soul] away from itself."[153] Programming, conversely, means intentionally swallowing the drug, medicine, or poison of the *pharmakon*: a discourse from outside that leads one astray, out of one's city, and out of oneself.

OPTIMIZATION AS OIKONOMIA: GOVERNMENT OF SELF AND OTHERS

But there is *another* other, in addition to the technician-teacher and the machine, at work in the technologies of both cyberfold and *epimeleia heautou.* The Greek Odyssey of conversion-to-self was often described as necessitating the navigational skills of the ship's pilot: the *kybernetikos*. This metaphor of piloting was associated not only with care of the self, as well as medicine, but also political governance.[154] *Kybernetikē* would also, later, become "cybernetics." This etymology highlights that both practices follow the model in which government of self is always

[152] Foucault, *Technologies,* 31.
[153] Foucault, *Care,* 54.
[154] Foucault, *Hermeneutics,* 249.

related to the government of others. Before becoming a more generalized practice, self-care and self-mastery were essentially political technologies, meant for use by a political leader: "The Socratic problem is how to teach the virtue and knowledge required to live well or also to govern the city properly."[155] Xenophon's *Oeconomicus* makes clear that "governing oneself, managing one's estate, and participating in the administration of the city" are *isomorphic* techniques; it is for this reason that self-mastery is so highly emphasized. The Greek technology of the self is therefore an *economic* art that takes the same form at multiple scales or scopes of its application.[156]

Programming, similarly, through the dissemination of software products and the construction of digital infrastructures that pattern our lives in increasingly inescapable ways, is a government of others, and a question of economy. Establishing whether, in the case of programming, the relationship between government of self and government of others is also isomorphic, and on what scales, is beyond our scope. It is a question of the manner in which the cyberfold of subjectivation intersects with, is subsumed by, or extends more macroscopic diagrams. One direction such an analysis might take is an investigation of the diagram of optimization. As indicated in Chapter 1, optimization is the logic that governs the distribution and figuration of material resources in the computer; for this reason, it is perhaps one of the most important dimensions of the "other" that the programmer folds into himself—though not the only one. Does one find optimization at other levels, carried there by the ubiquity of digital technologies? One might study discourses and practices around time-space constraints in other industries or mediatic forms that rely on these technologies, or trace optimization to the practices of self-actualization and entrepreneurship through which many technologies of the self today become techniques of neoliberal governmentality. A more truly Foucauldian extension of this chapter would consider more macroscopic dimensions of programming: the discourses of self-actualization, freedom, and disruption with which tech companies communicate internally and externally; the structure of the tech campus as a workplace; the manner in which code circulates

[155] Foucault, *Courage,* 27.
[156] Foucault, *Use,* 74-76.

online; the technical genealogy of different platforms that are built on one another.[157] It would relate these observations to technologies' interactions with users: the discourses of connectivity that saturate social media, the voluntary and involuntary methods of data collection, the blurring of the boundary between consumers and producers of digital media. Through such a series of expansions and contractions of scope and scale, the trick would be to join up these disparate levels with different and intersecting diagrams, preserving their discontinuities while elucidating a broader structure.

Getting free: Going on a trip/voyage, exercise

In this chapter, we have described programming as configuring the self in a reading and writing exercise that, in the same movement, attempts to understand and operate through the logic of a foreign entity. This adaptation of one's thinking to another discourse is coextensive with adapting the machine to one's own ends. This particular technique for folding is, perhaps, truly pharmacological in that it serves as both remedy and poison: leading one out of the strictures of self-identity, and into the flexible pattern of optimization.

In the introduction to *The Use of Pleasure*, Foucault explains his turn away from techniques of power to techniques of the self in studying sexuality: He was motivated by the curiosity that "enables one to get free of oneself." Philosophy today, he goes on, ought "to explore what might be changed, in its own thought, through the practice of a knowledge that is foreign to it. … At least if we assume that philosophy is still what it was in times past, i.e., an 'ascesis,' *askēsis,* an exercise of oneself in the activity of thought."[158] Programming as folding is an *askēsis* of writing and reading—an exercise of oneself that leads to a transformation, in a manner formally similar to the task of writing philosophy itself. Derrida, also, compares his writing process to an athletic activity. The text he reads is an unmasterable horse that he is riding; "the other thing is watching what I do and carries me off at the very moment I try all sorts of mastering manoeuvres."[159] I would like to end here with a passage from an astonishingly

[157] One example of a study of code as product of a specific culture and software industry is Federica Frabetti's *Software Theory: A Cultural and Philosophical Study* (London: Rowman and Littlefield, 2014).
[158] Foucault, *Use,* 8, 9.
[159] Derrida, *Points,* 66.

prescient interview Derrida gave in 1996 on the subject of "word processing," writing on the computer. Derrida's computer user is not a programmer, and the qualities of programming that involve *thinking* or mental exercise are perhaps less apparent here. But this passage plays on many of the themes we have developed in the past three chapters: the flux of forms, the diagram that carries and exceeds us, addiction, the navigation of this torsional movement outwards. It shows that the trajectory by which the computer user becomes alien to herself is also one in which she becomes invisible to herself, seeing herself without seeing herself surfing this cybernetic fold: perhaps freedom, perhaps an abdication of (political) responsibility, perhaps an abdication of subjectivity itself.

> The computer installs a new place: there one is more easily projected toward the exterior … toward the aspect that is thereby wrested away from the presumed intimacy of writing, via a trajectory of making alien. Inversely, because of the plastic fluidity of the forms, their continual flux, and their quasi immateriality, one is also increasingly sheltered in a sort of protective haven. No more outside. Or rather, we see ourselves without seeing ourselves enveloped in the scroll or the sails of this inside/outside, led on by another revolving door of the unconscious, and exposed to another coming of the other. And it can be sensed, differently, for the 'Web,' this WWW or World Wide Web that a network of computers weaves all about us, across the world, but also about us, *in us*. Think about the 'addiction' of those who travel day and night in the WWW. They can no longer do without these world crossings, these voyages by sail [*à la voile*], or veil [*au voile*], crossing or cutting through them in its turn.[160]

[160] Derrida, "Word," 28.

CONCLUSION

Finally, I would like to indicate, in a rather speculative manner, some possible implications of my argument: first by highlighting some of the problems raised by this comparison of programming to critique, and then by considering this position in relation to a philosophy of originary technicity.

The content of critique?

In Chapter 1, I described programming as the creation of figures of time and space, or diagramming, and in Chapter 2, as a writing that constantly materializes the structure of mediation. In Chapter 3, programming is a technology of the self. Each of these descriptions is also a comparison to the "shape" of thinking of certain philosophers or critics: Althusser, Deleuze, Derrida, Foucault. I have insisted that this comparison is one of *form,* as if the writing of these thinkers exhibited a kind of pattern that is isomorphic with the operations of programming. But perhaps I have constructed a formal fallacy, a "play on words" that has unwittingly shifted the terrain, pulling the rug out from under both critique and programming, leaving them suspended and decontextualized, and resulting in an evacuation of the political meaning of either practice. If so, the least I can do is not "spirit away the corpse" of what has been killed in this substitution.[161]

The differences between programming and the work of the 20th century French philosophers consulted here are obvious and might be counted dozens of different ways. The technologies of the self Foucault describes are explicitly aimed at ethical self-transformation; programming is not. Nor is it a parrhesiastic mode of "speaking truth to power"; in fact, unlike scientific or mathematical discourses, programming holds no relation to "truth" at all. Unlike Foucault's archaeology or genealogy, it is not historical. Derrida's figures of speech are deployed

[161] Althusser, *Reading*, 40.

in the service of a critique of a Western metaphysics of presence; programming's are not. In relation to symptomatic reading, Rooney highlights that, "for Althusser, a problematic is the structure of presuppositions that constitutes a discourse, its enabling conditions, *historical and political*; the problematic defines the objects within a field, fixes lines of inquiry, and delimits the form of the solutions thinkable within its limits."[162] We have described the terrain or "problematic" of programming as the semiotic structures constructed by the programmer and/or defined by the machine's field of interpretation. While programming involves a questioning, interpretation, or manipulation of this semiotic terrain, it does *not* necessitate questioning of that larger problematic: the diagram of optimization, the historical, political, and economic mode of production that create the condition for software engineering, and so on.

If programming follows *some* of the right "forms" of reading and writing, then, it is on the wrong topic. One might speculate whether the relevant differences are ones of "content." In this case, the question would concern the possibility of a separation of form and content. To do a symptomatic reading, does the book have to be *Capital*? Does the "subject" in a proper technology of the self have to be political, its goals ethical? Does ethical self-transformation require the "other" to be human? Or can it also be an animal? Or can it also be a machine? My comparison of programming to critique has swept these questions aside. Perhaps the attempt to extract a kind of abstract form or structure of critique has been wholly misguided. In a 1971 lecture on the subject "What is Critique?", Foucault begins by emphasizing the apparent contradictions posed by attempting to answer this question.

> One will be surprised to see that one tries to find a unity in this critique, although by its very nature, by its function…it seems to be condemned to dispersion, dependency, and pure heteronomy. After all, critique only exists in relation to something other than itself.[163]

The "something other than itself" to which critique responds is a *specific* form of governmentality. It addresses the question of "how not to be governed *like that*, by that, in the name of those principles, with such and such an objective in mind and by means of such

---

[162] Rooney, "Symptomatic," 133. Emphasis mine.
[163] Michel Foucault, "What is Critique?" trans. Lysa Hochroth, in *The Politics of Truth*, ed. Sylvère Lotringer and Lysa Hochroth (New York: Semiotexte, 1997), 25.

procedures, not like that, not by them."[164] In this sense, programming as I have described it, in absence of any political content, can not be *like* societal critique, even in a merely "formal" or "isomorphic" manner, because the forms of critique are inextricable from its object or content; my folly here is that of classic Aristotelian hylomorphism. Such an understanding of critique, in which form of expression and object of critique are co-adapted, might be best illustrated by the Derridean modality of deconstruction; the manipulations to which Derrida submits his signs enact his linguistic and philosophical critique of the Western metaphysics of presence, of logocentrism, and so on.

One might object that programming *can* be used to build a technology that opposes, say, capitalism or the state. But this is irrelevant to a description of programming "in general," as it merely addresses the applications, rather than the formal and technical imperatives, of programming. Moreover, such a "critical" technology could be constructed following the same formal procedures and diagrams of optimization and security as a Google product; although the "content" seems correct, it does not operate through formal means married to their ends. Conversely, one might point to uses of programming languages that subvert their intended usage by following certain aesthetic, rather than functional criteria: code poems, for example, or coding competitions where the aim is to write a program as confusing as possible, or with the fewest number of characters, and so on.[165] But besides lacking any explicit political aim, such "ironic" uses of programming languages, although not entirely divorced from actual programming practice, evade the technical considerations that define this specific kind of reading or writing—i.e., that make programming programming.

---

[164] Foucault, "Critique," 28.

[165] The aims of the International Obfuscated C Code Contest, for example, are: "To write the most Obscure/Obfuscated C Program within the rules; to show the importance of programming style, in an ironic way; to stress C compilers with unusual code; to illustrate some of the subtleties of the C language." ("Goals of the Contest," IOCCC, accessed April 19, 2020, https://www.ioccc.org/) Code golf is a competition to implement an algorithm with the fewest characters possible. I would add that, even within functionality-oriented software engineering, programming produces aesthetic objects; the figures described in Chapter 1 are one such example.

Irony and the (originary) technicity of critique

Despite his reservations, Foucault does give a kind of "general definition" of critique—but one that is equally difficult to reconcile with our description of programming. "Critique is the movement by which the subject gives himself the right to question truth on its effects of power and question power on its discourses of truth."[166] Besides the differences in *content* we have just described— programming itself does not question truth or power—we also have, here, a *formal* dissimilarity. The "movement" Foucault describes here is that of the old Greek technologies of the self: a reflexive self-allocation, a "conversion to self" that sets up an impenetrable subject. In Chapter 3, we emphasized that the subjectivation of programming follows an opposite trajectory: outwards, not "back home." Indeed, in this lecture, Foucault says that critique is a *virtue.*[167] "Virtue" is *virtù*, masculinity, the Greek technology of the self found in the self-reflexive *enkrateia*: "In this ethics of men made for men, the development of the self as an ethical subject consisted in setting up a structure of virility that related oneself to oneself."[168] Here it is important to remember that this virtuous practice on the self is inextricable from the isomorphic "relation of domination, hierarchy, and authority that one expected, as a man, a free man, to establish over his inferiors."[169] If, in my comparison of programming to Foucauldian critique, I have evaded this modality of self-reflexive insubordination, it is in part because I find this kind of "virtue" a rather tough pill to swallow. In fact, it seems to me that, if the programmer-as-subject *does* give "himself the right to question truth on its effects of power and question power on its discourses of truth," then this modality is easily reconcilable with what is most destructive in Silicon Valley's mantra of "move fast and break things": the heedless "innovative disruption"

[166] Foucault, "Critique," 32.
[167] Ibid., 25.
[168] Foucault, *Use*, 83.
[169] Ibid.

of existing practices and institutions that goes hand-in-hand with a reckless disregard for the law, the privacy of citizens, and so forth.[170]

Although the two are intimately related, I have chosen to emphasize the "pharmacological" strand of Foucault's thought—the impulse to "get free of oneself" via a foreign discourse—rather than self-reflexivity. Judith Butler ends her "What is Critique? An Essay on Foucault's Virtue" with a description of an instance of Foucault's own performance of critique in the same 1971 lecture. She describes how, in response to a question on the nature of the "will not to be governed," he says that it is:

> '*like* an originary freedom' and something '*akin* to the historical practice of revolt' [Butler's emphasis]. Like them, indeed, but apparently not quite the same. As for Foucault's mention of 'originary freedom,' he offers and withdraws it at once. … What discourse nearly seduces him here, subjugating him to its terms? And how does he draw from the very terms that he refuses? What art form is this in which a nearly collapsible critical distance is performed for us? And is this the same distance that informs the practice of wondering, of questioning?[171]

This gesture, Butler says, through which Foucault flirts with the idea of an "originary freedom" that, as a transcendental value, is antithetical to his archaeological or genealogical project, is a virtuous act of courage in which Foucault risks *himself* as a subject in relation to his own discourse, "at the limit of the epistemological field" that he has himself constructed. This performance is exemplary of the aesthetics of existence Foucault calls the technology of the self. As a courageous act of speech, it is like the Greek *parrhēsia*, but a kind of *parrhēsia* that leads *out* of one's own subjectivity, rather than back to it.

No doubt, computer programming *is not* critique. But, if we say that it is "like" critique, where does that leave us? Or rather, where does it leave critique? Here I am reminded, also, of Donna Haraway's opening to the "Cyborg Manifesto," in which she explains that the cyborg is

---

[170] "Innovative disruption" was coined by Harvard Business School professor Clayton Christensen in the 1990's, and has since served as a Silicon Valley buzzword. (Nitasha Tiku, "An Alternative History of Silicon Valley Disruption," *Wired*, October 22nd, 2018, https://www.wired.com/story/alternative-history-of-silicon-valley-disruption/). "Move fast and break things" is Mark Zuckerberg's now-famous motto (Hemant Taneja, "The Era of 'Move Fast and Break Things' is Over," *Harvard Business Review*, January 22nd, 2019, https://hbr.org/2019/01/the-era-of-move-fast-and-break-things-is-over).

[171] Judith Butler, "What is Critique? An Essay on Foucault's Virtue," *transversal texts*, May 2001, https://transversal.at/transversal/0806/butler/en.

an *ironic* political myth: "Irony is about contradictions that do not resolve into larger wholes, even dialectically, about the tension of holding incompatible things together because both or all are necessary and true. Irony is about humor and serious play. It is also a rhetorical strategy and a political method."[172] If we say, while delighting in its impiety but also very seriously, that programming is like critique, then perhaps this is also a rhetorical gesture that might lead to a kind of strategic re-positioning of critique, highlighting precisely its status *as* strategy, its functional or technological qualities.

As Foucault says in "What is Critique?", critique is an "instrument, a means for a future or a truth that it will not know nor happen to be."[173] Although the emphasis of the lecture lies elsewhere, this formulation is nevertheless remarkable because it is in direct contrast to the tradition of thought exemplified by Horkheimer's "Critique of Instrumental Reason." The critic, like the programmer, is a *user.* Understanding critique as technological makes it coextensive with the whole range of strategies through which living things adapt or respond to their circumstances via technology. This does not entail a reduction of human behavior to mechanistic response; rather, it speaks to the manner in which humanity emerges only through and with technology.[174] Foucault's teacher, Georges Canguilhem, makes the non-mechanistic character of such an assertion clear via an analogy to a discursive situation: "The relationship established between the living and its milieu is like a *debate* in which the living brings its own norms of appreciating the situation, where it is in command of the milieu and accommodates itself to it."[175] The relationship between life and milieu is neither antagonistic nor unidirectionally deterministic; rather, both inform the other. Similarly, the programmer, "in conversation" with the technological milieu (the computer), both imposes his own will on it and must adapt himself to it. This is what it means to fold the outside into the inside, and, as a critic, to develop a strategic response to power without being able to step outside of its effects. Programming is like reading and writing

---

[172] Haraway, "Cyborg," 1.
[173] Foucault, "Critique," 25.
[174] A position perhaps developed most extensively and explicitly by Bernhard Stiegler, following Derrida, under the concept of "originary prostheticity." See, for example: "Who? What? The Invention of the Human," *Technics and Time, 1: The Fault of Epimetheus*, trans. George Collins and Richard Beardsworth (Stanford: Stanford University Press, 1998), 134-179.
[175] Georges Canguilhem, "The Living and its Milieu," trans. John Savage, *Grey Room* (vol. 3: Spring 2001), 21. Emphasis mine.

in that all three are animated by a certain technological orientation: a specific question or problem, a definite constellation of power, the immersion within and creation of a terrain. Critics have labelled the philosophers discussed here—Foucault, Derrida, Deleuze—as thinkers of "originary technicity."[176] Our re-enactment of the scene Butler examines vis-à-vis Foucault might, therefore, be to say: critique is something like an originary technicity, something akin to the practice of computer programming.

If, here, a comparison of programming and critique on the formal level has been made possible by a certain evacuation of the political content of the latter, then an appreciation of the significance of this elision entails recognition of another manner in which the two are similar: as technological practices that are always a specific, strategic response. The rhetorical, quasi-ironic positioning of programming *as* reading, writing, or critique might allow one to develop new directions in which critique might move in relation to the semiotic forms and diagrams of power of the digital age. We began Chapter 2 with a description of the programmatic disruptions to which Derrida submitted his writing of *Circonfession* and *The Post Card.* Such techniques of re-formatting, cutting and pasting, and so on, as he acknowledged in 1995, are no longer "disobedient," semiotically or politically, in the time of digital technology. "It was theorized and it was done—then." Rather, "we must invent other 'disorders,' ones that are more discreet, less self-congratulatory and exhibitionist, and this time contemporary with the computer."[177] In order to construct such formal innovations, however, one must first understand the forms of reading and writing native to programming languages—to which this project has attempted to contribute. In the same way, however, that the pleasures of programming exceed the solution of any specific problem, in addition to any such "strategic" aims of my argument, this writing also developed out of a desire merely to document the contorted aesthetic figures and semiotic constellations produced in this particular mode of reading and writing. The computer diagram that scales levels of technicity, the software supplement, the variability of materially-determined meaning, the

---

[176] See, for example: Arthur Bradley, *Originary Technicity: The Theory of Technology from Marx to Derrida* (New York: Palgrave Macmillan, 2011). This also, of course, includes Simondon. Althusser's theory of ideology might also be subsumed under this concept.
[177] Derrida, "Word," 25.

pharmacology of debugging—all these expressions of a foreign discourse hint, in both their familiarity and their intoxicating strangeness, at "the possibility of a difference, of a mutation, of a revolution in the propriety of symbolic systems" as well as a mode of subjectivity and resistance, an aesthetic experience (for Foucault, they are the same thing), that makes us other than what we already are.[178] Like critique, a trip, an exercise, an "instrument, a means for a future or a truth that [we] will not know nor happen to be."[179]

[178] Barthes, *Empire*, 4.
[179] Foucault, "Critique," 25.

Thank you

Thank you so very much to Péter Szendy, for listening as well as reading, for being willing to go along when the aim was nowhere in sight, and whose insightful guidance and dialogue both transformed my writing and led it back to itself.

Joan Copjec's thoughtful and meticulous interventions completely renewed this project when I didn't know where it could go; thank you for opening up to me so many ideas that I am happy to know will continue to carry me in the future.

My work here is also hugely indebted to Ellen Rooney's teaching on form and critique (faithful, as Donna Haraway writes, "as blasphemy is faithful"), as well as to the members of the 2019-2020 Pembroke seminar, "On the Question of Critique."

Thanks to the scores of Brown Computer Science undergraduate teaching assistants who taught me diagramming.

And my parents, Jutta and Jim, for everything!

Most of all, thank you to the many many friends/classmates who have read, edited, and shaped my thinking, including: Miles, Jonah, Liby, and Isabelle — <3.

BIBLIOGRAPHY

Aarseth, Espen. *Cybertext: Perspectives on Ergodic Literature.* Baltimore: Johns Hopkins University Press, 1997.

Althusser, Louis. *Reading Capital.* London: New Left Books, 1970.

Barthes, Roland. *Empire of Signs*. Translated by Richard Howard. New York: Hill and Wang, 1982.

Bratton, Benjamin. *The Stack: On Software and Sovereignty.* Cambridge, MA: MIT Press, 2015.

Butler, Judith. "What is Critique? An Essay on Foucault's Virtue." *transversal texts.* May 2001. https://transversal.at/transversal/0806/butler/en.

Canguilhem, Georges. "The Living and its Milieu." Translated by John Savage. *Grey Room* (vol. 3: Spring 2001): 7-31.

Chun, Wendy Hui Kyong. "On Software, Or the Persistence of Visual Knowledge." *Grey Room* (vol. 18: Winter 2004): 26-51.

———. "On Sourcery, or Code as Fetish." *Configurations* (vol. 18:3): 299-324.

———. *Programmed Visions: Software and Memory.* Cambridge, MA: MIT Press, 2011.

———. *Updating to Remain the Same: Habitual New Media.* Cambridge, MA: MIT Press, 2016.

Deleuze, Gilles. *Foucault*. Translated by Seán Hand. Minneapolis: University of Minnesota Press, 1988.

———. *Two Regimes of Madness: Texts and Interview 1975-995.* Translated by Ames Hodges and Mike Taormina. Edited by David Lapoujade. Cambridge, MA: MIT Press, 2006.

Derrida, Jacques. *Dissemination.* Translated by Barbara Johnson. Chicago: University of Chicago Press, 1983.

———. "Letter to a Japanese Friend," in *Derrida and Différance.* Edited by David Wood and Robert Bernasconi. Evanston: Northwestern University Press, 1988.

———. *Limited Inc.* Translated by Samuel Weber and Jeffrey Mehlmann. Evanston: Northwestern University Press, 1988.

———. *Of Grammatology.* Translated by Gayatri Chakravorty Spivak. Baltimore: Johns Hopkins University Press, 2016.

———."Paper or Myself, You Know… (new speculations on a luxury of the poor)." Translated by Keith Reader. *Paragraph* (vol. 21:1): 1-27.

———. *Points…Interviews, 1974-1994.* Translated by Peggy Kamuf. Edited by Elisabeth Weber. Stanford: Stanford University Press, 1995.

———.*The Post Card: From Socrates to Freud and Beyond.* Translated by Alan Bass. Chicago: University of Chicago Press, 1987.

———. "Typewriter Ribbon: Limited Ink (2) ("within such limits")." Translated by Peggy Kamuf. *Material Events: Paul de Man and the Afterlife of Theory.* Edited by Tom Cohen, et al. Minneapolis: University of Minnesota Press, 2001.

———. "White Mythology: Metaphor in the Text of Philosophy." Translated by F.C.T Moore. *New Literary History* (vol. 6:1): 5-74.

———. “The Word Processor.” *Paper Machine.* Translated by Rachel Bowlby. Stanford: Stanford University Press, 2005. 19-32.

“Dynamicland.” *Dynamicland.* Accessed March 31, 2020. https://dynamicland.org/.

“The Effects of Computer Programming on the Brain.” *Grindd.* July 5, 2013. Accessed March 31, 2020. http://www.grindd.com/blog/2013/07/the-effects-of-computer-programming-on-the-brain/.

Eisenman, Peter. *Diagram Diaries.* New York: Universe, 1999.

Ernst, Wolfang. *Chronopoetics,* trans. Anthony Enns. London: Rowman and Littlefield, 2016.

Fazi, Beatrice. *Contingent Computation: Abstraction, Indeterminacy, and Experience in Computational Aesthetics.* London: Rowman and Littlefield, 2018.

Fisher, Mark. *Flatline Constructs: Gothic Materialism and Cybernetic Theory-Fiction.* Brooklyn: Exmilitary Press, 2018.

Foucault, Michel. *About the Beginning of the Hermeneutics of the Self: Lectures at Dartmouth College, 1980.* Chicago: University of Chicago, 2015.

———. *The Care of the Self.* New York: Pantheon, 1986.

———. *The Courage of the Truth (The Government of Self and Others II): Lectures at the Collège de France, 1983-84.* Translated by Graham Burchell. Edited by Fréderic Gros. New York: Picador, 2008.

———. *Discipline and Punish.* Translated by A. Sheridan. New York: Pantheon, 1995.

———. *The Government of Self and Others: Lectures at the Collège de France, 1982-83.* Translated by Graham Burchell. Edited by Fréderic Gros. New York: Picador, 2008.

———. *The Hermeneutics of the Subject: Lectures at the Collège de France, 1981-82.* Translated by Graham Burchell. Edited by Fréderic Gros. New York: Picador, 2008.

———. *Technologies of the Self: Lectures at University of Vermont Oct. 1982.* Amherst: University of Massachusetts Press, 1988.

———. *The Use of Pleasure.* New York: Pantheon, 1984.

———. “What is Critique?” Translated by Lysa Hochroth. *The Politics of Truth*. Edited by Sylvère Lotringer and Lysa Hochroth. New York: Semiotexte, 1997. 23-82.

Frabetti, Federica. *Software Theory: A Cultural and Philosophical Study.* London: Rowman and Littlefield, 2014.

Galloway, Alexander. *The Interface Effect.* Cambridge, UK: Polity, 2012.

———. “Language Wants to Be Overlooked.” *Journal of Visual Culture* 5 (vol. 3:2006): 316-331.

“Games.” Massey University of New Zealand. Accessed March 31, 2020. https://www.massey.ac.nz/~mjjohnso/notes/59302/l05.html.

“Goals of the Contest.” International Obfuscated C Code Contest. Accessed April 19, 2020. https://www.ioccc.org/.

Haraway, Donna. “A Cyborg Manifesto: Science, Technology, and Socialist-Feminism in the Late Twentieth Century.” *Manifestly Haraway.* Minneapolis: University of Minnesota Press, 2016.

Hayles, N. Katherine. *My Mother Was A Computer.* Chicago: University of Chicago Press, 2005.

———. "Print is Flat, Code is Deep: On the Importance of Media-Specific Analysis," *Poetics Today* (vol. 25:1): 67-90.

———. *Writing Machines.* Cambridge, MA: MIT Press, 2002.

Healy, Kieran. "Fuck Nuance." *Sociological Theory* (vol. 35:2): 118-127.

Hui, Yuk. *On the Existence of Digital Objects.* Minneapolis: University of Minnesota Press, 2016.

Joque, Justin. *Deconstruction Machines: Writing in the Age of Cyberwar.* Minneapolis: University of Minnesota Press, 2018.

Kittler, Friedrich. "There is No Software." *CTheory* (1995). Accessed March 31, 2020. www.ctheory.net/articles.aspx?id=74.

Krishnamurthi, Shriram. "Scope," *Programming and Programming Languages.* Accessed March 31, 2020. https://papl.cs.brown.edu/2018/Interpreting_Functions.html#%28part._.Scope%29.

Manovich, Lev. *The Language of New Media.* Cambridge, MA: MIT Press, 2001.

Parisi, Luciana. *Contagious Architecture: Computation, Aesthetics, and Space* .Cambridge, MA: MIT Press, 2013.

"Priority Queue (Java Platform SE 7)." *Oracle.* 2018. Accessed March 31, 2020. https://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html.

Rooney, Ellen. "Live Free or Describe: The Reading Effect and the Persistence of Form." *differences* 21 no. 3 (2010): 112-139.

———. "Symptomatic Reading is a Problem of Form." *Critique and Postcritique.* Edited by Elizabeth Anker and Rita Felski. Durham: Duke University Press, 2017). 127-152.

Simondon, Gilbert. *On the Mode of Existence of Technical Objects,* trans. Cecile Malaspina and John Rogove. Minneapolis: University of Minnesota Press, 2016.

Taneja, Hemant. "The Era of 'Move Fast and Break Things' is Over." *Harvard Business Review.* January 22nd, 2019. https://hbr.org/2019/01/the-era-of-move-fast-and-break-things-is-over.

Tiku, Nitasha. "An Alternative History of Silicon Valley Disruption." *Wired.* October 22nd, 2018. https://www.wired.com/story/alternative-history-of-silicon-valley-disruption/.

van Zanten, Boris Veldhuijsen. "The very first recorded computer bug," *TNW.* Sep 18, 2013. https://thenextweb.com/shareables/2013/09/18/the-very-first-computer-bug.

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

Wiener, Norbert. *Cybernetics, or Control and Communication in the Animal and the Machine.* Cambridge, MA: MIT Press, 1948.

Images

Doeppner, Thomas W. “C Pointers.” Lecture slides for CSCI0330: Introduction to Computer Systems, Fall 2017.

“Games,” Massey University of New Zealand, accessed March 31, 2020, https://www.massey.ac.nz/~mjjohnso/notes/59302/l05.html.

“How to perform recursion operation in Java,” CodingSec, accessed March 20, 2020, https://codingsec.net/2016/09/perform-recursion-operation-java/.

“Priority Queue Java,” JournalDev, accessed April 17, 2020, https://www.journaldev.com/16254/priority-queue-java.

Wong, Stephen. “Eclipse Debug Perspective Screen Shot,” COMP310, Rice University, 2017, accessed April 10, 2020, https://www.clear.rice.edu/comp310/Eclipse/debugging.html.