1 Introduction

Code Heard 'round the World

```
if n_elements(yrloc) ne n_elements(valadj) then message,'Oooops!'
```

—briffa_sep98_e.pro

Who reads computer source code, and what does it mean to them? Pundits and poets, hacktavists and humanities professors, lawyers and laypeople—far more than just programmers are reading code for a wide range of reasons. Whether finding Easter eggs or getting a whiff of code smells,[1] seeking aesthetic pleasure or searching for ways they have been manipulated, the curious have a sense that something awaits them in the code. If we are going to take up this communal practice of reading code, we must read it critically, for the way we read code is just as important as the fact that we are reading code. Consider the following example.[2]

In 2009, leaked emails from the Climate Research Unit (CRU) of England's University of East Anglia seemed to hand a smoking gun to climate change deniers, proof positive that climate change is a fabrication. These emails included code used to model climate change, and the comments in that code seemed to indicate a manipulation of the data. Soon the blogosphere was buzzing with quotations from the code, particularly from a file called Harry_Read_Me, named for its programmer, Ian Harris, who collaborated with Keith Briffa on the code.[3] As he attempts to document his progress, making sense of and reconciling the various data, Harris writes in the code comments, "What the hell is supposed to happen here? Oh yeah—there is no 'supposed,' I can make it up. So I have : -)."

Other comments throughout the leaked files express the frustrations of a programmer as he complains about the lengths to which he has to go to reconcile conflicting

and at times contradictory data. Seizing on this code as raw meat, climate change contenders called the comments proof that the data used to demonstrate global warming was bunk, a conspiracy in the code. This source file purportedly provided a glimpse behind the curtain of a mass deception, and the staging ground for that charade was the computer source code itself.

The public criticism did not stop at the code's comments, either; conspiracy hunters also dove into the workings of the code, written in Interactive Data Language (IDL), discovering what Harris had labeled a *fudge factor*. Consider this excerpt (elided with [...]):

```
;****** APPLIES A VERY ARTIFICIAL CORRECTION FOR DECLINE*********
;
yrloc=[1400,findgen(19)*5.+1904]
valadj=[0.,0.,0.,0.,0.,-0.1,-0.25,-0.3,0.,-0.1,0.3,0.8,1.2,1.7,$
2.5,2.6,2.6,2.6,2.6,2.6]*0.75          ; fudge factor
if n_elements(yrloc) ne n_elements(valadj) then message,'Oooops!'
;
[...]
;
; APPLY ARTIFICIAL CORRECTION
;
yearlyadj=interpol(valadj,yrloc,x)
densall=densall+yearlyadj
```

In brief, the code takes a set of data, for years 1400–1994, and adjusts that data by adding, or subtracting in the case of the negative numbers, another set of numbers (listed as `valadj`, presumably for *value adjust*). The code makes these adjustments using what programmers call *magic numbers*, or unnamed constants, numbers introduced into the code with no explanation of their reference (Farrell 2008, 22). By adjusting the numbers in this way, the programmer created a data set that conforms better to his expectations. The *fudge factor* comment labels the numbers that adjust the data set.[4] Semicolons precede content that is not to be processed—in other words, comments in the code (and lines with only white space). The all-caps header comment stating that this is "a very artificial correction" seems to announce unambiguously that this code is pure manipulation. However, those labels lack a bit of context.

Rather than manufacturing some great deception, this code was written as an interim step in the process of consolidating several sources of data. Although it could

be said that *all* code represents only one iteration of work in progress, this particular file was acknowledged to be provisional by its developers. Harris's other writings identify this adjustment in his code as a temporary placeholder measure; these markers in the code identify places where he had to temporarily correct a discrepancy. The leaked code was apparently written while Harris was preparing a paper with his team about the disparity between tree records and temperatures, entitled "Trees Tell of Past Climates: But Are They Speaking Less Clearly Today?" (Briffa, Schweingruber, Jones, Osborn, Harris, et al. 1998). In other words, this code was not written as the final word in climate change but to label and investigate a problem observed in the recorded data, in which latewood density (the measured density of tree rings) did not correlate to changes in temperature. The programmer wrote in this fudge factor temporarily while trying to account for the discrepancy, only to rewrite the code in the same year in a way that addresses the divergence more systematically.[5]

However, despite the meaning of the code to its programmer, the code came to signify something very different in the broader world. News reports and amateur bloggers seized upon the files, posting excerpts in their diatribes. Code was literally becoming the means of debate, used as evidence in arguments for and against the scientific validity of climate change. The damning comments were featured on major networks, such as the BBC and CBS, as well as in news magazines and newspapers. Bloggers seized upon the comments of the Harry_Read_Me.txt file and lambasted the coder for his fudge factor. The files were referred to as "the smoking gun" of climate change fabrication. As one blogger posted, "things like 'fudge factor' and 'APPLIES A VERY ARTIFICIAL CORRECTION FOR DECLINE' don't require programming expertise to understand" (Wheeler 2009). But apparently they do. Even that blogger recommends that readers consult someone who knows how to read code before exploring on their own. Regardless of the reader's training or comprehension, code had become a means of public debate.

In our digital moment, there is a growing sense of the significance of computer source code. It has moved beyond the realm of programmers and entered the realm of mainstream media and partisan political blogosphere. Those discussing code may not be programmers or have much fluency in the languages of the code they are quoting, but they are using it to make, refute, and start arguments. There is also a growing sense that the code we are not reading is working against our interests. Pundits and academics alike are challenging algorithms used for everything from police profiling to health insurance calculations. Scholars, such as Safiya Noble and Joy Buolamwini of the Algorithmic Justice League, have called public attention to evidence of bias in AI systems and other software. However, often these discussions are limited to assessing

the algorithms based on their effects. As Noble writes, "Knowledge of the technical aspects of search and retrieval, in terms of critiquing the computer programming code that underlies the systems, is absolutely necessary to have a profound impact on these systems" (2018, 26). Although the code bases for many proprietary systems are inaccessible to us, an analysis of algorithms can go further through an analysis of the code.

Without access to the code, whether because it is proprietary or generated on the fly, as in the case of some machine-learning algorithms, analysts can only comment on the apparent operations of the code based on its effects. The operations of the code are left in the hands of those who can access it, usually those who have made or maintain it, and those who can read it. If code governs so much of our lives, then to understand its operations is to get some sense of the systems that operate on us. If we leave the hieroglyphs to the hierarchs, then we are all subjects of unknown and unseen processes. That anxiety drove online communities to pour through the leaked climate code—and led them to misread it. This example demonstrates that it is not enough to understand what code does without fully considering what it means.

Like other systems of signification, code does not signify in any transparent or reducible way. And because code has so many interoperating systems, human and machine-based, meaning proliferates in code. To the programmer, this fudge factor may have been a temporary patch in the larger pursuit of anomalies in the data, marking the code to bracket a question he wished to pursue later. However, read by a different audience and from a more suspicious perspective, that temporary solution becomes outright deception. An editorial in the magazine *Nature* claimed, "One lesson that must be taken from Climategate is that scientists do not get to define the terms by which others see them and their place in society" ("Closing the Climategate" 2010). The same can be said for code. Its meaning is determined not only by the programmer's intention or by the operations it triggers but also by how it is received and recirculated. The history of the misinterpretations of the Climategate code becomes part of its overall meaning.

That is not to argue that code can be removed from context (though portions of code are frequently recontextualized) or that code means whatever people say it means. Rather, the meaning of code is contingent upon and subject to the rhetorical triad of speaker, audience (both human and machine), and message. Although even that classic rhetorical triad is a bit poor when explaining a system of communication that is to varying degrees tied to hardware, other software, and state. In the process of its circulation, the meaning of code changes beyond its functional role to include connotations

and implications, opening to interpretation and inference, as well as misinterpretation and reappropriation. The Climategate code is forever tied to these online debates, and that history gives it significance far beyond the designs of its creators. Code is a social text, the meaning of which develops and transforms as additional readers encounter it over time and as contexts change. That is the argument and provocation of this book.

It is time to develop methods of tracing the meaning of code. Computer source code has become part of our political, legal, aesthetic, and popular discourse. Code is being read by lawyers, corporate managers, artists, pundits, reporters, and even literary scholars. Code is being used in political debate, in artistic exhibitions, in popular entertainment, and in historical accounts. As code reaches more and more readers and as programming languages and methods continue to evolve, we need to develop methods to account for the way code accrues meaning and how readers and shifting contexts shape that meaning. We need to learn to understand not only the functioning of code but the way code signifies. We need to learn to read code critically.

But reading code requires a new set of methods that attend to its specific contexts, requirements, and relationships. As the next examples demonstrate, communication via code is hardly straightforward.

A Job Interview

Let us consider a question: What does code mean? First, it would be useful if I defined what I mean by *code*. For now, I will say, *computer source code*. Then I should probably say what I mean by *mean*. But such an approach is full of folly. So let me restate the question with another, expressed in relatively unambiguous words.[6]

The new phrasing of the problem could take the form of a job interview.[7] For the sake of this argument, let's imagine two programmers, a man and a woman, who are applying for the same job. Before they are interviewed in person, they are sequestered in a room and asked to solve a popular challenge, to write a program that computes an anagram of a string. An anagram contains all the same characters of the initial string or word in a different order (e.g., *critical code* becomes *circa dice lot*), although the code for this challenge does not have to produce recognizable words, just rearranged characters. At the end of their challenge, the programmers will submit their code electronically so the employers will not know who wrote which version.

One of the candidates submits the following JavaScript code:

```
function anagram(text) {
   var a = text.split("");
   for (var i = 0; i < a.length; i += 1) {
       var letter = a[i];
       var j = Math.floor(Math.random() * a.length);
       a[i] = a[j];
       a[j] = letter;
   }
   return a.join("");
}
```

Simply put, this code creates a function (anagram) that splits the string up into an array of individual characters and then repositions the items in that array (or the letters) in a random order before combining them.

However, the other candidate submits a very different solution:

```
function anagram(text) {
   return text.split("").sort(function () {return 0.5-Math.
random()}).join("");
}
```

This approach performs the same operation but accomplishes its task in one line of source code. This candidate splits any word into characters, reorders them randomly, and then joins the letters again. However, this code takes an unusual approach by passing to the sort function another function, which chooses a random value between 0.5 and 0.5. As a result, instead of comparing each item to the others—for example, which is greater or lesser—the sort function assigns that evaluation at random, performing a kind of end run around sorting. This code in effect replaces the sorting operation with coin tosses (plus the potential for a "0," in which case the two items are considered equal, as if the coin had landed on its edge).[8]

Which programmer will win the job? Certainly, the one who wrote the stronger code—but which one was that? The answer is clear: it depends on how the readers—in this case, the people doing the hiring—interpret the code.

On the one hand, the first version lays out its steps clearly. The process is easy to identify, and each step uses fairly basic, straightforward techniques. On the other hand, the second function may seem more clever for its concision. It uses the more sophisticated comparator function. This concept is typically not taught until more advanced

courses because its operations are specific, even idiosyncratic, to this language. No doubt, programmers who read these code samples probably have already begun to form impressions of the two candidates in their minds: the one careful, organized, ordinary, perhaps fastidious; the other a wit with a tendency to show off. But every business is different.

Different companies, different programming cultures, different priorities, different work tasks require and desire different abilities. One set of employers might value most the code that is easy to read, modify, and sustain. Others might see the one-line solution as a sign of strong programming instincts, the ability to fully internalize the coding structures.[9] Interpreting further, each employer might have different assumptions about which code was written by the man and which by the woman. (Perhaps you have already made such assumptions.) Depending on their biases or hiring needs, they might favor one or the other, again depending on how they have read gender out of or into the presentation of the code. One need not look any further at the recent exposure of the gender divide in computer culture than the recent events at Google to see the crisis in gender equity in Silicon Valley and beyond.[10] This example and the use of such programming challenges in hiring suggests that code conveys more than mere ability: it expresses identity, even if these challenges are used primarily to make hiring decisions less dependent on other channels of identity, such as résumés or in-person interviews.

The programmers, aware of these possible interpretations, are choosing their strategies as well, realizing that they are representing themselves to the prospective employer. They cannot assume their code will be interpreted in a particular way. Employers may see the one-liner as a sign of a tendency to obfuscate or the multiline solution as a sign of a pedestrian approach to problems. The programmers may be torn between conflicting ideals and aesthetics taught in their programming classes or developed by interacting with other programmers professionally or otherwise online. They also may be influenced by their exposure to or immersion in various programming paradigms or languages that emphasize one virtue (say, reusability) over another (legibility or optimization).

Code in this example offers a medium for a *Turing test*, Alan Turing's thought experiment that challenges a computer to pass itself off as a human. However, rather than a computer and a human trying to prove who is more skilled at speaking human language, this scenario presents two programmers attempting to prove who can speak the language of computation more fluently, or at least most successfully in the dialect of the corporation from which they are seeking employment.[11] Only the surface challenge asks whether or not the human can speak in a way the computer understands.

The deeper challenge asks the programmers to communicate who they are to other humans, as coworkers and collaborators, through their use of code. Their code is not so much a litmus test, proving whether they can perform the task, as it is an essay exam, communicating character through process, values, and approaches to challenging questions. Simply put, code in this test proves again to be more than a functional tool: it is a means of expression between humans through communication with machines.

The many trade-offs in the minds of the programmers faced with this task and the wide variety of ways of reading these programs demonstrate just a portion of the complexity involved in the act of communicating through code. The meaning of the code goes beyond the functioning of the program because both programs perform the same task—a task which itself seems to have little significance for either party, though so much (i.e., employment) is at stake. Instead, the code itself is located within a broader communication exchange, one that is bound to time as marked by developments in programming paradigms, languages, hardware, and networks. Certainly, not everyone who reads code, as in this case with the employers, even shares the goal of realizing that particular software process. This is crucial because much writing about code, from textbooks to style guides, suggests that code is being written primarily to create a certain effect through software, which neglects the use of code in training exercises and artistic explorations, just to name two examples in which the computer program was not written to execute a solution to a problem. Here, the problem of the anagram was the sword in the stone, the evocative object.[12] This book will explore the ways source code becomes meaningful through its circulation in diverse contexts, how its tokens convey meaning beyond their functionality, and therefore how code serves as a communication medium including and beyond the realms of practical application on specific machines.

In this hiring example, source code is not merely the objective right answer to the straightforward mathematical question, not something that can be assessed as empirically better or worse. In fact, the notion of empirical truths or unquestionable values in computer programming proves to be even less stable than it is in the material world, the social constructions of which have been well articulated by theorists and philosophers. In either solution, the code is merely one version of how this process may be implemented. Its final form is the result not of mathematical certainty but of collected cultural knowledge and convention (cultures of code and coding languages), haste and insight, inspirations and observations, evolutions and adaptations, rhetoric and reason, paradigms of language, breakthroughs in approach, and failures to conceptualize.

Code is a meaningful measure of a job candidate's abilities and predilections because code communicates more than merely its functionality; as an interdependent object circulating through culture, it amasses even more meaning. The examples from the job interview situation were hypothetical but typical. Once we return our attention to code from the real world with a history of revision and correction, funding and fudging, functioning or failing, we have even more to discuss—or, as we say in the humanities, to unpack. As Alan J. Perlis writes in the foreword to Harold Abelson and Gerald Jay Sussman's foundational *Structure and Interpretation of Computer Programs*, "Every computer program is a model, hatched in the mind, of a real mental process. These processes, arising from human experience and thought, are huge in number, intricate in detail, and at any time only partially understood" (1996, xi). Unpacking the meaning of these programs locked in their technosocial context is the job of critical code studies (CCS).

Protesting in Code

Code holds a meaning beyond, and growing out of, what it does. That meaning does not necessarily depend on code authors intending to communicate something to an audience beyond themselves, as in the case of code comments. Consider two rather different examples.

In the summer of 2011, at a political protest in India, a young woman appearing in a photograph that was later shared on the online discussion board Reddit holds a sign (figure 1.1), which reads:

```
#include <india.h>
#include<jan lokpal bill.h>
#include <students.h>
void main( )
{
  do
    {
      no_of_indians ++;
      printf("Protest continues.");
    } while(lokpal bill not passed);
  printf("Corruption free India");
  getch();
}
```

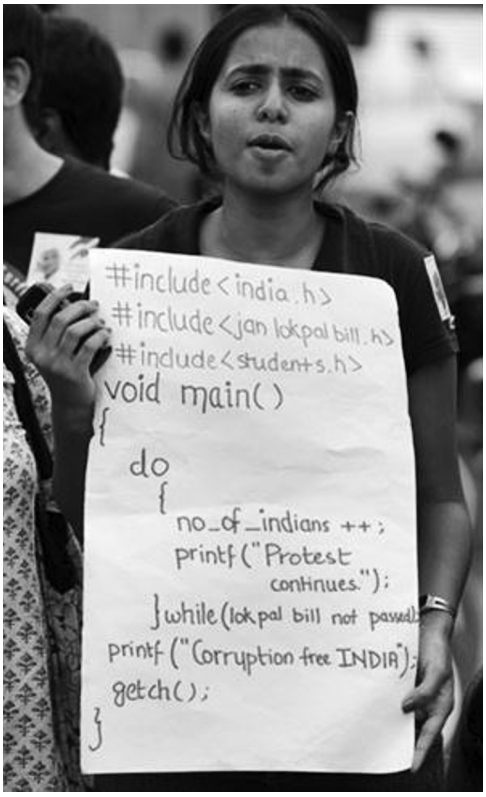

Figure 1.1
A woman holds a protest sign written in code.

The woman is protesting (mostly) in the programming language C. She is protesting by writing in code.

One interpretation of this sign is that the protester is calling for India, particularly the students primarily in engineering and computer science, to join in the support of the Jan Lokpal Bill. This code offers a little loop in which the number of Indians protesting should increase until the bill is passed, at which point, according to the program, India will be corruption free.[13]

Stepping through the functionality of this code, or pseudocode, helps to identify ways that it conveys its political message.[14] The code on her sign includes several libraries: `india`, `jan lokpal bill`, and `students`. In other words, it uses the preprocessing directive `#include` to load files with the names listed. The `.h` notation is the conventional way to name a header file, or files that are loaded first. In the context of

the protest, the code seems to be playing on the pun to "include" the people of India (represent the wishes of the people), to "include" the bill (pass the measure), and to "include" students (invite into the process students, the main audience of this sign). Placing these three items as header files symbolically gives them priority because their contents are established as fundamental declarations for the rest of the code.

The `void main()` call begins the code of this particular function.[15] Next, the code begins the function with a loop (`do`) that increases the number of Indians (`no_of_indians`) and prints the message "Protest continues." while the bill is not passed (a condition).[16] Once the loop ends, the function prints "Corruption free India." The `getch()` function means that inputting any character will exit out of the program. In the analogy, the role of this last instruction, or waiting for any key to be pressed before terminating the program, is not obvious. Perhaps at some point the program, and by consequence the protest, will need to be interrupted. Pressing a character to stop the program could represent pressing for more character (integrity) or impressing upon the character of activated citizens, although there are not enough context clues for a definitive reading. But by placing the code on a sign and holding it up in a public event as political expression, the woman holding the sign is inviting open public interpretation by noncomputational systems.

The Reddit discussion of the photo is 169 comments long (over four hundred if you add the separate threads under the "India" and "pics" Reddit forums) as commentators (aka Redditors) weigh in on the elegance of her code (or perceived lack thereof), global outsourcing, crowdsourcing code, and criticism of the way programming is taught in universities, not to mention discussion of the subject of her protest, the Jan Lokpal anticorruption bill in India. That bill, also known as the Citizen's Ombusdsman Bill, was designed to establish a corruption oversight body called Jan Lokpal, which had first been proposed in 1968. The 2011 campaign, driven by political activist Anna Hazare's hunger strike, gained widespread attention due to a social media campaign largely run on Twitter, targeting India's media-savvy middle class (Stancati and Pokharel 2011). To create a protest sign in computer code during this particular campaign is to address directly India's influential body of professionals in the technology area. To express a political message in code-like text is to address a readership skilled in reading computer code, their professional language.

The debate over this code vacillates between thoughtful consideration of her point and (more often) the kinds of trolling common to message boards such as Reddit. A good deal of the commentary focused on perceived deficiencies of the code, which incites remarks about her sexual desirability and her perceived inferiority as an alleged outsourced programmer.[17] By protesting in code, this woman, presumably an Indian

national, had triggered protectionist, elitist, and chauvinistic reactions from Redditors far beyond India, offering glimpses of *toxic geek masculinity*[18] or what I call *encoded chauvinism* (see chapter 5) that can overshadow programming cultures. Consequently, this code presents us with an opportunity to discuss not only its central topic, the Jan Lokpal Bill, but also the cultures of programming that emerge. To focus on the code of this sign, its competencies and whether it would compile, rather than its meaning in context, is to demonstrate the way the centrality of the machine, the need to make code utterances legible to the unambiguous processing of the compiler, takes precedence from and in turn deprecates other explorations of meaning. As one Redditor puts it in the top-voted comment, "I love that everyone in the comments immediately starts correcting errors in the code. Please never change internet" (account deleted, August 26, 2011, comment on "Protesting in C"), to which another replies, "Well if she's going to be making a sign that no one but programmers will understand, she might as well write it well" (m7600, ibid.). Style and technical validity clearly take priority in this coding community, at once identifying both the centrality of functionality in this unambiguous form of expression and the way that emphasis obscures other aspects of its communication. Critical code studies seeks to explore making these secondary meanings primary.

Admittedly, this protest sign hardly offers an everyday use of code. In fact, although the protestor is using code-like language, this sign has more in common with *codework*, a style of creative writing that uses code and code-like orthography to create poetic art, than it does with programming in the context of software development.[19] This code was written primarily to express a political position, not to produce a program. Nonetheles, this code speaks volumes in context. The fact that a woman in India holds up a sign written in C-like code, shows an expression of the intersection of gender, economics, and politics as women of the subcontinent face the gender divide in a growing and increasingly liberal middle class. Although there is much to discuss in this code extracted from a machine-centered programming context into a human-centered political forum, such as a march, I contend that even code written in an ordinary program to run on a computer can "speak," as Geoff Cox and Alex McLean (2012) put it in their book.

In contrast, consider a second example, selected for having as many parallels (e.g., perceived gender, nationality of the programmer, and aim for political expression) with the first as possible.[20] On the code repository GitHub, programmer Tapasweni Pathak has posted a project for a web app called Women on GitHub (Pathak et al. 2016).[21] This app, which makes use of a framework called Heroku, takes a list of names of female programmers "who inspired [the contributors] to code more." Two contributors,

Prabhanshu Attri and Fatima Rafiqui, have added code for displaying that list on a web page. The following is the PHP and HTML code that creates a grid of tables, each displaying a user in a profile box (figure 1.2), reminiscent of trading cards for baseball or soccer idols:

```
219 $count = 1;
220 while ($row = $result->fetch(PDO::FETCH_ASSOC)) {
221  echo '<div class="mdl-cell mdl-cell--3-col mdl-cell--4-col-
tablet mdl-cell--4-col-phone mdl-card mdl-shadow--3dp">
222    <div class="mdl-card__media user-img">
223      <img class="avatar" src="'.$row['avatar_url'].'"
alt="'.$row['name'].'">
224      <span class="id-element
mdl-typography--font-light">'.$row['id'].'</span>
225      <nav class="menu-'.$count.'">
            ... [ Navigation menu ] ...
234       echo '</nav>
235 </div>
236 <div class="mdl-card__supporting-text">
237 <span class="mdl-typography--font-light
mdl-typography--subhead">
238 <table>
239 <tr>
240   <td><i class="fa fa-user"></i>
241   <td><h4 class="android-header">'.$row['login'].'
</h4>
242</tr>';
243 if (strlen(trim($row['name'])) != 0 &&
!empty(trim($row['name'])))
244 echo '<tr>
245   <td>
246   <td>('.trim($row['name']).')
247 </tr>';
248 if ($row['company'])
249 echo '<tr>
250    <td><i class="fa fa-group"></i>
251          <td>'.$row['company'].'
252 </tr>';
```

```
253 if ($row['location'])
254 echo '<tr>
255      <td><i class="fa fa-location-arrow"></i>
256    <td>'.$row['location'].'
257 </tr>';
258 if ($row['created_at'])
259 echo '<tr>
260   <td><i class="fa fa-clock-o"></i>
261 <td>Joined on '.$row['created_at'].'
262 </tr>';
263 echo '</table>
264 </div>
```

This code produces the basic layout for the card. After setting the counter to one (`$count = 1;`), it loops through all of the data using a `while` statement. The subsequent lines

Figure 1.2
Sample image from Women on GitHub.

display aspects of user accounts using an `echo` statement (PHP for print or display).[22] Each element of the featured programmer's profile is surrounded by either `<div></div>` or `<span></span>` tags for formatting or styling that item, whether through placement, size, or other design elements. The `<class>` specifications indicate which formatting to apply from a Cascading Style Sheets (CSS) file.

The second section (starting at line 236) creates the lower half of the card using conventional code that can be read by anyone who has tried to make a webpage using tables. The code includes a `<div>` or div tag, which creates the subsection; `<table>` organizes the space; and `<tr>` creates rows and `<td>` cells in columns, producing a growing procession of inspiring women programmers. Each row has two columns, one for an icon and one for the information. Several conditional statements check whether particular items (e.g., company and link) exist before displaying them, or inserting the HTML to display them, using `echo` statements. The initials *fa* in the class refer to Font Awesome, an open-source set of icons created by Dave Gandy,[23] which further links this project to the open-source community and its ethos of collaboration. Font Awesome gives the community icons for making professional-grade web apps, rendering artifacts and objects legible in the contemporary web design ecology. So too is the code awesome for presenting, as visual icons, the many women programmers in a profession currently skewed heavily toward men.

As the illustration shows, these programmer baseball cards (as in figure 1.2) are fairly straightforward, presenting an avatar, username, company, location, and join date. One item stands out, though: a white number on a black background superimposed onto the avatar. No Font Awesome icon identifies that number, nor are there other clues as to its meaning, yet its prominence on the image gives it significance. It was not until I looked into the code that I knew its denotation, the GitHub user number (`.$row['id'].`, line 224). As it turns out, this number is not obvious in GitHub either because it is not listed on a user's profile page or in the URL for a user.[24] By looking in the code, I could find the denotation of this number, and that exploration led me to consider its connotation.

On the face of it, that tally number is only significant to the program that generated it; however, on further consideration, the number conveys additional meaning by tying each contributor to when she joined GitHub, indicating how many other people had previously registered. A complete set of these numbers would offer the rate at which GitHub's user base expanded and, when combined with other profile information, could tell us how the proportion of users who are women changed over time.[25]

The Women on GitHub software gives this number further rhetorical significance through placement. Layered over the avatar image in such a prominent place, the

number offers a form of authentication and perhaps even validation of these programmers: here are their official numbers, the computer-generated signs of their membership in this online community. The fact that the meaning of this number lies in the code is appropriate to the software's audience; it is written, like the protest code, for programmers who have the ability to interpret its meaning by inspecting the code. The repository invites these readers who code to explore, to discover, to contribute, and to join in, to be counted as another woman on GitHub or a confederate, extending, revising, or forking this code. In this way, the code has made meaningful to its viewers a number that otherwise only has meaning in the software, mirroring the process whereby we find a larger meaning of this number by exploring the computer source code, code designed to present the significant number of women programmers contributing to this platform for open-source development.

Compared to the codework of the protest sign, this excerpt offers mere ordinary code. Pathak says she does not even list the project on her resume (December 13, 2016, comment on Kalbhor 2016). Neither do the two programmers who created the web page. Nonetheless, code does not have to be extraordinary or difficult to read to be remarkable. Arguably, this code does more computational work than the protest sign because this code will produce effects on a web browser when it is run with the other files. Rather than calling to increase the number of student protestors, this code is part of a system that literally increases the number of women displayed on the Women on GitHub webpage, growing as contributors add names and as GitHub assigns their numbers. Unlike the sign that calls for inclusion, this code does the yeoman's work of building signs of inclusion and involvement, one table cell at a time.

Context also makes this code meaningful. This code was contributed by a collaboration between women and men to celebrate female programmers. Not only does the app display women who program, but the repository, too, speaks of women and men collaborating on open-source software development. Furthermore, this code speaks because it can function. Whereas the code on the sign would not compile, this web app code does. Whereas the protest code is displayed on a poster, this code lives in a repository where it can continue to grow and develop. Whereas the code on the sign inspired forum posts, this code has inspired collaboration, expanding, forking. The goal of this comparison is not to call one coding act better than another but to demonstrate the ways meaning in code arises from its context, not independent of its functionality but growing out of its functionality.

Although it is important to note the difference between code and codework, I do not want to call one of these examples a superior form of speaking through code or even more worthy of explication. We have to be mindful of a kind of chauvinism

that creeps into discussions of programming, what I call encoded chauvinism (see chapter 5),[26] whereby we assert a hierarchy based on an arbitrary judgment of what is "real" or "good" or "right" code. Surely that chauvinism is driving much of the commentary on the Reddit boards mentioned earlier, and though it can grow out of rigor and critical integrity, it typically serves to suppress, to denigrate, and to diminish the work of others in a way that is poisonous to programming culture and its development. To write code that runs is not more important than the creative act of taking code out into the streets on a protest sign. In fact, I read the second program through the lens of the first. I juxtapose these examples to model one kind of code-reading practice (code read by comparison to other code) and to argue that speaking in code does not require programmers to make code behave like spoken language or to create puns with code. A person writing what to them is ordinary, functional code is making meaning already. Critically reading code does not depend on the discovery of hidden secrets or unexpected turns, but rather examining encoded structures, models, and formulations; explicating connotations and denotations of specific coding choices; and exploring traces of the code's development that are not apparent in the functioning of the software alone. As with all texts, the discovery and creation of meaning grow out of the act of reading and interpretation. As the product of creative processes carried out over time by human and machine collaborations, code offers details and foundations for those interpretations. Even very ordinary code designed to achieve some everyday purpose, some practical goal, to produce some output or process, carries with it meaning beyond what it does. I argue that computer source code since its inception has been a means of communication (and hence ideological), but that we are only beginning to discover the methods of explaining that ideology or, more broadly, that meaning.

When reading the examples of the protest sign and the Women on GitHub project code, I do not and cannot approach them in an ideologically neutral way. Instead, my reading is informed by feminist and postcolonial critical theories. Those theories attach valences to what might otherwise be framed as mere technology, a view which merely disguises its own ideological assumptions. My reading is influenced, for example, by the work of Roopika Risam, who in her book *New Digital Worlds* (2018) identifies a need for postcolonial digital archives structured on a more intersectional model. *Intersectional* here means drawing together multiple interconnecting aspects of identity. As Risam explains, "Within colonized and formerly colonized nations and for people outside of dominant cultures, access to the means of digital knowledge production is essential for reshaping the dynamics of cultural power and claiming the humanity that has been denied by the history of colonialism" (46). Risam and I are influenced by

Giyatri Spivak, whose seminal essay "Can the Subaltern Speak?" (1994) theorized how those without power, outside the cultural hegemony, cannot speak until they can represent themselves. Women programmers would not constitute a subaltern according to Spivak's definition because they have too much access to the tools of power. However, her theorizations of power and speech illuminate the dynamics at play in the significance of this code. In the global economy, female programmers born outside of first-world, Western countries, though part of a professional, educated class and though living elsewhere, such as Europe or the United States, have lower status and frequently lower pay than their male counterparts, especially those from more privileged economies, such as the United States. Arguably, by speaking in code, both the protest sign and the web page offer examples of women representing themselves. Although this is not the place for a full elaboration on Risam or Spivak's theories, this example offers one of the ways outside heuristics—in this case, postcolonial theory and intersectional approaches—can inform our code-reading practices.

I call the act of interpreting culture through computer source code *critical code studies*, and in this book I will attempt to characterize but not limit its methods primarily through a series of case studies. Critical code studies names a stance toward code as a unique semiotic form of discourse, the meaning of which requires specific techniques that are still being developed, even as the natures of code and programming are rapidly evolving. In other words, code is a unique expressive milieu that operates like, but is still distinct from, other forms of communication primarily due to its relation with hardware and other software systems.

Critical code studies names the methods and the scholarship of those involved in the analysis of the extrafunctional significance of source code, and this book offers a collection of case studies to demonstrate some of its potential.

The Origins of Critical Code Studies

Critical code studies grew out of a desire to read digital objects with more attention to their unique construction.

In 2004, when I was analyzing conversation agents, or chatbots, such as ELIZA, I was trying to find a way to perform what N. Katherine Hayles (Hayles and Burdick 2002) and Lev Manovich (2002) were calling *media-specific analysis*. What were the unique properties of conversation agents? What made analyzing them different from analyzing other digital or print objects? Soon it became obvious: the code. But how does one read code? In my search for answers, I found a few scholars who wrote about the ontology of computer source code. Kittler (1992) had written a bit. Lawrence Lessig (2006)

had approached code in a broader sense, bringing ideas from the legal world. Code had also been taken up by a handful of other scholars, specifically Adrian MacKenzie (2005, 2006), Florian Cramer (2005), Loss Pequeño Glazier (2006), Alan Liu (2004), and Alexander Galloway (2004). Their gestures were powerful opening movements into the realm of code, but they did not include many examples of actually interpreting the code. If code is, as these critics were suggesting, a unique semiotic realm, what could one say about any one passage of code?

I had been trained in literary theory and cultural studies, as well as what we were then calling *new media*, so I sought tools there. Semiotics offered tools for analyzing any sign system, and deconstruction complemented that study by poking around in the cracks and fissures. Cultural studies offered a way to take the text off its pedestal, while also helping to change the object of study from "text" as a set of characters to "text" as any cultural artifact. The critical theories aimed at underlying structures of oppression and possibility, from feminism to Marxism, queer theories to postcolonialism and theories of race and racial formation, also provided frameworks for critiques.

Around the same time, Noah Wardrip-Fruin, Matthew Fuller, and Lev Manovich were beginning to theorize software studies, while Nick Montfort and Ian Bogost were launching platform studies. What all these studies had in common was their emphasis on analyzing the particularities of different categories of technology. However, platform studies focused primarily on hardware, at least in its first outing, on the Atari 2600 in *Racing the Beam* (Montfort and Bogost 2009), and in *Expressive Processing*, the first book in the Software Studies series, Wardrip-Fruin (2009) essentially bracketed the code. Focusing on code could supplement these, so simultaneous with the birth of these two branches of new media studies, I proposed critical code studies in an essay (updated in chapter 2), which I presented at the Modern Language Association meeting and which was published in *Electronic Book Review* (*ebr*).

The goal of that essay is to instigate scholarship on methods of interpreting code. I argued (and continue to argue) that rather than bracketing the code, we should read it, beginning with the tools of semiotic and cultural analysis and continuing by developing new methods particularly suited to code. We had to get past the understanding of code as meaningless math (also, as it turns out, a false conception) and instead approach it as a culturally situated sign system full of denotations and connotations, rendered meaningful by its many and diverse readers, both human and machinic. Daunting though it seemed, the time had come to take code out of the black box and explore the way its complex and unique sign systems make meaning as it circulates through varied contexts. I was discovering in the process a richness that computer

scientists already knew, a sense of significance that grows out of and yet goes beyond the function of the code—only I had the additional benefit of heuristic tools developed in the humanities for interpretation and exegesis.

However, not everyone was so excited about this proposal. For example, some of the computer scientists who heard about the idea and read some of the early critical code studies writings (particularly my piece linking heteronormativity and malicious software worms) responded with derision and alarm (discussed in Marino 2016 and McPherson 2018). From its doubters' perspective, CCS marked another invasion of the humanists into what is known as "the science wars," a fierce contest between theoretical physicists and the humanists they felt were making much ado about insufficiently understood advances in science, specifically quantum physics. What would happen, they asked, when these literary scholars haphazardly applied their high theory to something so far outside their realm of expertise? The charges of imperialism and imperiousness were clear.

Not wanting to alienate the very community of experts whose works and whose realm I sought to explore, I spent the next few years in conversation with computer scientists. We convened online forums, five biannual Critical Code Studies Working Groups (CCSWGs), which included scholars of all levels and various backgrounds, especially computer scientists.[27] Out of those conversations came models for the interpretive practices demonstrated in this book, along with the basis of mutual respect born out of careful expressions and translations of our positions. As I say in the original essay, even the words *interpret* and *meaning* do not signify the same ideas in computer science as in the humanities. The group also included several scholars whose training bridged the gap between the humanities and computer science, further helping to cross the divide.

Born of those working groups were articles, conference presentations, and books, including the first CCS manuscript, the ten-authored *10 PRINT CHR$(205.5+RND(1)); : GOTO 10* (Montfort et al. 2013; aka *10 PRINT*), which analyzed the code of its title. Wendy Chun (Marino 2010a) presented what would be her first chapter of *Programmed Visions* (2011). Dennis Jerz (2011) offered the FORTRAN code of William Crowther's *Adventure* for collective annotation. Mark Sample's "Criminal Code: Procedural Logic and Rhetorical Excess in Videogames" (2013) offers a reading of C++ code in the video game *Micropolis*, the forerunner to *Sim City*.[28] Tara McPherson presented an early version of her analysis of the intersections of the civil rights movement and the development of Unix (2010; revised and extended in 2018). Federica Frabetti offered a take on a misread bug in the code of the first space probe, Mariner 1 (2010).

Subsequently, several books have expanded the cultural contexts for examination of code. David M. Berry's *The Philosophy of Software* (2011) takes up a formal consideration of the ontology and epistemology of code, as does Adrian Mackenzie's *Cutting Code* (2006). Bradley Dilger and Jeff Rice have collected reflections that center on HTML tokens in *A to <A>* (2010). As I've mentioned, *Speaking Code* (2012) by Geoff Cox and Alex McLean offers an exploration of the way code acts as speech in the form of a duet, with text by Cox intertwined with code passages by McLean. D. Fox Harrell's *Phantasmal Media* (2013) offers programming as a potentially disruptive culturally situated practice. James Brown Jr. applies rhetorical theory to code in *Ethical Programs: Hospitality and the Rhetorics of Software* (2015). In *Coding Literacy: How Computer Programming Is Changing Writing* (2017), Annette Vee analyzes code from a legal perspective and examines what the framework of literacy brings to larger conversations about teaching code in cultures. Finally, the most recent work, Kevin Brock's *Rhetorical Code Studies* (2019), offers a specific application of critical code studies that reads code as a means of rhetorical discourse.

A lot of the early work has been spent trying to understand what is possible to read and code and even what code itself is. Several scholars have found it useful to speak of code as *performative language*, drawing upon the speech act theories of J. L. Austin. Hayles, Cox and McLean (2013), and numerous others have found this framing to be productive because code seems to do what it says. However, it is clear that speech act theory is only partially applicable. Others, such as Daniel Punday (2015), have cast programming as writing. Still others, such as Roger Whitson (in the 2012 CCSWG), have suggested that actor-network theory might be usefully applied. In this formulation, code becomes an entity (an actor) as it networks together other bodies of code, machines, and humans. Certainly, such a theorization speaks better to Kirschenbaum's sense of the critical importance of context to interpreting code (see chapter 2).[29] If the study of natural language and semiotics is any indication, these initial theoretical frameworks are merely the beginning.

There are quite a few texts that take up the ontology of code. Chun's *Programmed Visions* (2011) and Hayles's *My Mother Was a Computer* (2005) offer important interventions that situate computation and our fascination with it. Gabriella Coleman's *Coding Freedom* (2012) takes up the ethos of the culture of open-source programming. Jussi Parikka's *Digital Contagions* (2007), although containing very little code, offers a robust media archaeology of viruses and worms. Because of their philosophical interventions into the study of technoculture, these works provide a foundation for readings of code.

Critical code studies has also had an effect on the production of code-based art projects. Artists, such as Nick Montfort and Stephanie Strickland (2013), have begun

to publish their source code (or even essays embedded in their source code) with an awareness that critical code studies scholars will be perusing their code later. Similarly, artists such as J. R. Carpenter (2011) and Brendan Howell (Griffiths et al. 2010) have published book versions of procedurally generated literary texts that have included their source code. The creators of the Transborder Immigrant Tool, discussed in chapter 3, have likewise published their code along with excerpts of the verbal poetry of the piece. As these artists foreground the code in the publication of their works, they invite readers to include an exploration of their code in the interpretation of the larger work.

This brief overview does not even begin to count the many books published in the field of computer science that are discussing code beyond functionality. One notable collection is *Beautiful Code* (Oram and Wilson 2007), which asked influential programmers to share essays on examples of code that were beautiful in their eyes. Opening the discussion about aesthetics, including multiple perspectives, and acknowledging the subjectivity of aesthetic claims is the starting point for recognizing code as a realm of discourse that deserves deep discussions that go beyond functionality and efficiency.

Over ten years after it began, the movement to develop critical code studies is well on its way as a field. Much of the initial pushback has dissipated because scholars have become more aware of the pitfalls and possibilities of this field. The idea of literary scholars or what we call *digital humanists* interpreting code is no longer novel but is now accepted and so can do what I hoped it would do—supplement other projects of cultural and media analysis. Take, for example, Anastasia Salter and John Murray's platform studies book on Flash (2014), which includes analysis of code from a representative piece of ActionScript. Software studies, platform studies, and media archaeology having dug their foundations can now work to strengthen one another rather than existing in unnecessary, balkanized fiefdoms.

As the field of critical code studies has expanded, so has the range of approaches. In a recent working group, Jessica Marie Johnson and Mark Anthony Neal, editors of and writers of the introduction to a special issue of *The Black Scholar* entitled "Black Code" (2017), led an exploration through the Afro-Louisiana History and Genealogy Database and the Trans-Atlantic Slave Trade Database, asking how the encoding of slave trade in manifests and in scholarly projects can either dehumanize or offer a means to reemphasize the humanity of victims of enslavement. By exploring databases and spreadsheets, this study also brought into focus one of the most widespread and yet understudied environs for programming, a reminder that the work of programming and coding is happening in places that may look nothing like what we expect as well

as noting that contemporary programming activities often have predigital origins that also need to be interrogated.

And yet the work is far from done. In fact, it is still very much in the early stages, which is why I am writing this book: not to offer a complete compendium of all CCS practices, but to share some of the initial reading approaches that I have found useful. I hope this collection of case studies will inspire others to create and discover what they can make of code.

E-Voting Software

To read and interpret code does not necessarily mean to discover secret content or to uncover its unconscious bias or even to reveal unusual functioning. Instead, to read code critically is to explore the significance of the specific symbolic structures of the code and their effects over time if and when they are executed (after being compiled, if necessary), within the cultural moment of their development and deployment. To read code in this way, one must establish its context and its functioning and then examine its symbols, structures, and processes, particularly the changes in state over the time of its execution. Reading these elements requires a philosophical heuristic, also referred to as *critical theory*, although that can be a contentious term. Suffice to say, all interpretation relies on a stated or unstated philosophical disposition, at the very least. For the third example, then, let us take code from an open-source voting system, specifically the free and open-source software (FOSS) Votebox system.[30] It is worth noting that when reading any new piece of code, even for a programmer who is not trying to perform an interpretation, it can be difficult to get a handhold. I offer this example not to give a definitive reading but to model some strategies for approaching any piece of code, and specifically code that constitutes software.

The first thing I look for when analyzing a piece of code is the context. Who wrote the code and why? In this case, its purpose is electronic voting. Electronic voting, or *e-voting*, has been a divisive topic of debate for the past two decades, dating back to problems with punch cards: the infamous "hanging chads" in the 2000 US presidential election (Conrad et al. 2009). E-voting at once offers a potential solution to ballot box stuffing and election rigging and an unprecedented opportunity for election hacking in times when the electronic distribution of our information seems increasingly vulnerable. In the decades after the panic over paper ballots, e-voting machines also became a source of fear. For example, in the 2016 US presidential election, the threat of international tampering loomed over these vulnerable systems. How do these historical contexts frame the software?

Next, I examine the general class of software. Voting software, like many types of software, for a time was mostly constituted of proprietary systems unavailable to the public. However, as questions mounted about the security of voting software and as the code of some software was leaked, this practice began to change. Some groups began to argue for the need for open-source voting software that could be readily examined by the public voting on these machines. Their release of free and open-source software led to the subsequent release of the code for proprietary software by companies in the commercial market. With that context briefly presented, I turn my attention to this specific piece of software: who created it, when, where, and why. This begins the historical, archaeological, and sociological research that will ground the reading. If the authors of the code are still alive, I may try to interview them.[31] Otherwise, their documentation is extremely valuable in this process.

Votebox presents one free and open-source, digital-recording electronic (DRE) voting software developed by researchers at Rice University. That the code was created for research purposes rather than other goals, such as hactivism, artistic reflection, or commerce, informs the way I read this code. Votebox is written in Java, although it originated in C#, in what the authors hope is a "clear, modern, object-oriented style" (Sandler, Derr, and Wallach 2007, 360). This emphasis on clarity may reflect the scholarly goals for creating this software or the desire for the code to be legible to a broader public for auditing purposes. In any event, the confines and affordances of the chosen language and the software design paradigm are key to interpreting any piece of code, and these programmers note that their choice of Java added length due to its sheer verbosity in comparison to Python (ibid., 360).

The comments of the code, particularly well-documented code, offer a guide to its functionality but often convey more. In his book *The Philosophy of Software*, Berry (2011) notices that the comments in the Votebox code always refer to the voter by using the male pronoun:

```
/**
* This is the event that happens when the voter requests
* to challenge his vote.
*
*    format: (challenge [nonce] [list-of-race-random pairs])
*
* * @author sgm2
*
* */
```

Berry reflects on the implications of these pronouns in the code (2011, 116). A quick search of the current release code turns up over seventy instances of *his* in the code and its documentation. Some have argued that comments are not a part of the code, but others see comments as situating and contextualizing code (Douglass 2010). Just as Berry reads these comments as implying a gender of the voter, another might argue that the code outside the comments does not offer any signs of gender. Later, Berry considers the affordances of the voting interface in contrast with a traditional ballot and the ways the systems bound or constrain certain aspects of the voting process.

How else might we approach this code? We could explore its cryptographic security in relation to contemporary conversations of voting insecurity. We could consider the readability of the code in a project built on increasing the transparency of the voting process. We could place its approach against other varieties of voting software that have since been released. What vision of democratic participation of voting does this code implement?

Consider the following passage from the VoteBox.java file (excerpts from lines 195–221):

```
currentDriver.getView().registerForChallenge(new Observer() {
   /**
     * Makes sure that the booth is in a correct state to cast a
ballot,
     * then announce the cast-ballot message (also increment
counters)
     */
     public void update(Observable arg0, Object arg1) {
         if (!connected)
            throw new RuntimeException(
               "Attempted to cast ballot when not connected to any
machines");
         if (!voting || currentDriver == null)
             throw new RuntimeException(
               "VoteBox attempted to cast ballot, but was not
currently voting");
         if (finishedVoting)
                  throw new RuntimeException(
                "This machine has already finished voting, but
attempted to vote again");
```

```
                finishedVoting = true;

                auditorium.announce(new ChallengeEvent(mySerial,
                            StringExpression.makeString(nonce),
                                BallotEncrypter.SINGLETON.
getRecentRandom()));

                    BallotEncrypter.SINGLETON.clear();
                }
            });
```

As the comments that accompany this code indicate, this section will "listen for challenges [sic] ui events. When received, discard the ballot (as the vote is no longer countable)/ ... and reply with the random key needed to decrypt this particular vote." If the three conditions are met (i.e., the machine is connected, voting was underway, and a vote had not already been cast), the program will note that voting is finished, then announce the challenge event to the auditorium before clearing the ballot. A challenge ballot will not be counted in the final election. If any of the three conditions are not met, which would indicate another problem with the voting software's processes, runtime exceptions are thrown in the form of error messages.

Within this passage of code, key constructs of the software appear. For example, Votebox uses a model of a persistent auditorium, a closed network separated from the internet by an "air gap," with which each device or instance of Votebox is constantly communicating to create transparency and redundancy of records of the voting (Sandler and Wallach 2007). Also, because any voter can challenge any vote, the software is always listening for a challenge event ("Votebox" 2008). We might consider how those constructs and their implementation attempt to intervene in contemporary concerns about paper balloting. How does the auditorium object implement the concept of transparency while providing redundancy in registered vote counts?

There are many further technical considerations to explore as well. How does the choice of Java impact the implementation? How does an object-oriented approach serve or confound the programmers' goals? How does this passage of code interact with the larger Votebox system and Votebox interact with the software and hardware on which it will run? How do the constructs in Votebox, such as the auditorium, supervisor console, and challenge event, instantiate a vision of elections in democracy?

In further media archaeology, we could also dig further into the context in which this code was created. How do the goals of the academic researchers who built the

code manifest themselves in the code itself—for example, the carefulness of the documentation? How do these goals differ from those of the creators of commercial voting software?

It is not my intention to limit these readings or even perform them here but instead to demonstrate how much more fruitful a reading can be when performed on real-world code, not merely hypothetical code or pseudocode. These questions offer models of ways to begin to interrogate and explore the code as a text.

Certainly, for every line of flight from one reading of the code, one could present an alternative reading or response. For example, consider Berry's remark about gender in the code. If code comments are essentially distinct from operational code, does the use of the male pronoun in the documentation have any real implications on the code itself? Are there other ways in which the voter/user has been gendered? Such ambiguity, such indeterminacy, such uncertainty may produce unease in more empirically minded positivists. However, uncertainty is fundamental to the search for meaning. Code may have unambiguous effects on software and the state of the machine, but the implications of those effects are anything but. Exploring and making meaning from symbols requires a letting go of the need for right answers, for that which is empirically verifiable.

From this example, a few general techniques suggest themselves. To explore code, a scholar should first read the code and its documentation to determine what the code does. If the scholar is not very familiar with the programming language or architecture, a guide with more fluency may assist. In rare occasions (including some in this book), a scholar can discuss the code with its authors, keeping an eye out for an intentional fallacy or the sense that the program's author can definitively say what the code means. Regardless of the scholar's experience with code, I recommend reading code with one or more others to discover a variety of perspectives. The meaning of symbols in communication is never an individual affair. Code's meaning is communal, subjective, opened through discussion and mutual inquiry, to be contested and questioned, requiring creativity and interdisciplinarity, enriched through the variety of its readers and their backgrounds, intellectually and culturally.

Code can be difficult to navigate even for its own author. To aid in this process, I find it useful to scan the overall architecture while reading comments if available. Guide and commentary texts (articles by the creators or critics of the code) can be priceless here. If possible, the scholar should see the code in action, executing it (once compiled, if needed), even using additional software to monitor state changes. It is also useful to explore the genre of the code to better identify typical techniques versus

innovations, aberrations, anomalies, or other notable variations, guided by the question, What's worth noting?

However, reading code does not mean staring at monochromatic character strings. Douglass (2011) early in the evolution of CCS suggested using a syntax highlighter or even an integrated development environment (IDE) to see the code the way developers do. IDEs also include advanced features for tracing variable states, arguments, and functions while providing a nonlinear means for exploring code. There are several freely available IDEs, such as Eclipse or Netbeans, which was used to develop the Transborder Immigrant Tool (chapter 3). Alternatively, code readers could use web applications such as Jupyter Notebooks, R Markdown, Apache Zeppelin, or Spark Notebook, which enable the creation of documents with running code in them so that code readers can more readily see the effects of code. Reading code is not like examining other texts because software contains assemblages of processes with changing states (Vee 2017). Nonetheless, this emphasis on reading more complex software does not exclude the reading of code-like artifacts through CCS.

Throughout my readings, I have found some basic questions to be useful: How does the implementation of the code (inside) reflect or contrast the functioning of the software (outside) or vice versa? How typical is this code within its more general class of software or algorithms? Where do its core ideas lie? What is anomalous about this code (for so much code is reused from similar code found elsewhere)? What methods or other sections of the code seem to capture the central idea of the software, its key contributions? (See the final chapter of this book for more of these initial questions.)

But the code is not enough in itself. It is crucial to explore context. Who wrote the code? When and why? In what language was the code written? What programming paradigm was used? Was it written for a particular platform (hardware or software)? How did the code change over time? What material or social constraints impacted the creation of this code? How does this code respond to the context in which it was created? How was the code received by others? Although many of these questions address the technical aspects of the code, I do not want to limit context to the material condition of the code itself. Other paratexts (what Mackenzie calls *non-code-like entities*) also impact the meaning of the code.[32] The *critical* in critical code studies encourages also exploring the social context through the entry point of code.

Asking these questions is not enough. Critical reading requires critical lenses. Examining the code from a critical perspective requires a system of analysis or critical hermeneutics as reading practices. To this point, most of the techniques and questions I have listed are not much different from those any programmer or technical analyst might use when analyzing a piece of code. However, critical code studies apply additional

lenses drawn from philosophy, cultural studies, media theory, and literary theory. Some of these lenses focus on aspects of identity, others on issues of power, and still others on ontology and signification. They are largely drawn from, but are not limited to, philosophy and semiotic analysis, then adapted to the specific attributes of source code. Theories that seem to apply mostly to social environs, such as queer theory and postcolonial theory, offer considerable insights into the technosocial realm. Critics may (and perhaps cannot help but) choose their hermeneutic before choosing the code that it helps interpret, or, more productively, they can follow a more emergent approach, seeing which hermeneutic the particular code and contexts warrant.

Although every piece of code has its own context, these foundational techniques have proved useful in my interpretations. I have applied them to even a single line of code (with nine other authors in Montfort et al. 2013) and found them useful tools in opening an exploration. To the observant, the universe can be seen in the line of code.

What Does It Mean to Interpret Code?

When I approach programmers about interpreting their code, a wry smile arises on their lips. After a bit of discussion, it becomes clear that they suspect I want to read their code as an English major would read a poem by Nikki Giovanni or a sonnet by Shakespeare. Will I treat their methods as stanzas? Their routines as rhymes about roses? Am I elevating their just-in-time or labored code to the status of art? Their smiles reflect their ambivalence about their own code. On the one hand, they tend to dislike their code and feel a certain degree of shame. Perhaps it can be said of code what da Vinci said of art: It is never finished, merely abandoned, meaning that code as a unit of mechanisms is always partial and potential. It can always be improved, developed, reworked, reimagined, and repurposed. On the other hand, programmers seem to feel a degree of bemusement that I would try to understand them through their code, treat them as I would an author, seek their fingerprints or their signature style. Will I celebrate them or use their code against them? But critical code studies is not some grand game of gotcha, nor some return to the harsh critiques programmers received at school, online, or at work. Nor does critical code studies primarily seek to understand and map the mind of the inspired computer programmer as artist. Some may pursue that tack, but the search for the genius of the programmer is not at the core of this project any more than it is the primary focus of archaeology or cultural studies. In fact, this misapprehension of the goals of critical code studies marks an instructive moment of miscommunication between humanities scholars and computer scientists about what we do.

Even in literary studies, at the start of the twenty-first century, interpretation is not that search for what the author secretly meant, that subjective hunt that computer programmers probably recall with dread from their English literature classes. Instead, interpretation is the systematic exploration of semiotic objects for understanding culture and systems of meaning. The subtle difference is that though many scholars still focus their attention on specific sets of authors, authorial intent and absolute meaning are not the ends of interpretation. Rather, more like the artifact examined in archaeology, the cultural object acts as an opening to a discussion of its significance within the culture that formed it. What aspect of culture and what realm of meaning (or *hermeneutic*) depends on the disposition of the scholar? The shift from the quest for the hidden meaning of the romantic (capital A) Author to the exploration of the object as a cultural text for exploring systems of meaning and culture is largely the result of the influence of semiotics and cultural studies, the field linked to the Birmingham school and Stuart Hall (see Hall 1992). Additionally, *semiotics*, the study of signs and signification, opens the interpretive process to all systems of information and meaning—what some might call *communication*, though that term is perhaps too strongly associated with another discipline. Cultural studies examines every object, every artifact, as a text for study. The distance between the haiku and the can of Coca-Cola as "texts" marks the shift from the study of artistry, on the one hand, to the broader study of signification and the manner in which objects acquire meaning on the other. Cultural studies scholars do not ask what the Coca-Cola Company intended by choosing red for the color of its cans and logo or what meaning it is hiding in the signature style of the words on the can. Instead, they perform a semiotic analysis on the possible meanings conveyed by those details of the can (the color red, the cursive script) and discuss what the can and, by extension, the company have come to represent.

In some ways, this type of analysis reverses the process by which this object came to be, as it was designed by artisans quite likely with feedback from other corporate executives in consultation with focus groups and consultants. At this point, one might note that the can is an object of commercial exchange. That too becomes part of the cultural analysis. Would the critic have any less to say about a cultural artifact that had a less purposeful marketing purpose, like a user's manual or a recipe in a cooking magazine? Probably not. Nonetheless, a reading of that text, an interpretation of that text, would require putting it in the context in which it is communicating: manuals, recipes, cooking culture. The cola can, in this case, is a text.

This is a useful moment to pause on this word, *text*. Because I am primarily a literary scholar with a background in analyzing poetry, prose, and plays, one might think that

I am implying that we read code the way we read poetry. But I am using that word not in the literary sense but in the much broader sense of cultural studies, situating the code as a cultural object of inquiry. Stuart Hall and others, for example, have written a book analyzing the Sony Walkman as a cultural text, examining its origins and impact on culture (Du Gay et al. 2013). Add to this works of semiotic analysis, such as Fredric Jameson's (1991) "reading" of the Biltmore Hotel as a text, and a more robust notion of *text* emerges, one that has subsequently influenced and expanded what is studied by literary scholars as well. The source code is a text, and to read it as a cultural object would be to consider it in all its history, uses, and interconnections and dependencies with other code, hardware, and cultures (Hall 1992).

However, to say that code is a cultural text is not to deny that it is the text of the code that specifically interests me. Most of the code examples I will be discussing here are also made of text—and by that, I mean written characters, numbers, letters, and symbols.[33] Although critical code studies examines code as it operates—whether compiled or not—in its executable form, CCS seeks to address a specific omission in contemporary scholarship on technology and technoculture. The text of code offers an opportunity for close analysis that often is too quickly abandoned by scholars in order to move into looser discussions of what software does or seems to do. Code should be read, and read with the kind of care and attention to detail that a programmer must use when debugging it, further augmented by the many reading strategies and heuristics that scholars have been developing for the interpretation of other kinds of texts and supplemented by new kinds of reading practices that speak more directly to the nature of code in its contexts.

So what is culture? Recently, I had a discussion with a computer scientist about critical code studies, and I raised this notion of coding culture. He mentioned the high number of Indian programmers working in this particular part of the country and suggested I speak with them about the way their cultural background appears in code. Although that inquiry would no doubt lead to some interesting exchanges, this suggestion revealed something to me: for him, *culture* largely signified *ethnic culture*. When humanities scholars evoke culture, they are typically referring to any social sphere, from particular workplaces to activities (e.g., skater culture, knitting culture), from regions of the world to virtual worlds (e.g., *World of Warcraft*), from realms of production and commerce to realms of scientific and academic inquiry. In programming, these cultures emerge around coding paradigms, languages, roles, and specializations, but also from an ethos or ideology, such as the FOSS community (see Coleman 2012). All these cultures, or subcultures, possess rituals, discourse conventions, meeting grounds (virtual or in real life), et cetera. They have shared texts, shared values and norms, shared

vocabularies, and shared tools. As a result, any artifact, object, or text offers a glimpse of the cultures in which it was produced and circulated.

CCS emerges as a close-reading practice at the very time when other scholars are advocating "distant" and "surface" reading. At this early stage in the interpretation of code, we still need to develop methods that make sense for and of code. Thus, though some are using software to perform computational analyses on large corpuses of texts, these early code readings look closely at smaller portions of code, closer to the readings of individual poems. However, this interpretation of code as a cultural text grows out of a very different analysis than the close examination of a lyric poem that seeks the answer to some riddle the reader believes to be hidden in the text (unless, as in chapter 7, that code was written to be poetry). The cultural text is not the work of art whose every aspect is admiringly explored but instead an object that is read with the archaeologist's attention to detail and meaning in context. That does not mean that this interpretive practice is somehow objective or that the practitioner is only the documentarian of the technology industry. Interpreting code requires the search for and creation of meaning tied to but not restricted to the intended purpose of the source code. Meaning is something on top of materiality, and its pursuit is deeply subjective, but that makes it no less valuable to the pursuit of understanding our world.

Meaning is not a straightforward process, and interpretation in the early twenty-first century has been radically altered by the advent of what Paul Ricoeur (2008; originally published in 1965) calls "the hermeneutics of suspicion" embodied in contemporary critical theory, specifically deconstruction and poststructuralism (356).[34] Although it is well beyond the scope of this book to detail the influence of these two enormous approaches to knowledge, it would be disingenuous for me to represent interpretation free from critical and creative interventions. This suspicion grows from the understanding that the arbitrary nature of language and symbolic representation and the difference (or *différance*) between the signifier and signified open the process of communication to a host of social and ideological influences.

Deconstruction and its network of critical theories, stretching from feminism to critical race studies to queer studies, to name just a few, open texts to understanding beyond their surface meaning, seeking out gaps and remainders. They read, if you will, between the lines. I am advocating for reading practices that explore exactly the very human levels of ambiguity and social contest of which computational machines are assumed to be but never have been free. In other words, with studies of technology and innovation, analyzing culture through code will include discussions of race and ethnicity, gender, sexuality, socioeconomic status, political representation, ideology, et cetera—all those messy human topics that complicate our social relations because

the walls of a computer do not remove code from the world but encode the world and human biases.[35] These skeptical and suspicious reading practices in concert with more traditional ones combine to make these critical approaches to code more than a documentation of its place in the world and instead a means of discussing that world. I call these approaches *critical code studies*. Critical code studies names the applications of hermeneutics to the interpretation of the extrafunctional significance of computer source code.

This book offers an introduction to critical code studies through demonstrations of some of its approaches, methods, and gestures. Chapter 2 contains an updated version of that original argument, although the manifesto spirit remains intact as a document meant to incite scholarly inquiry and debate. Over a decade has passed since the publication of the manifesto, and many of its ideas have been pursued by scholars, whether under the name of CCS or not. Others of its ideas continue to be contentious. The subsequent chapters offer case studies, readings of specific passages of code in their cultural contexts, to demonstrate some initial reading methods and explore what and how code means.

Chapter Overviews

Reading code is not like reading other objects. It has its own unique characteristics, specifically unambiguous consequences on other software and hardware. Because of the specific functional nature of code, that it is a symbolic construction in an artificial language embedded in a processing system, any reading depends on a precise and accurate understanding of the exact operations of that code. That is not to say that reading poetry or history requires less rigor, but instead that at least one part of this process is unambiguous, even if its meaning is open for debate. Also, due to its complexity, code can be difficult for human readers, even to those familiar with programming, to parse. Moreover, often it is part of software that can be more than a million lines long, interoperating with systems of similar length. Not to mention the fact that code also can run differently on different hardware and in different operating environs.

For these reasons, this book is written around a series of case studies of readings of relatively small and self-contained passages of code. Scholars of print or even digital texts typically can rely on the reader's familiarity with their object of study and merely excerpt as needed. A scholar of history can depend on familiarity with the general events in history. However, because I do not have that luxury, in order to make these readings as clear as possible I have chosen to explain how the code works through summaries and detailed annotations before giving my interpretation. Also, rather than

excerpting individual lines of code, I include a large portion of the code so that the reader can see some context and hopefully to encourage alternative readings of the same code. In at least one reading, my analysis of Kittler's code in chapter 6, I will remark on code beyond the passage excerpted at the start of the chapter. Not every code study requires this approach, but because code can be very difficult to follow and because this book is written for a range of readers, from experienced programmers to those with little experience programming, I want to make sure that everyone can access my readings to evaluate my interpretations and hopefully develop their own.

Thus, the first part of each case study chapter presents a code excerpt, explaining its context and its functioning in detail. The second part presents an exploration of the meaning of the code beyond and yet growing out of what that code does—the *extra-functional significance*, as I call it. Because of this book's wide target audience, to the extent possible, I will attempt to define terms, to explain programming structures, and to render the code legible for the uninitiated programmer. That said, this book cannot offer a comprehensive introduction to computer programming. Rather, it is my hope that the book will be intriguing enough to nonprogrammers to draw them deeper into the study of computers and programming languages, for programming is one of the key literacies of our age.[36]

Before embarking on those case studies, I offer in chapter 2 a revised version of the *Electronic Book Review* essay that launched this endeavor. Although I have reworked various parts of this essay, I have left large portions in their original state because this manifesto now plays a historical role in the emergence of this field. Therefore, some of the gestures and ideas in the case studies that follow go beyond what was envisioned in this initial call. Nonetheless, it contains both the spark and seed of what grew from it.

Chapter 3 offers an example of code written for a hacktavist art project, the constructs and comments of which inform, extend, and complicate the meaning of the larger work. This chapter examines code from the Transborder Immigrant Tool (TBT), a mobile phone application designed to help sustain travelers in the last mile of their journeys, migrating across a border between two nation-states by giving them directions to water and playing recorded poems full of survival advice. This code was written in Java (J2ME) by an artist collective with an eye toward those who would encounter the code as an art project. TBT also was brought to a secondary audience of politicians and political pundits, who reacted to their own imagined executions of the code. Following the collective's categorization of TBT as Mayan technology, I situate this code as a form of ritualized knowledge, prescriptions and provocations, written as procedures. The reader who traverses the code encounters an experiential passage through dramatic

potential scenarios. Reading TBT demonstrates the ways code can contain world simulations that present ideologies and counterideologies and how named methods and variables can inform the broader reading of the code. This chapter offers an example of code embedded with metaphors that extend its meaning.

Chapter 4 turns from meaning embedded into the code to meaning that is misread during the public circulation of code. In this chapter, I explore climate modeling software, which when leaked caused the public uproar known as the *Climategate scandal*. In that whirlwind of online outcry, an overheated internet of climate change deniers and others thought they had found the smoking gun, the code that proved climate scientists were manipulating data to deceive the public. As it turns out, the smoking gun was more like the discarded cigarette butt of a programmer creating a momentary patch in a visualization. This code is notable not for how it functioned on the computer but for its role in the larger debate about climate change. It is a clear example of the way the debate over the meaning of code is already a realm of public discourse. This chapter will examine the way code becomes a tool of political discourse when recontextualized in public conversations and how code taken out of context can lead to misunderstandings that can influence public debate.

Chapter 5 examines the role English plays in higher-level programming languages and its colonizing effect on programmers subject to learning and using it. For this examination, the chapter presents FLOW-MATIC, a predecessor of COBOL, which presents an English-like syntax. A team led by Grace Hopper created FLOW-MATIC to offer business managers a programming language less intimidating than some of the contemporary alternatives. This chapter discusses the dream of natural-language programming and how natural language enables and muddles critical readings of code. On the one hand, FLOW-MATIC demonstrates the way legibility and writability in programming adhere to different criteria than natural spoken and written languages. On the other hand, FLOW-MATIC offers a clear sign of how the linguistic culture of what some have called *global English/es* were built into programming languages. This chapter shows how code readings rely on more than recognizably "readable" elements because the function of language is fundamentally different in programming environs. In looking back on the work of a pioneering woman, this chapter also reflects on the way the gender divide has expanded over the ensuing years in professional programming cultures. At the end of the chapter, I offer several recent projects in which artists have developed alternatives to English-based languages in order to contest the colonizing effects of global English embedded in code.

Chapter 6 situates making code as a theoretical practice: it brings together the approaches of code, software, and platform studies, along with media archaeology, in

its consideration of the work of one of that field's most eminent theorists, Friedrich Kittler. This chapter explores the code of a computer graphics program called a *raytracer* written by Friedrich Kittler, the media theorist who provocatively wrote "There Is No Software" (1992).[37] Rather than a chapter-long gotcha, this case study examines the ways in which Kittler used the code to develop a mastery of C and assembly language, while tracing out the algorithms of the physical and mathematical formulas he would theorize. In this chapter, I argue that Kittler's work in programming languages illuminates, informs, and extends his theories.

Chapter 7 examines code written to generate poetry that its author also situates as poetry. This chapter examines Nick Montfort's Taroko Gorge and its progeny in the context of Montfort's desire to create code objects that inspire others to play with and revise them. Prior to making Taroko Gorge, Montfort had, with Andrew Stern, given a presentation in which he framed Joseph Weizenbaum's ELIZA as just such an inspirational program and a model for code-centered electronic literature. In this chapter, I analyze Taroko Gorge and its variants in light of ELIZA and its legacy to examine the ways these adapters reimagine and remix the poem. Although most code is not written to be poetry, Taroko Gorge invites its exploration as a literary object and thus offers an opportunity to read code as a literary text.

Finally, chapter 8 offers some thoughts about the future application of critical code studies, particularly in the academy, and outlines steps for commencing a critical code studies reading.

To those who look to critical code studies to make better programmers, I say that I hope these readings make programming better by enriching our understanding of what it means to communicate through code. This is not the first book of critical code studies (*10 PRINT* gets that distinction), nor will it be the last. Let the case studies that follow exemplify some initial methods of this approach to inspire deeper and richer explications of source code in the larger exploration of culture through digital artifacts.