CHAPTER 2

Rhetoric and the Algorithm

The algorithm is perhaps the concept most central to rhetorical code studies, and it is necessary to examine how algorithmic procedures are related to humanistic scholarship in general and to rhetoric in particular. This relationship can be demonstrated by tracing a path from the origins of the algorithm through its adoption from mathematics by computer science and engineering to its role in the critical work of humanities research. Following this brief history of the algorithm and its connection to humanistic work is an interrogation of how the algorithm plays an integral role in rhetorical activity. Such activity can be understood from a perspective that Hayles (2012) has referred to as "technogenetic," meaning that it identifies the interrelated codevelopment (or, for Hayles, coevolution) of human and technological entities (10). For rhetoricians, this means not *only* digital or electronic technologies but all apparatuses, broadly employed, for the purpose of making and communicating meaning as well as the "specific implications" Gillespie (2014) has identified "when we use algorithms to select what is most relevant from a corpus of data composed of traces of our activities, preferences, and expressions" (168). Algorithmic construction of meaning, the execution of a kind of "knowledge logic" (Gillespie 2014, 168), works to facilitate action in a variety of digital contexts, emerging from their predecessors in new and familiar ways.

From Algorithm to Algorithmic Culture

An algorithm is, in broad terms, a procedural framework for accomplishing a particular task. Understood simply, it is the description of a task-oriented procedure through its component operations (i.e., its steps). The algorithm's most common explicit disciplinary usage occurs in engineering, computer science, and mathematics, where the algorithm exists as

a procedure with a discrete number of tasks whose operations make use of clearly defined conditions that impact subsequent decisions within the procedure. For example, the algorithm for a web page to display a certain time of day is likely to rely on determining where, geographically, the IP address for a given user's computer exists: if it is on the Atlantic coast of the United States, the time's display may accordingly adjust to UTC-5, or five hours behind Greenwich Mean Time. An algorithm for washing one's clothes may involve a condition wherein the washer is cleaning a load of white or colored garments; if the former, the algorithm may involve a step regarding the inclusion of bleach into the mix whereas washing colors would not involve that step.

The algorithm as a concept has its origins in the mathematical writing of Abu Abdullah Mohammed ibn Musa al-Khwarizmi, a ninth-century Persian mathematician whose work is commonly considered to have served as the basis for today's algebra (*al jabr*). In fact, the word algorithm is also generally said to be a reference to al-Khwarizmi's name (Hillis 1998). As noted by Steiner (2012), however, the algorithm *as a concept* predates al-Khwarizmi's work, or formal mathematics in general, by several millennia. As Steiner observed, the algorithm—procedural activity—existed long before al-Khwarizmi explicitly described the algorithm as a concept involving procedure. Even though it was not necessarily defined as a clear concept until al-Khwarizmi established it, Steiner argued, the algorithm has played a number of important roles in daily or common cultural activities for millennia:

> The Babylonians employed algorithms in matters of law; ancient teachers of Latin checked grammar using algorithms; doctors have long relied on algorithms to assign prognoses; and countless numbers of men [. . .] have used them in an attempt to predict the future. (Steiner 2012, 54)

Although the idea of algorithmic procedure has been a part of human culture and behavior long before the ninth century CE, it is through al-Khwarizmi's writing that the algorithm becomes codified as a procedural framework whose functionality is articulated through a specific grammar; specifically for al-Khwarizmi and his work, this grammar would later come to be called algebra.

For al-Khwarizmi and for mathematicians since, the algorithm was the procedural framework through which a mathematical equation would be

calculated. By constructing a framework to which a mathematician could adhere in order to solve discrete problems, the capabilities of symbolic systems to reflect logical procedures were suddenly clearly articulated. One example of al-Khwarizmi's algebraic algorithm in action demonstrated how a mathematician could determine the value of a particular squared number: "[if 'f]ive squares are equal to eighty;' then one square is equal to one-fifth of eighty, which is sixteen" (1831, 7). In mathematical notation, this equation can be demonstrated in the following steps:

Step 1: $5x^2 = 80$
Step 2: $x^2 = 80 * 1/5$
Step 2a: $x^2 = 80/5$
Step 3: $x^2 = 16$

The algorithm can be extended further to determine the value of the square root:

Step 4: $\sqrt{x^2} = \sqrt{16}$
Step 5: $x = 4$

The procedure to determine the value of x^2 involves condensing as many relevant operations of the equation as possible so as to calculate quickly and accurately the numerical value of x^2. While the symbolic mathematical notation by which algorithms could be most efficiently expressed was not developed until several centuries after al-Khwarizmi, the potential of algorithmic power for analytical and utilitarian employment had been clearly established.

This power has become most obviously demonstrated through the application of algorithms for computational ends, thanks to the rise of computers and the scientific and engineering disciplines dedicated to their study and development. In the mid-nineteenth century, Ada Lovelace would lay out a vision for the potential of computers to operate by means of programmed (algorithmic) instructions:

> A new, a vast, and a powerful language is developed for the future use of analysis, in which to wield its truths so that these may become of more speedy and accurate practical application for the purposes of mankind than the means hitherto in our possession have rendered possible. Thus not only the mental and the material, but the theoreti-

> cal and the practical in the mathematical world, are brought into more intimate and effective connection with each other. (2002, 19)

Lovelace's description of a possible language through which to manipulate computer technologies (and specifically Charles Babbage's "Analytical Engine") refers implicitly to algorithmic procedure for the sake of mathematical analysis. What had not yet developed at the time, but which would emerge just after her death, was a clear logic to drive algorithmic grammars toward practical ends.

Modern-day computers operate on a form of logic known as Boolean logic, after the nineteenth-century logician George Boole, who attempted to replicate the patterns of human thought through the logic of mathematical algorithms (Hillis 1998). There are only a few fundamental concepts that drive Boolean logic, the most notable being the binary states of "true" and "false" (or other arbitrary comparison states, e.g., "on" and "off," or "1" and "0"). Shannon (1937) demonstrated how electrical circuits, by being opened or closed, could serve as a viable application of Boolean logic. Shannon's work was used as the basis for programming machines to perform mathematical calculations, which in turn set the stage for the development of current computer technologies.

By checking the state of one or more given data elements within a computational system—what is referred to as "input"—a Boolean algorithm can allow a software program to execute particular computational tasks so as to express a relevant "output" body of data. Hillis (1998) has described algorithmic procedure and the computation it enables as being "all about performing tasks that seem to be complex (like winning a game of tic-tac-toe) by breaking them down into simple operations (like closing a switch)" (4). In other words, computer science makes significant use of Boolean-powered algorithms in part because algorithms, especially procedures that can be *automated* by a computational system, align effectively with the Boolean foundation upon which computers and electronic data work.

Due in no small part to the increasing ubiquity and status of computer technology in contemporary society, the algorithm has become a significant concept for a wide range of popular culture as well as for science and mathematics. Berlinski's (2000) *The Advent of the Algorithm*, MacCormick's (2011) *Nine Algorithms That Changed the Future*, and Steiner's (2012) *Automate This: How Algorithms Came to Rule Our World* all explicitly identified the algorithm as a paradigm-shifting phenomenon whose importance has had

world-changing effects. For Steiner, the increasing control that complex computer algorithms possess in contemporary culture is critically significant, as those who can create, understand, and manipulate complex algorithms with digital technologies have arguably become a new ruling class (an argument that has been taken up by other critics, such as Rushkoff 2011). Across these texts, there is a shared central premise that algorithms, especially those meant to be expressed via computer technology, are quickly gaining—or already possess—a prominent role at the heart of the networks and systems that power society. This prominence extends far beyond the significance of the culturally relevant algorithms that have persisted for centuries (e.g., those used in medicine, law, etc.). It may be accurate to say that to be aware of algorithmic procedure or to be able to work *with* algorithms is to be able to influence the trajectory of social, cultural, and political development to extents far beyond those phenomena that can be influenced by individuals who are unaware or uninvolved with algorithmic procedures.

While the vast majority of academic and professional discourse related to algorithms—which has overwhelmingly taken place in the disciplinary spheres of mathematics, computer science, and engineering—has focused on computational algorithms, the conceptual constraints surrounding algorithmic procedures tend to vary from discipline to discipline; the specific field in which a scholar or practitioner works has some influence upon how the scholar is likely to approach algorithms and their potential for certain tasks. This understanding is important as the particular language used by a scholar or practitioner to describe and explain what an algorithm *is* and *does* illuminates the recognized potential(s) that its author, or his or her disciplinary community, attributes to algorithmic procedure.

For example, computer science tends to base its definition(s) of the algorithm on the logic of the precise and discrete mathematical models that serve as the foundation for the discipline. Brassard and Bratley (1996) defined an algorithm as "a set of rules for carrying out some calculation, either by hand or, more usually, on a machine" (1). For Hillis (1998), it is "a fail-safe procedure guaranteed to achieve a specific goal" (78). Black (2007) defined the term as "a computable set of steps to achieve a desired result" (n.p.). Clearly, there is a conventional understanding of the algorithm as relating to replicable procedures made up of discrete operations to be executed through a computer or with its assistance. But not all scholars in the field describe algorithms in such discipline-specific terms. Edmonds (2008), for example, has stated that

> [a]n *algorithm* is a step-by-step procedure which, starting with an input instance, produces a suitable output. It is described at the level of detail and abstraction best suited to the human audience that must understand it. In contrast, *code* is an implementation of an algorithm that can be executed by a computer. (1)

Edmonds is quick to separate algorithm as concept from the code-centric applications of algorithmic procedure for computer science (e.g., software), organizing the latter within the hierarchy of the former; as a result, through the expanded definition of the term, the possibilities of the algorithm beyond the scope of computer science become much more situational and flexible than might otherwise be conventionally assumed.

Other fields with less direct foundation in mathematics often more readily accept the algorithm to possess these more fluid qualities than do those fields relevant to computer science. Accordingly, the means by which algorithms are approached and used for ends in these other disciplines are quite different as well. Ramsay (2011) described the algorithm as a concept in relatively flexible terms as "a method for solving some problem" (18). This definition has some significant overlap with that of *heuristic*, generally defined as a broad framework for problem solving and which, in rhetoric, is tied closely to the canon of invention through its emphasis on discovery (Herrick 2016, 27). While such overlap is not inherently problematic, the less "clear" a problem-solving method becomes, the more difficult it may be to reach consensus on whether that method performs in either an algorithmic or heuristic sense.

Gillespie (2016), meanwhile, has identified a range of context-specific metaphors for algorithm and how its understanding is communicated: trick, synecdoche, talisman, and commitment to procedure. Each of these metaphorical associations, Gillespie suggested, indicate both an identification of a particular audience (one that is expected to grasp and accept said metaphor) as well as of the individuals and communities employing those metaphors, since they do so for specific purposes in order to induce their audience(s) to some relevant action. For Gillespie (2016), this employment is "discursive work [that] the term performs" while rhetors and audiences are forced to navigate its multiple meanings (18).

Each of these cases brings to mind Steiner's (2012) overview of the algorithm as procedure relevant beyond computer science to law, medicine, grammar, and other diverse efforts toward predicting the future. Through these definitions of algorithm as procedure intended *for purposefully solving*

problems, nonscientific disciplines bring to the conversation an emphasis on how humans approach the accomplishment of particular tasks (and not necessarily what those approaches will be in any given situation). In order to clarify what the significance of this conceptual shift means for the rhetorical study of algorithms, I will situate the applications of algorithmic procedure across fields within the humanities.

Algorithmic Criticism in the Humanities

An inclusive understanding of algorithms as problem-solving procedures certainly incorporates into its scope the computational algorithms that drive electronic technologies and computer software. But humanistic inquiry relating to algorithms focuses on how a particular algorithmic procedure reflects the goals and values of its developer(s) and on the means by which computational procedures facilitate novel approaches to critical engagement and meaning making rather than focusing on the technical expertise that conventionally accompanies particular forms and applications of computation. Such inquiry explores both the complex situations that algorithms impact *and* the situations in which certain algorithms are composed, including how those compositions are structured in order to make a particular engagement. For rhetoric, this combination involves multiple scales of rhetorical activity, from the exigences that spur the creation of a given body of code to the specific devices used to frame and describe any response(s) to those exigences.

Even though my focus here will ultimately consider how algorithms are useful and significant components of persuasion, especially in regards to contemporary rhetorical activities, I first want to outline how algorithmic procedure is used in multiple fields within the humanities. The relationship between algorithm and humanistic production has existed for millennia, from the classical *enthymeme* to the more recent phenomenon of digitally mediated manipulations of massive data sets.

Humanistic Algorithms before and without Computers

The idea of procedure—whether explicitly connected to "algorithm" or not—as a means of generating or facilitating action has accompanied creative and critical practices since early uses of mnemonic devices to recite oral poetry; as Ong (2002) has noted, "In a primarily oral culture, to solve effectively the problem of retaining and retrieving carefully articulated

thought, you have to do your thinking in mnemonic patterns, shaped for ready oral recurrence" (34). As I explore below, the application of diverse algorithms for humanistic, and especially rhetorical, purposes remains a significant component of productive and critical activity, activity that reflects Nowviskie's (2014) description of algorithms employed for humanistic activity that "can be understood as problem solving and [. . .] as open, participatory, explorative devices" (151). The identification of an algorithmic humanities is integral to understanding how algorithms work for persuasive ends, a necessary requisite for examining algorithms and code as rhetorical communication.

ENTHYMEME AS ALGORITHM

As I have argued elsewhere (Brock 2014), the algorithm most central to Western rhetoric is the *enthymeme*, a concept that has predominantly been defined as an incomplete logical argument that, through its procedural logic and presentation, compels an audience to complete the argument in order for it to be properly and effectively expressed (Bitzer 1959; Walker 1994). Specifically, the enthymeme is a rhetorically oriented *syllogism*, a set of premises (major and minor) that, in combination, lead to some sort of conclusion or result. For a complete syllogism, the relationship between these elements is explicitly stated (see below). For an enthymeme, this relationship is *probable* and implicit, as its logic remains procedurally suspended until audiences identify it on their own. As Hairston (1986) has argued, "The person who argues from an enthymeme is [usually] not trying to *prove* a proposition; he or she is only trying to establish high probability" for an audience to accept the rhetor's proposition (76). The most well-known syllogism includes the following components:

1. All humans are mortal. (Major premise)
2. Socrates is a human. (Minor premise)
3. Socrates is mortal. (Conclusion)

This syllogism works categorically, meaning that it defines Socrates based on the categorical descriptions into which he fits, according to the statements' parameters (all *A* are *B*; *C* is *A*; therefore, *C* is *B*). More complex syllogisms can be constructed to create disjunctive or conditional reasoning. For example, the following is a disjunctive syllogism:

1. We will meet either in Paul's office or in the conference room. (Major premise)

2. We are not meeting in the conference room. (Minor premise)
3. Therefore, we are meeting in Paul's office. (Conclusion)

Conditional syllogisms generally include a determination of a condition being met. For example, one might say, "If it is nighttime, then one needs to drive with one's car's headlights on" as one of its premises. These distinctions in how syllogisms can be constructed are crucial for algorithmic logic, since the variety of arguments made possible through categorical, conditional, and disjunctive reasoning *all* work in varied ways to dramatically increase the flexibility with which one could frame the argument for a particular case logically and rhetorically.

A syllogism may be chained together with other syllogisms to create a *polysyllogism*, a more complex and nuanced line of reasoning than any of its component logics might establish on its own. Carroll (1973) demonstrated a number of polysyllogisms as puzzles to be solved (what a rhetorician might reframe as "enthymemes to be completed") in his classic *Symbolic Logic*. One such example is presented here:

(1) All writers, who understand human nature, are clever;
(2) No one is a true poet unless he can stir the hearts of men;
(3) Shakespeare wrote "Hamlet";
(4) No writer, who does not understand human nature, can stir the hearts of men;
(5) None but a true poet could have written "Hamlet." (Carroll 1973, 170)

To complete the polysyllogism, one would ultimately need to reason that Shakespeare was clever. More fully, the deduction would recognize that a true poet is clever, and Shakespeare is argued here to be a true poet. Specifically, Shakespeare is a true poet [statement 5] (as he wrote *Hamlet* [statement 3]). As a true poet, he can stir the hearts of men [statement 2], which means he is a writer who understands human nature [statement 4]. Because he is such a writer, he is clever [statement 1]. The deductive process for Carroll's polysyllogistic example is less direct or linear than a simpler syllogism, but its computational nature is indisputable. One *cannot* move forward with any of these ideas until their relations have been appropriately sorted or processed. Indeed, many contemporary algorithms overwhelmingly rely on complex polysyllogistic reasoning to compute data dynamically in numerous iterations.

In contrast to a fully articulated syllogism, the enthymeme functions

algorithmically by leaving implicit reference(s) to one or more of the syllogism's components so that an audience will compute the logic of those missing components. Walker (1994) has hinted at the enthymeme as a kind of algorithm in his description of the enthymeme as the center of "argumentative or suasory procedure" (61). Similarly, Walton (2001) argued that the enthymeme suggests a "plausibilistic script-based reasoning" commonly explored in artificial intelligence research (93). One such example of the enthymematic algorithm is the following statement that recalls an example syllogism presented earlier in this chapter: *all humans are mortal; thus, Socrates is mortal*. This statement obliges the reader to make an internal logical leap in order to discern that Socrates is mortal *because he is a human*. The "Socrates" syllogism could be rephrased by using any two of its three components; for example, the enthymeme might be stated as follows: Socrates is a human and therefore is mortal. The reader's ability to follow this argument hinges on recognizing (i.e., processing) the implicit association that Socrates' mortality is dependent on his humanity *because all humans are mortal*. In other words, the enthymeme provides an opportunity for a rhetor to directly engage an audience in the expression of an algorithm for rhetorical ends.

Specifically, the computational operation—the completion of the syllogism—is constructed in such a way as to convince the audience how best to calculate the remaining variables. The audience "becomes" a computer in order to express this procedure by reaching the (likely anticipated) outcome. For developers, the enthymematic analogy can be extended to *types* of procedures in order to suggest how to solve other sorts of computation in similar manners. In this sense, the algorithm works beyond the constrained sense of "a method for solving some problem" (Ramsay 2011, 18) and instead demonstrates its fundamental flexibility and contingent nature as a process through which an audience is led to persuade itself to achieve action, via such means as deliberation and conditional deduction.

One trend that aids the emergence of rhetorical code studies is the relatively recent exploration by several rhetoric scholars of the bounds of enthymematic persuasion. Specifically, these scholars have examined whether the enthymeme can function as a rhetorical tool with value beyond the scope of conventional discourse. For instance, Smith (2007) demonstrated how visual arguments make use of enthymematic principles of probability and implicit syllogistic completion in order to persuade viewing audiences. Walton (2001) identified the enthymeme as an integral component of artificial intelligence research, tying together technological

(and technologically mediated) modes of "thought" (i.e., reasoning) and communication.

Conceptually speaking, at the heart of the enthymeme is a recognition of computationally algorithmic procedures as inherently interrelated with this central form of rhetorical reasoning. The outcome for an enthymematic algorithm is expressed by a collaborative effort on the part of both rhetor, who initially provides an enthymeme as part of his or her argument, and audience, who completes its logic as part of an engagement with that argument. While algorithms in technological contexts may initially seem less enthymematic than mechanical, they nonetheless require the contingent interpretation of input to expression in order for its subsequent action to successfully communicate its meaning.

THE ALGORITHMIC RHETORICAL SITUATION-ECOLOGY

Another of the most significant algorithmic frameworks related to the study of rhetoric is that of the "rhetorical situation" (Bitzer 1968; Vatz 1968) and its more complex ecological rearticulations (Cooper 1986; Edbauer 2005). The rhetorical situation involves several interrelated components that together facilitate an effective rhetorical activity. First, a rhetor identifies a relevant *exigence* they wish to engage. An exigence is commonly defined as "an imperfection marked by urgency [. . .] something waiting to be done" (Bitzer 1968, 6). It is the catalyst for rhetorical, and any subsequent, action. Further, an exigence requires the capacity for change to occur; a problem that cannot be avoided or resolved exists outside the bounds of rhetorical intervention. It is only once this initial variable is established, whether understood as "recognized" or "invented," that the algorithmic quality of the rhetorical situation can begin to be processed.

The second situational component is *audience*, the body of individuals that a rhetor hopes to induce to action, a body whose members "are capable of being influenced by discourse and of being mediators of change" (Bitzer 1968, 8). An identification of audience includes a recognition of the audience's values, background, and ideological leanings as well as of the modes of communication most likely to persuade the intended audience through their employment. For code, as with other forms of communication, such an evaluation relies upon recognizing when and how a particular approach could be appropriately effective, not whether that approach it is the "objectively" most superior means of influencing an audience.

The third component is *constraint*, the limitations and influences exerted upon rhetor and audience alike as part of the expression of a rhe-

torical activity. For Jasinski (2001), constraints "are circumstances that interfere with, or get in the way of, an advocate's ability to respond to an exigence" (516). Rhetorical constraints include all restrictions upon a rhetor in how they might induce an audience to act, from modes of communication to particular suasive strategies or even specific language decisions.

Once an exigence has spurred a rhetor to act, and once that rhetor has identified an audience and the set of constraints framing the audience's potential reception of the provided argument, what comes next? The rhetorical situation's algorithm is expressed: the rhetor makes use of their chosen variables to bring about the intended outcome, that is, the change sought in regards to the spurring exigence. Jasinski (2001) has suggested that even the *definition* of the relevant situation is itself a sort of exigence to be transformed through its identification. The range of possible responses to a given rhetorical act is contingent on the relevant exigence, audience, and constraints, and an audience may not always respond in the same way to a given rhetor or argument.

It is in this space for chaos, for unpredictable or unanticipatable responses, where the algorithmic character of the rhetorical situation becomes most intriguing. The rhetor anticipates how each situational variable *might* (or is *likely* to) influence the others, but this anticipation ultimately remains one influence of many upon the situation. It is only when the rhetor attempts to *express* their argument through the situational algorithm that any action, whether intended or not, can be achieved. Whether consciously or otherwise, a rhetor computes the procedure that emerges when *enough* of the situational variables have been identified to assemble an argument in pursuit of a particular goal.

Edbauer (2005) has observed that the rhetorical situation, as defined by both Bitzer (1968) and Vatz (1968), is problematic in its reduction of rhetorical activity to a single equation of exigence + audience + constraint + rhetor, since numerous situations interrelate at any time in a larger rhetorical *ecology* of diverse actors, motivations, and exigences. Edbauer's (2005) critique is significant for rhetorical code studies in that it draws greater attention to the complexity of the connected algorithmic procedures that make up acts of rhetorical communication. In other words, while it is important to recognize how rhetor, audience, constraint, and exigence influence one another in a bounded, individual situation, it is just as important to acknowledge that these dynamics are themselves components of an

even greater ecologically rhetorical algorithm affecting broader discursive trends across various agents and channels of persuasive potential.

Recognizing the flexibility of the rhetorical situation-ecology is integral to an understanding of code-based action as rhetorical. Specifically, it points to the indeterminacy between specific computational operations and the *potential* for those operations to facilitate ranges of activity in what they do (i.e., how computation results in subsequent action), how they are constructed, and what they suggest about inventing similar operations for *other* computational purposes. While the particular steps of a given procedure are generally thought of as discrete, the possibilities they suggest—and the situational concerns they address and create—are inherently complex and indeterminate, and discussions of the rhetorical qualities of procedure necessitate this recognition.

AESTHETIC AND POETIC ALGORITHMS: THE OULIPO

While I ultimately lead us toward a discussion of algorithms for rhetorical purposes specifically, I first want to discuss briefly some creative (aesthetic and poetic) approaches to algorithmic scholarship and production that have gained traction in other fields within the humanities. For practices of literary composition as procedure, the group of mathematicians and writers in the mid-twentieth century, who gathered under by the name Oulipo (an acronym for the French name Ou*vroir de* Lit*térature* Po*tentielle*, or "Workshop for Potential Literature"), may provide the clearest and most thorough insight on how creative works may be generated from the constraints of algorithmic structures. The members of the Oulipo recognized that "[m]athematics—particularly the abstract structures of contemporary mathematics—proposes thousands of possibilities for exploration, both algebraically (recourse to new laws of composition) and topologically (considerations of textual contiguity, openness and closure)" (Le Lionnais 2007, 27). That is, the Oulipo acknowledged the possibilities of flexible and contingent meaning making within the framework of mathematical computation, specifically regarding how computation could be used to influence the composition of rhetorical texts. As noted by Queneau (2007), the objective of the Oulipo was "[t]o propose new 'structures' to writers, mathematical in nature, or to invent new artificial or mechanical procedures that will contribute to literary activity: props for inspiration as it were, or rather, in a way, aids for creativity" (51). The composition of *structures*, complete with rules and constraints for successful invention within

those structures, is significant for this discussion because it emphasizes the possibilities of procedural expressions rather than the skill with which a particular expression is created in reflection of the nuances surrounding a specific situation.

The explicit play with algorithmic procedure by the Oulipo was embraced because, according to Bénabou (2007), "[i]f one grants that all writing [. . .] has its autonomy, its coherence, it must be admitted that writing under constraint is superior to other forms insofar as it freely furnishes its own code" (41). It is not so much that writing within a set of constraints produces texts of a higher quality, but that a text *recognized to be* written under constraint(s) was considered superior by the Oulipo because its author(s) and readers alike could appreciate how its algorithmic procedures were expressed in order to produce the text "output." This is not to suggest that the limits of procedural constraint (the amount and range of expressions that could be produced) are necessarily *restrictive*. For example, Queneau's *Cent Mille Millardes de poèmes* ("One hundred thousand billion poems"), a collection of ten sonnets that all share the same rhyme scheme, offers the reader 10^{14}, or 100,000,000,000,000, potential poems that could be constructed by the reader swapping in or out a given line from one of the sonnets with the appropriate line from another, e.g. for sonnets A–J, one could use line 1 from sonnet A, line 2 from sonnet D, line 3 from sonnet F, line 4 from sonnet A, etc. (Berge 2007, 118–21).

While the members of the Oulipo explored these structures to demonstrate the possibilities of invention through procedural constraint, it is also accurate to say that they highlighted the existing means of rhetorical invention and arrangement and offered new insight into these means by drawing attention to the procedural structures that underpin decisions surrounding particular rhetorical situations. To repeat Ramsay's (2011) definition from earlier in this chapter, the algorithm is complicated far beyond its general definition as a "method for solving some problem" (18). In particular, Oulipian algorithms function rhetorically in that they emphasize the *potential* for constructing meaning through particular literary structures of constrained computation. This potential has even begun to be explored in depth with software code languages; Lopes (2014) built on Queneau's (2009) *Exercises in Style* in order to demonstrate stylistic influence on programming activities by following Queneau's example of writing the same vignette through the lenses of ninety-nine different literary styles. Lopes wrote the "same" program, a means of analyzing term frequency, in more than thirty different variations, based on the fundamental

stylistic concerns that guide each approach. As Lopes noted, "By honoring different constraints, we can write a variety of programs that are virtually identical in terms of *what* they do, but that are radically different in terms of *how* they do it" (2014, xii). Rhetoricians interested in the complexity of diverse possibilities that emerge from constrained situations—whether digital in nature or not—can take a great deal from the Oulipo's experiments with algorithmic approaches to invention.

Humanistic Algorithms in the Age of Computers

While algorithmic procedure and humanistic activity have been intertwined for much of human history, it is also true that computer technology has radically transformed this relationship. As a result, algorithm has, through code, developed beyond serving as a means for performing traditional rhetorical or critical action through the construction and execution of procedural activities. Specifically, it has become a form of meaning making in its own right. For computers, the algorithmic code that makes up digital software is rhetorically powerful thanks to its ability to engage data, machine, and human alike, albeit in different ways and for different ends. For most scholars, rhetorical engagement with and in code serves primarily as a tool to facilitate other iterative experiences, but when we recognize code as an algorithmic mode of communication, used specifically for rhetorical ends, we can approach a moment of clarity in which we reconsider how code influences us—humans and nonhuman technologies—to act through a quality that can best be described as algorithmic persuasion.

ALGORITHMIC CRITICISM

Possibly the most explicit use of algorithmic procedure to facilitate humanities scholarship, both conventional and unconventional, is Ramsay's (2011) concept of "algorithmic criticism." For Ramsay, algorithms provide avenues for literary research and interpretation through their expressive transformations of texts into novel "paratexts" that reveal insights that are otherwise unavailable or difficult to recognize. Paratexts, for Ramsay and other literary scholars interested in algorithmic criticism, are texts transformed, deformed, and performed in innovative ways and for various ends through procedural mutation and reconfiguration; Ramsay argued that "[t]he critic who puts forth a 'reading' puts forth not the text, but a new text in which the data has been paraphrased, elaborated, selected, trun-

cated, and transduced" (2011, 16). Among the means by which Ramsay demonstrated the literary potential for algorithmic procedure is the aggregation and computation of particular types of data in order to restructure the literary reading experience.

For example, Ramsay compared the most frequently used terms and ideas provided by the major characters in Woolf's *The Waves*, demonstrating how the commonalities between characters' most frequent terms allow readers to draw new connections between those characters. Included in table 2.1 (Ramsay 2011, 13) is an excerpt from the expressed set of frequently used terms for two of the novel's six protagonists using a relevant algorithm employed by Ramsay. Ramsay's line of inquiry sought out the terms that were not only most frequently used *but also* were not used frequently by any other character, that is, the terms that were either unique to, or at least primarily associated with, each protagonist.

One conclusion related to new connections that Ramsay draws is that although the character Louis (as the only Australian in the group) is self-conscious of his accent, the other characters seem to pay that cultural distinction little attention: no one else, for instance, ever mentions the terms "Australian" or "accent." To the others, these concepts seemingly do not matter since they do not appear among the terms used frequently by those characters (Ramsay 2011, 12–14).

This sort of reading by Ramsay—in which previously unrecognized

Table 2.1. Excerpted lists of term frequency from Woolf's *The Waves*, compiled by Ramsay (2011)

Louis	Neville
mr	catullus
western	doomed
nile	immitigable
australian	papers
beast	bookcase
grained	bored
thou	camel
wilt	detect
pitchers	expose
steel	hubbub
attempt	incredible
average	lack
clerks	loads
disorder	mallet
accent	marvel

meaning is exposed through expressions of algorithmic arguments—demonstrates not only a new form of *reading* but also a new form of *research*: the development of a body of paratexts reflecting the logics of computational approaches to critical investigation and interpretation. In other words, the emphasis for algorithmic criticism should be placed as much on how relevant critical work occurs, and what happens in the process, as on what algorithmic criticism reveals through its individual, iterative expressions.

Ramsay's approach to criticism clearly identifies algorithmic procedure as a valuable tool to be used for scholarly criticism. Specifically, the use of algorithms to explore novel means of reading allowed Ramsay to generate paratexts that transform a given text or set of texts into new objects of study. When employed in this manner, the algorithm is a method for interpretation that, in part, quantifies those components of an original text that a critic has deemed potentially significant, which is the text as transformed into a paratext. Further, the algorithm itself becomes an argument for interpreting a text through a particular interpretive lens (i.e., the parameters of the algorithm itself), drawing connections that may or may not be clearly "present" or important in a traditional reading of the text. The significance of Ramsay's algorithmic criticism is that it affords scholars a new way of engaging texts for humanistic ends by algorithmically discovering meaning within a text. That said, the constitution of algorithmically transduced literature generated through the deformation of an initial text does not interfere with conventional scholarly work: the algorithmic critic produces his or her own text (the algorithmic paratext) to be interpreted in addition to the original text.

CRITICAL CODE STUDIES: ALGORITHM AS COMMUNICATION

In contrast to Ramsay's work with algorithms *as a tool or lens* for criticism is the main body of scholarship related to critical code studies, which puts at its focus code *as text*. Implicitly, this focus suggests that a given body of code, and its author(s), have something meaningful to offer to its reader, whether that meaning is provided intentionally or not. An additional significance of focusing on code is that *the algorithm itself* becomes the object on which inquiry is centered: how it is structured, phrased, and expressed all contribute clearly and explicitly to the interpretive experience. In this light, code is no more a simple tool than any other form of language, and its capacity for meaning making is not only acknowledged but emphasized and celebrated.

It is easy to consider the semiotic value of code when its syntax closely resembles that of natural language, and many languages use syntax resembling that of English. The idea that code could and should be written primarily for human readers, rather than for the computers that interpret the code, has its origin in Knuth (1992), whose idea of "literate programming" required a radical reconsideration of how computer science might be approached. For Knuth, it was crucial that code be composed in such a manner that its meaning was clearly articulated to human audiences; its ability to be executed properly or accurately by a computer was secondary. Essentially, the idea is that human collaborators should be able to comprehend any of their colleagues' work and the functional intent of that work. This perspective has been echoed by influential programmers since then, including in several texts on fundamental programming principles and practices (Kernighan and Plauger 1978; Kernighan and Pike 1999). Matsumoto (2007) urged his audience specifically to treat code "as an essay," and he demonstrated doing so through a series of example programs written in the Ruby language (477).

There is a relationship between saying (describing) and doing (computing or executing) in source code and the executable programs they describe, but this relationship is not always emphasized or addressed in specific code texts. "Codework," a kind of code poetry, blurs that relationship with a similar relationship in written language between "saying" and "appearing," what Lanham (1993) referred to as a bistable oscillation between looking "AT and then THROUGH" texts, an oscillation that is never eradicated but might fluctuate to varying extents between audiences and contexts (5). Cayley (2002) wrote codeworks in order to experiment with the possibilities of maximizing code's ability to perform computational tasks while also clearly communicating its goals to a reader in a conventionally understandable manner. For Cayley, "codework" as a term highlighted its distinction from the vast majority of code that is technically productive but not intended to be artistic in form. The following is a brief excerpt from one such codework:

```
if invariance > the random of engineering and not
   categorical then
  put ideals + one into media
  if subversive then
    put false into subversive
  end if
```

```
if media > instantiation then
  put one into media
end if
```

(Cayley 2002)

The above was composed by Cayley in HyperTalk, a programming language developed in the late 1980s for the Macintosh hypermedia program HyperCard. Its syntax, like that of many other high-level programming languages, closely resembles that of English, making it easily readable by humans (assuming those humans understand English). The larger program from which this excerpt was taken is a text generator, with the various terms Cayley included here (such as **`media`**, **`subversive`**, and **`ideals`**) serving as "containers" for variable data values as part of the code's expression. For example, **`media`** holds a number the value of which changes depending upon certain conditions (such as whether the current value of **`media`** is greater than the current value of **`instantiation`**), not unlike the contingent meaning of *any* term for a particular context and discourse community. The meaning of the code, then, emerges not only from what the code text *says* in English (or at least near to it) but what it *does* computationally, even if we might view the functional purpose of this, or any other example, program to be trivial (Sample 2013). Cayley acknowledged the code's "ambiguous address" of both human reader and computer interpreter, implicitly suggesting that there existed rhetorical situations for each of these audiences (2002).

While Cayley's example is relatively readable and accessible, most code—as Marino (2006) has observed—does not closely resemble literature or other genres or forms of conventional discourse. This means that the dichotomy of audiences for code-based communication may *seem* to skew more toward the technological than to the human. A number of rhetoricians and critical code scholars have interrogated this balance, questioning the value of a traditional human-oriented communicative hierarchy in favor of a more distributed, relational network of human and nonhuman actors (Arns 2005; Cummings 2006; Brown 2015; Nicotra 2016).

Arguments in Code as Algorithmic Meaning Making

Just as algorithmic procedure has provided means for artistic and poetic invention, so too have algorithms served rhetorical invention, described and expressed not only in code but in a variety of other communicative

modes. The notion that an algorithm can and does work rhetorically is a radical departure from its conventional definition, which emphasizes discrete procedural execution of its components. In some computer science contexts, there is significant discussion about the optimal way to construct a particular program or function in a given language, focusing not so much on what a procedure *means to do* but rather on how to clearly, effectively, and efficiently state and structure the steps of that procedure. There is an implicit recognition of the procedure's purpose and its value, but those qualities do not occupy the center of discussion. As Edmonds (2008) noted, novice programmers often find themselves needing to shift mentally "from viewing an algorithm as a sequence of actions to viewing it as a sequence of snapshots of the state of the computer" (6). This shift, he argued, is significant because it draws attention to how code computes data from one action to the next within a procedure rather than on what the end result does or means. In other words, the scale of critical inquiry assumes that the point or goal of a procedure is already determined or understood, rather than remaining germane to the discussion about how best to solve a relevant problem.

The novice mentality described by Edmonds (2008) is closer to the mark for thinking critically in regards to algorithms and rhetoric than the conventional "learned" mentality, since the novice student has not yet been trained to disregard certain critical perspectives in order to approach the algorithm "correctly" for academic or industrial purposes. Berlinski (2000) noted that an algorithm is, in addition to the strict, conventionally computational procedure he had initially provided for his reader, "a set of rules, a recipe, *a prescription for action*, a guide, a linked and controlled injunction, an aduration, a code, an effort to throw a complex verbal shawl over life's shattering chaos" (xvi, emphasis added). Berlinski's reference to a "prescription for action" is not just a means of defining a particular action to occur in a certain way but to call for that action to be undertaken by a certain audience, laying bare the algorithm as a procedural engagement with a rhetorical exigence.

What a recognition of algorithm *as meaning making* offers rhetoricians, then, is an opportunity to explore how a seemingly "machinic" manner of discourse—the code of digital technologies and media—can provide insight into the interplay among the canons of rhetoric to influence the potential expression(s) of a particular rhetorical situation. For example, how does style affect code-based composition practices in order to facilitate action in human agents? How might a critical acceptance of a technological

code compiler as corhetor (inventor, arranger, etc.) alter our understanding of the rhetorical concepts of constraint or suasion? As digital technologies become more advanced and accessibly modifiable (in the sense of code syntax reflecting natural language and performing machinic functions), these rhetorical concerns become increasingly significant. If we are to understand how we communicate with and persuade one another with the aid of digital technologies, it is important to understand how we are capable of stimulating particular types of action through the use of those technologies. A consideration of the rhetor's ability to affect, at one or more code levels, constraints that extend to other modes and means of action is vastly different from traditional approaches to rhetoric, which may take as given the technological mechanisms constraining particular discursive efforts.

Procedural Rhetoric

While it may appear trivial, there is a significant difference between algorithmic meaning making and the related idea of procedural rhetoric. Procedural rhetoric as described by Bogost (2007) deals with the influential qualities of procedure-based *systems* exerted upon individuals who make use of those systems. Bogost, however, did not elaborate on a wide variety of code-based algorithms as much as on how algorithms function in regards to interactive systematic procedures like video games and how games' procedures persuade players to act within a game's context. For Bogost and for others since (including Sample 2013), the rhetorical capabilities of a procedural system offered new ways of engaging specific populations that might otherwise be ignored or overlooked. A video game teaches its player the rules of its "world" through activity within the game; the player, through trial and error, explicit tutorial, or both, learns what behaviors and perspectives are acceptable and "valid" while engaging that game's system. The rhetorical action that occurs at the user level emphasizes the way(s) a game's developers intend for its content to be encountered by a player.

But there is also significant rhetorical action at the developer level, demonstrated by the ways through which the developers constrained particular means of user interaction with the processes of a given game world, effectively making use of another level of procedural rhetoric—wherein one developer persuades another—to facilitate the game itself. As Bogost described it, "processes define the way things work: the meth-

ods, techniques, and logics that drive the operations of systems from mechanical systems like engines to organizational systems like high schools to conceptual systems like religious faith" (2007, 3). This definition suggests that code—as a symbolic representation of algorithmic procedure—might play a prominent part of Bogost's discussion of procedural rhetoric. Bogost noted, however, that some processes that "might appear unexpressive, devoid of symbol manipulation, may actually found expression of a higher order" and explained how those humans who are perceived as "breaking procedure" in actuality are constructing new processes, and expressing them, in order to complete tasks (2007, 5). This adjustment of emphasis (to "higher order" expression), while hinting at the possibilities of computation for rhetorical ends, facilitated Bogost's close analysis of video games and gameplay experiences as procedural expressions.

Rhetorical procedure has been explored further by Lanham (2003), who referred to such procedures as "tacit persuasion patterns" (119). For Lanham, tacit persuasion occurs constantly, since we are almost never acutely aware of all influences attempting to exert themselves upon us. As Lanham observed, individuals often "feel" the presence of tactic persuasion patterns "subconsciously, even if we do not bring that knowledge to self-awareness" (2003, 120). This description hints not just at a passive acceptance of rhetorical appeals but a subconscious engagement with the variables of a rhetorical situation as well as enthymematic arguments provided by a rhetor across multiple levels of language. As a scholar interested in speech and writing, Lanham focused primarily on persuasive techniques available in discursive language, such as rhyme, chiasmus, alliteration, and anaphora. Each of these devices provides a rhetor with the ability to draw or hold the attention of an audience that might otherwise have not bothered to heed an argument. Such devices suggest a significance in the argument through the affordances of languages' stylistics. While Lanham did not address the possibilities of tacit persuasion patterns or devices in artificial (code) languages, they nevertheless exist and are already used frequently by many developers working collaboratively, as will be discussed in subsequent chapters.

Rhetorical code studies continues this conversation in regards to the potential for expressive action through the languages of computer code. Code might be described as "inexpressive" in that it creates and communicates meaning in ways that often differ from conventional invention and delivery of discursive arguments; it is precisely because of this quality, however, that its procedural nature can demonstrate expressive out-

comes in novel and unique ways. Here I set aside the specific expressive acts of procedural execution (which would highlight the general user's experience with a particular software program) and instead discuss how the construction of algorithms in and through code as both *text* and *process* functions persuasively on user and developer as well as, to a lesser extent, on technological systems including user workstations, servers, network routers, sharing economies, etc. In other words, my interest in algorithms is focused on how code, as a medium to describe algorithmic procedures, is articulated for rhetorical purposes.

Algorithms We Live By: Recognizing Rhetorical Algorithms

In order to discuss how an algorithm can act rhetorically, I turn to investigate how algorithms in code are composed: what they do (the actions they attempt to induce), what they say (how their operations are constructed), and how they say it (what those operative constructions mean to different audiences). These qualities are not necessarily any different from conventional rhetorical concerns of invention, style, and delivery, but their construction in code might certainly make it seem as if they are, and Beck (2016) has demonstrated how algorithms and code function rhetorically through their performative and linguistic natures. One particularly significant quality of code, as with other forms of language, is the symbolic action (cf. Burke 1969) that a given code operation, or set of operations, provides, its descriptive qualities making its purpose and function(s) understandable to various human audiences and facilitating further activity.

Computational processes are similarly metaphorical; the rhetorical actions they symbolize succeed primarily because of the metaphor-driven ways human audiences are influenced to understand those processes as arguments that influence their audiences in particular contexts for particular purposes. It is also important to observe that writing or speaking about algorithms (or, potentially, even *to* algorithms; see Gallagher 2017) is itself a metaphorical activity that frames procedure as a kind of description. As Bogost (2007) has noted, "only procedural systems like computer software actually represent process with process" (14). With this in mind, I will identify some integral means by which code processes make meaning for developer audiences in what those processes do functionally and in how they are structured for the sake of human readability as well as for technical or technological expression.

In order to make this argument, I need to demonstrate how relatively

simple code algorithms appear to, and do, work rhetorically. In the following pages, I address several examples of increasing complexity whose code texts communicate at various levels the meaningful action they mean to effect through the computational operations that make up their text forms. First, I look at FizzBuzz, a program that iterates through a specific body of data, used in hiring tests to determine applicants' knowledge of algorithmic principles. Second, I examine quine, a program that outputs its entire code content. Third, I turn to HashMap concordance, a program that tracks word frequency across a set of input text (in this case, Mary Shelley's *Frankenstein* and Bram Stoker's *Dracula*).

CASE 1: FIZZBUZZ

The first example to be scrutinized, the "FizzBuzz test," is relatively simple in construction and intent (although elsewhere I have explored the rhetorical canon of style in greater depth than is offered in the discussion below; see Brock 2016). It is the focus of a common hiring test for programmers, intended to weed out applicants who are unable to work through the basic principles of algorithmic procedure and expression. The goal of this algorithm is to take the numbers 1 through 100 and print them out, one at a time, *unless* they are multiples of three, in which case the word "Fizz" should be printed, *or* if they are multiples of five, in which case the word "Buzz" should be printed. There is an implicit extra rule here; specifically, numbers that are multiples of fifteen (that is, both of three and of five) should print out "FizzBuzz." The entire output of the program—which is identical across all four of the examples provided below—is the following single line of text (although some variations on the test would include line breaks, spaces, or some other distinguishing markers between each output iteration):

```
12Fizz4BuzzFizz78FizzBuzz11Fizz1314FizzBuzz1617Fizz
  19BuzzFizz2223FizzBuzz26Fizz2829FizzBuzz3132Fizz
  34BuzzFizz3738FizzBuzz41Fizz4344FizzBuzz4647Fizz
  49BuzzFizz5253FizzBuzz56Fizz5859FizzBuzz6162Fizz
  64BuzzFizz6768FizzBuzz71Fizz7374FizzBuzz7677Fizz
  79BuzzFizz8283FizzBuzz86Fizz8889FizzBuzz9192Fizz
  94BuzzFizz9798FizzBuzz
```

Depending upon the language, and even more importantly depending upon *the way a programmer approaches the problem*, there may be dozens of

ways by which one could construct this algorithm effectively, as suggested by Lopes (2014) in regards to the task-frequency program she wrote in more than thirty different programming styles. The path ultimately taken by any particular developer to solve his or her problem provides a great deal of rhetorical insight about that developer's abilities, limitations, and preferences in computing data, using certain languages, and working in certain development environments.

While the immediate pragmatic goal of the FizzBuzz test is for an applicant to demonstrate to a potential employer his or her competence as a programmer, the applicant also demonstrates more generally an understanding of how "best" to use a given language (whether defined as computationally elegant, readable, or possessing some other quality). Further, the applicant also communicates through the written code how he or she *thinks* computationally through the frame of that particular language in order to solve certain types of problems. It is also important to note that my analysis of the FizzBuzz test, focusing on stylistic influence on procedural expression in code texts, is not meant to erase or elide the discriminatory tactics that can accompany use of such a test or other means of constructing organizational communities (Steinberg 2014; Burleigh 2015) or the underrepresentation of women and some minorities in the software industry and OSS communities (Lopez 2017; Reagle 2013). Both of these issues influence who is likely to have learned programming (and gain appropriate credentials or degrees) and pursue a position for which this test would be administered.

In table 2.2, the FizzBuzz algorithm has been written in two similar but significantly different ways using the syntax of the JavaScript scripting language, a popular code language used in web pages, PDF documents, and even desktop applications. This sort of algorithm is described as a *loop* because it continues to compute results so long as the proper conditions are met, in this case while the input amount (`i`) is a number lower than or equal to 100. Example 2.2.a, on the left, frames its computation in an initial "catch-all" condition statement, that is, that `i` is a multiple of three *or* of five. (The syntax `i%3` checks whether "`i` divided by 3" has a remainder of zero.) Then it checks each of those subconditions independently of one another. This means that `i` could simultaneously be both a multiple of three and a multiple of five, triggering the operation that will execute when each of those conditions is met (printing both "Fizz" and "Buzz"), without, in its current form, impacting the outcome of the other conditional computation. In comparison, example 2.2.b, on the right in

table 2.2, functions due to a logic of exclusion. First, it checks whether **i** is explicitly a multiple of fifteen. If it is not, then the loop checks first if **i** is a multiple of three. If *that* condition is not met, *only then* does the loop repeat its check for **i** as a multiple of five. These conditions are dependent upon one another: that is, in the code on the right, a number computed to be a multiple of three is not also computed as a multiple of five.

The variations on how to construct and express a FizzBuzz algorithm are not limited to these sorts of conditional checks, either. Table 2.3 contains two examples of the algorithm as composed in the Ruby programming language. The major difference between these two examples written in Ruby lies not in how the specific conditions are constructed (although the construction does differ between the two) but rather in the *type of function* that is used to form the loop itself. This is significant in that there is a fundamental shift in the logical structure of the loop and also of the hypothetical larger program of which the FizzBuzz algorithm might be a representative part. Example 2.3.a, on the left of table 2.3, makes use of a **for** loop, which—as with the JavaScript examples—iterates through a body of data and computes each item within that body before moving to the next item. In these loops, that data has been the set of integers from 1 to 100, but **for** is not limited to iterating numerical data. (In Ramsay's algorithmic reading of *The Waves* discussed earlier in this chapter, the novel's text served as the data population, separated out into units of individual words.) Example 2.3.b, however, only *imitates* that kind of looping behavior. It technically repeats the operations within its scope a set number of times and, in doing so, manipulates the value of a variable (**i**) within its scope each time. This particular example *incidentally* includes a

Table 2.2. Two example FizzBuzz loops in JavaScript

Line	Example 2.2.a	Example 2.2.b
1	`for(var i=1;i<=100;i++) {`	`for(var i=1;i<=100;i++) {`
2	`  if ((i%3==0) || (i%5==0)) {`	`  if (i%15==0) {`
3	`  if (i%3==0) {`	`  console.log("FizzBuzz");`
4	`  console.log("Fizz");`	`  } else if (i%3==0) {`
5	`  }`	`  console.log("Fizz");`
6	`  if (i%5==0) {`	`  } else if (i%5==0) {`
7	`  console.log("Buzz");`	`  console.log("Buzz");`
8	`  }`	`  } else {`
9	`  } else {`	`  console.log(i);`
10	`  console.log(i);`	`  }`
11	`  }`	`}`
12	`}`	

line wherein the value of **`i`** is changed; it is not inherently connected to the **`times`** method used here. Because of how **`times`** operates, counting begins at zero rather than one, and thus **`i`** needs to have an extra number added to its current value (in line 2, the first operation within the **`times`** block) before the remaining operations can accurately be computed for the FizzBuzz problem as it is posed.

Conceptually and metaphorically, this distinction between **`for`** and **`times`** reflects a fundamental distinction between iteration and repetition. The **`for`** loop suggests to human and machine readers that similar types of data are going to be computed through a series of operations whose scope and syntax may be influenced by the specific data being calculated and modified at any given moment. In contrast, the **`times`** "pseudo-loop" suggests that it will execute *the same operations* a set number of times, independent of any input variables; any data manipulated differently from other data as a part of that loop is an incidental consequence of its code composition. More generally, each FizzBuzz example—and, more broadly, any block of code that processes a body of data—is a computational metonymy: the "loop," which implies a cyclical return to its origin, is actually more like a corkscrew. Its abbreviated description of operations to be executed across its data parameters never truly returns back to "the beginning" of the code, as each iterative execution transforms the code both in how it reads and in how it operates, just as the input data is transformed into appropriate output data.

While the function of the FizzBuzz algorithm, as represented by these examples, may not initially appear to be rhetorical in nature (since it checks a set of numbers and prints out numbers or words), it nonetheless

Table 2.3. Two example FizzBuzz loops in Ruby

Line	Example 2.3.a	Example 2.3.b
1	`for i in 1..100`	`100.times do |i|`
2	`  if i%3 == 0 then`	`  i = i+1`
3	`  print "Fizz"`	`  if i%15 == 0 then`
4	`  end`	`  print "FizzBuzz"`
5	`  if i%5 == 0 then`	`  elsif i%3 == 0 then`
6	`  print "Buzz"`	`  print "Fizz"`
7	`  End`	`  elsif i%5 == 0 then`
8	`  if i%3 != 0 && i%5 != 0 then`	`  print "Buzz"`
9	`  print i`	`  else`
10	`  End`	`  print i`
11	`End`	`  end`
12		`End`

serves as a concise example of the persuasive capabilities of code. Specifically, the FizzBuzz algorithm provides meaningful information to its applicant author, to any other human readers (e.g., the employer), and to the machine related to what the author understands about how to engage in the manipulation of a particular set of computer data. If the FizzBuzz code in each example is read as an excerpt from a larger program, its contents signal a set of rhetorical and computational decisions that have been made about how to most effectively accomplish its task. In essence, FizzBuzz communicates more, and *other*, than its output: it suggests to other agents how the author has identified at least *one* central means around and through which to compute relevant data and facilitate the desired result. Further, it implies a suggestion as to how one should understand and work with that data toward a perceived desired end.

CASE 2: QUINE

The second example to be discussed here is the *quine*, defined most concisely as a "self-reproducing" program. In other words, the output of a quine is the sum of its code content, meant to mirror that content perfectly. The term "quine" derives from the name of mid-twentieth-century logician Willard Van Orman Quine, who provided the following self-referential paradox: "'Yields a falsehood when appended to its own quotation' yields a falsehood when appended to its own quotation" (Quine 1976). Quine meant that the statement provided can accurately be neither true nor false, and it is only in the combination of concept and *quoted* concept that the paradox's meaning emerges. Rhetorically, the algorithmic quine offers an opportunity for a rhetor to consider what it means to construct a "logically sound" argument and how to communicate that logic to a given audience. Is the argument merely an appealing effort at constructing or suggesting the construction of meaning, or is there an internally valid consistency to a quine? Does the former require the latter? How transparent or opaque should its logical mechanisms be to the audience so that it can recognize (or not) how the rhetor works to persuade it to action? An algorithmic argument generates much of its appeal through its consistent logic, although this consistency does not necessarily demand any objective truth or accuracy to succeed.

One simple quine in code, written in the Ruby programming language, is a single-line program provided in its entirety:

```
x = "x = %p; puts x %% x"; puts x % x
```

("Quine" n.d.)

Within this line of code, there are two distinct operations that occur. The first is to define the variable x, and its value consists of the characters within the quotation marks; this operation ends at the second semicolon. In essence, x is defined as a data type called a "string," a container for some arbitrary set of alphanumeric characters (whose boundaries are identified by the quotation marks preceding and following the string's content). The second operation in the quine, beginning directly after that same semicolon, recalls and displays for the user the string's content exactly, via the **`puts`** function. It is important to note that this second operation *must* include its own call as part of its output in order for the quine program to be considered fully self-referential (i.e., the recall command itself has to appear in the output of the recall). When **`x`** is defined in part as **`%p`**, **`%p`** serves as a container in which other content might be substituted later; the statement **`puts x % x`** effectively inserts the content of **`x`** into itself so as to print out the quine's input correctly. Many programmers separate the content of a quine into two components, what they refer to as the "code" (the operations to be computed when the program is run) and the "data" (the noncomputing replication of those operations). This binary quality of the quine, for developers, allows them to make explicit note of what "runs" (i.e., what computes) and what is output (i.e., what is computed).

When we examine the code-based quine rhetorically, this distinction changes as we turn from computational success to meaningful action as a criterion of evaluation. The action of the quine's code is to reveal its data as procedure and output; the action of the data is to highlight the means by which it was revealed, the computations constituting the code itself. It is significant that the two components function reflexively rather than the code unidirectionally working "on behalf of" the data; the quine as a whole is displayed as an apparently complete persuasive entity. That is, because the quine outputs itself, "everything" is made clear to the user who executes it. It is this relationship that Cayley (2002) referred to when identifying code existing as both text and not-text (as objects of study); the code and data components of a program are inherently interrelated, but the reading of code as executable action and as meaningful language are two different activities. For the quine, much of its effect—its demonstration of its "completeness"—is due to the appearance of the quine as a transparent argument. In simple terms, this appearance suggests that the quine does (only) what it says and it says (only) what it does, presenting the *semblance* of a computational chiasmus that is completed when the input is ultimately displayed as output.

The quine *also* implies, though, that it is a simple program that merely presents a plainly stated message: its content. In a sense, this implication is the perfect argument that could be made by a rhetor: the tools used to make a case *make up* that case. Miller (2010b) has pointed out the common rhetorical tactic of self-denial as a way of playing down or otherwise concealing the acknowledgment that a persuasive effort is currently taking place. A significant component of this tactic, she argued, is *mimesis*, the idea that language has the potential to represent its subject so clearly and faithfully that it need not (or cannot) deceive in its representation. As a result, a rhetor must work not only to conceal his or her true intentions in using language for a given purpose but also to conceal the fact that the rhetor is concealing anything. This idea of language as apparently, but not actually, mimetic is key to an understanding of the quine and what it can achieve rhetorically (and, perhaps, reminiscent of the debate over technical communication as instrumental or humanistic; see Miller 1979; Johnson 1998; Moore 1999; Johnson 1999; and discussed further in Dubinsky 2004). It is not *simply* the sum of its parts (that is, its expression does not only equal its content); it is a means of suggesting, in more general terms, that all code can be reduced to such a description and that no further critical inquiry as to its purpose or mechanical procedure(s) is necessary.

One complication for the quine—and thus, by association, any executable software program—is that it is possible to hide from audiences what the true intentions *and content* of a code-based program may actually be. Thompson (1984) demonstrated the ease with which a skilled developer might circumvent the transparency of code composition and execution by inserting instructions into a UNIX machine's compiler software. The compiler is an intermediary, a translation program whose function is to transform source code (readable by humans) into an executable program (readable by the machine). Thompson revealed that it was possible to manipulate the code of a compiler in such a way that it would be undetectable to anyone using the compiler, *along with any other programs the compiler subsequently compiles*, for other purposes—including the execution of a quine. Thompson's point was that one could trust *only* the code one wrote: all else was potentially devious, even a program that purported to print the entirety of its own code. Whenever one interacts with an external entity (e.g., code written by another person), it may not only be *difficult* to tell whether some meaning is concealed but it may be *impossible* to verify whether any concealment has ever occurred unless attention is drawn thereto (cf. Miller 2010b). While the quine itself may not be to blame for

any malicious behavior on a compromised system, it is nonetheless reliant upon the ecology of technological and human agents in which it exists in order for its computation and subsequent expression to occur "successfully."

The quine provides a valuable example of code as a text that is not significant merely because it acts rhetorically through its expression—by revealing its content—but also because it emphasizes the complicated nature of code as a text-practice that does and says more than what it explicitly describes in its composition and expressed performance, in what Chun (2011) has described as "a crafty, speculative manner in which meaning and action are both created" (24). Just as with more conventional rhetorical activities that focus on meta-rhetorical subjects, the quine has the potential to influence audiences to reconsider the communicative event itself. In the case of the quine, this reconsideration results in an awareness of the self-description as a necessarily enthymematic, rather than a fully contained and transparent, argument. This argument calls attention to the ability of code to do more than it suggests, especially when it suggests that one has access to the code in its entirety, including the full functionality of its expressive ability.

CASE 3: HASHMAP

The third example to be discussed transforms and deforms existing texts—and conventional readings thereof—in order to bring to light new meaning that might not have been exposed in the text's original format. This sort of code, and its paratextual output, is aligned closely with the algorithmic criticism of literature promoted by Ramsay (2011) and others. Here the example code is included not so much to demonstrate the possibilities of code for literary criticism but instead to highlight how algorithmic manipulations of text work to engage different audiences rhetorically. This example was composed by Shiffman (2014) for, and to be included with, the Processing integrated development environment (IDE), which uses a streamlined syntax based on the Java programming language. Shiffman's code makes use of a type of data called a **`HashMap`**, which serves as a way to store a "collection" of individual data elements so that each has its own identifying **`key`** data. By default, the program takes a plain text file with the contents of Mary Shelley's *Frankenstein* and Bram Stoker's *Dracula*, the text for each novel made available through the electronic Project Gutenberg public domain library, and displays each word in a particular font size related to the frequency of each word's occurrence in

either novel. The program will *not* count a word if it appears in both novels or if it appears fewer than five times. The novel's words, reconfigured as a **HashMap** (a kind of data table that stores paired "key" and "value" data), offer in their recombined form a way to read meaning in each text, and in comparison to the other text, based upon the frequency of particular terms and concepts. Table 2.4 contains excerpts from Shiffman's code, which appears as an example project in two files ("HashMapClass.pde" and "Word.pde").

The program takes the contents of *Dracula* and *Frankenstein* and strips out all punctuation so that only words (data "strings" of characters, referred to as hash map "keys"), and the spaces between them, remain. Then each word is checked against the current contents of the **HashMap**

Table 2.4. Excerpted HashMap example code by Shiffman (2014) written for Processing, from "HashMapClass.pde"

Line	Code from "HashMapClass.pde" file in HashMapClass example (Shiffman 2014)
27	`  words = new HashMap<String, Word>();`
[...]	[... *These lines create variables populated by the novels' texts*]
50	`void loadFile(String filename) {`
51	`  String[] lines = loadStrings(filename);`
52	`  String allText = join(lines, " ").toLowerCase();`
53	`  String[] tokens = splitTokens(allText, " ,.?!:;[]-\"'");`
54	
55	`  for (String s : tokens) {`
56	`    // Is the word in the HashMap`
57	`    if (words.containsKey(s)) {`
[...]	[... *These lines locate the appropriate key and update its value*]
68	`    }`
69	`    else {`
70	`      // Otherwise make a new word`
71	`      Word w = new Word(s);`
72	`      // And add to the HashMap put() takes two arguments, "key" and "value"`
73	`      // The key for us is the String and the value is the Word object`
74	`      words.put(s, w);`
75	`      if (filename.contains("dracula")) {`
76	`        w.incrementDracula();`
77	`      } else if (filename.contains("frankenstein")) {`
78	`        w.incrementFranken();`
79	`      }`
80	`    }`
81	`  }`
82	`}`

and, if it does not currently hold a position within the **`HashMap`** container, is added thereto with a value of 1. If the word *does* already exist within the **`HashMap`**, then the appropriate value for that word is increased by 1. Once this check is completed, it is repeated for the next word in the novel, and then the next, until the entire novel has been iteratively "read" in this fashion. This activity—or at least an activity closely related to it—is described as "text mining," referring to the act of extracting meaningful data from an otherwise opaque or unexamined source text.

As these word count checks occur, each word that has been computed is written onto the screen at a random spot along the top with its incrementer value influencing the size of that word on the screen. Words that occur more frequently in either novel are displayed in larger font size and move down the screen more quickly, and only so many words are displayed on the screen at any given time, even though the total calculation of the novels' contents continues to execute. In other words, there is a distinction between what the user and his or her machine experiences, with the former engaged in a particular constrained reading of *Dracula* and *Frankenstein* that is acutely distinct from Processing's interpretation of the data as solely important in its numerical sense.

There are several meaningful activities that occur within the lines of the **`HashMap`** code that, together, dramatically alter the traditional concept of reading and writing. Perhaps the most significant of these is the incorporation of object-oriented programming (OOP) principles into these activities. In OOP, distinct "objects"—clusters of expressive code entities—are created from a framework of rules (called a "class"); each instanced iteration of an object obeys the same procedural rules but responds to those rules independently of other object instances; that is, it calculates and expresses its procedures without any *inherent* influence on any other object, although this may occur incidentally. The result is a set of objects whose behavior has the same fundamental principles but that emerges uniquely for each individual based upon the constraints of its being called or created as part of a given program; in this example program, each displayed word is a separate object. (As an aside, OOP is unrelated to the philosophical subfield that has been named object-oriented ontology for its focus on nonhuman entities.) What OOP allows for is the potential for a multiplicity of contextual meaning made possible through the expression of iterative object creation and activity emerging from an initial set of algorithmic procedures.

At the same time, the nuanced context of each iteration of a given class

Fig. 2.1. Example expression of Shiffman's (2014) HashMap code

is slightly flattened as it is expressed within the constraints of a particular situation; for example, as the **`Word`** "Lucy" is counted by Shiffman's program, the meaning of each use of the name reference is stripped away. There is no remaining distinction between when Lucy speaks, when she is referred to by other characters, or what sort of emotional tenor surrounds those utterances. Instead, the name, as a **`Word`** object defined by the code, becomes notable primarily as a unique string of alphabetic characters that repeats so many times over the course of the novel, represented visually in contrast to other word strings by its expressed size on, and speed with which it travels across, the screen. Lucy—along with any other word from either novel transformed by this program—gains significance through certain demonstrations of numerical frequency. Because of the way Shiffman coded his program, there are a few intriguing side effects of this shift in significance (see figure 2.1).

Through the expression of Shiffman's code, it becomes clear that the most frequent words in *Dracula* (and, potentially, in most novels or other substantial bodies of text) are words that might be considered the least important: the conjunctions, articles, and prepositions that link together more "significant" concepts and messages. It is possible to add into the code exclusionary conditions that would ignore these seemingly trivial words (often referred to as "stop" words), such as the single character "t" in figure 2.2, likely a remnant from the splitting of words by punctuation characters, outlined in line 53 of table 2.4; to support this theory, the word "didn" appears below "t" at roughly the vertical midpoint of the figure, nearly blending into "flies" to its right. Any subsequent analysis of word prominence via frequency made while ignoring such words or word fragments could be considered technically inaccurate or incomplete. Further, the decision to reveal the computations of word frequency in this manner

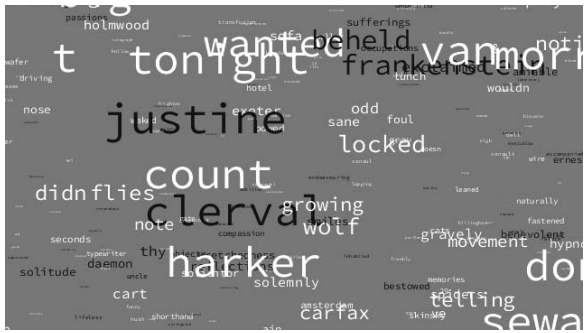

Fig. 2.2. Later expression of Shiffman's (2014) HashMap code

results in a very different visible outcome than through some other approach, such as generating a text list of each word and its final frequency count. Because only a subset of words from each novel appears on the screen at any moment, the user ultimately is left unaware of the significance of words that do not continually appear or that do so in a small font size (see figure 2.2).

In figure 2.2, a screenshot taken at another moment during the same expression of the code displayed in figure 2.1, there is a noticeable difference in the makeup of visible words. While the code continues to cycle through the words in each novel, the user is only given access at any time to a small set of the computational results. Words such as "mina," "murderer," and "bag" (visible in figure 2.1) have all been repositioned beyond the bounds of the program's output window, obscuring the visualizations of their frequency; but "harker" has cycled across the screen again at this point, its position unintentionally captured at nearly the same vertical point as in figure 2.1.

Why might a program seemingly so trivial—at least in terms of how it presents a "reading" of *Dracula* and *Frankenstein*—be worthy of discussion? The choices Shiffman made in creating this program signal to user and to developer a particular set of values related to reading and (re)writing these novels algorithmically. Word-level elements of the novel are deformed contextually from their original meaning and given new context through significance; the words that appear more often in the text appear more prominent in the code's expressive arrangement/layout. At the same time, the way Shiffman has chosen to determine what defines a word (based on the characters that might surround it) alerts other coders, *and the system expressing the code*, as to how to arrange and read the text he or she inputs. This is not simple instruction entirely devoid of persuasion: we are

influenced to (more likely) accept a developer's meaningful text making by the symbolic action engaged in by all participants of the algorithm's construction, interpretation, and computation. As part of this action, we may more effectively evaluate how successfully the author-developer and the computer have been in generating and demonstrating this argument for a new form of reading if we can understand how we are expected to interpret that argument and read the created paratext in a particular way.

Conclusions

More generally, each of these examples discussed above serves to demonstrate some fundamental qualities of code as a significant form of making meaning rhetorically. While on its own each program may seem to serve only a limited purpose, together the code functions provided in these examples work—alongside thousands of others used daily—in more robust software programs to persuade and influence numerous human and technological audiences to act in rhetorically meaningful ways. These ways may not always be visible, clearly recognizable, or discursive in nature, but they nonetheless create meaning and work to persuade the human and nonhuman audiences they engage to induce various types of change in the ecologies in which they operate.

Further, the potential for meaningful symbolic action that is demonstrated in these examples reflects the close connection that has existed for millennia between algorithmic procedure and humanistic activity, in terms of both creatively and critically oriented work alike. While the vast majority of scholarly rhetorical focus has thus far centered on a variety of communicative modes including writing, speaking, image, and even place, the scholarship on *procedure*—and especially algorithmic procedure demonstrated through digital and computational media—as means of persuasion is relatively sparse. Rhetorical code studies, however, serves as a space in which to tease out the rhetorical potential of algorithms, and of software code as a particularly significant contemporary form of constructing algorithms, to facilitate action in audiences.

The following chapter demonstrates the potential of code to facilitate action through an examination of the spheres of discourse *surrounding* the composition of code in development communities. Exploring written communication about code can provide a means of demonstrating the rhetorical strategies used by programmers to develop and promote software among certain populations (namely, other current and potential pro-

grammers). The values that developers stress in their code and discursive commentary, in addition to the persuasive tactics they use to build particular types of identities and communities, highlight a set of qualities likely to be communicated in and through the code they produce. Building on this foundation, we will then turn to bodies of code texts to see how those persuasive expectations—and the strategies believed to facilitate the realization of those expectations—play out in a given development case.