

1 Vocable Code

If program code is like speech inasmuch as it does what it says, then it can also be said to be like poetry inasmuch as it involves both written and spoken forms.¹ The analogy to poetry suggests numerous aesthetic and critical possibilities for code, beyond its serving simply as functional instructions.² There are many examples of programmers working in this spirit, referring to the “poiesis” of code (its making), and working with the connections between the structural and syntactic qualities of written code and poetry, or in spoken performances where source code is read aloud as if a poem; Franco Berardi reading the source code of the “I Love You” virus in 2001 is one such example.³ This esoteric practice makes historical reference to artists expressing language as found material and arbitrary noise, in bruitist poetry and simultaneous poems from the Dada period for example, where texts in different languages were read aloud at the same time. Using these ideas as a point of departure, this chapter aims to draw attention to the aesthetic aspects of programs but particularly to their speech-like qualities, to make apparent the vocable elements that underpin some of the unstable aspects of coding practices.

Pertinent to the initial discussion are the Dadaist “sound poems” of Kurt Schwitters, who described his method of combining diverse elements to undermine the boundaries between aesthetic forms. A deliberate confusion was produced, somewhat similar to coding practices in which written form and output are conflated. In the case of Schwitters, output was expressed through his arrangements of what he called the “banalities” of objects and language by focusing on vocables (the recognizable elements that are capable of being spoken). Treating these like data open for reassemblage, he explains: “These materials are not to be used logically in their objective relationships, but only within the logic of the work of art. The more intensively the work of art destroys rational objective logic, the greater the possibilities of artistic form. Just as in poetry word is played off against word, so in this instance one will play off factor against factor, material against material.”⁴

This explains how his language experiments play with unorthodox combinations and lack of referentiality. For example, *Ursonate* (1922–1932) is a sound poem

composed of nonlexical vocables (strings of phonemes that make “non-words”) performed by Schwitters himself entirely from memory. The full poem is highly structured, arranged in four movements over a total of 29 pages, and takes approximately 30 minutes to read. An excerpt follows (and you can hear a recorded recital at UbuWeb):

```
Fümms bö wö tää zää Uu,
                                pögiff,
                                kwii Ee.

Ooooooooooooooooooooooooooooo,
dll rrrrr beeeee bö
dll rrrrr beeeee bö fümms bö,
    rrrrr beeeee bö fümms bö wö,
        beeeee bö fümms bö wö tää,
            bö fümms bö wö tää zää,
                fümms bö wö tää zää Uu:5
```

The phonetic aspects of the human voice are foregrounded at the expense of more conventional understandings through semantics and syntax, and in this way the poem is more directly grounded in the sound-making apparatus of the body (close to what some call “pure meaning,” a single, simple sound or tone, free from mixture or combination). Yet even in their abstraction, the sounds would be familiar to any German speaker (like Schwitters himself), who would understand them relative to precise learned movements of their own vocal tract. In one sense, this is poetry about the shape of words in the mouth, and the direct, perhaps synesthetic relationship between mouth shape and the sound generated.

The last part of this excerpt from the *Ursonate* features a progressive scan across part of the vocabulary of the piece, each line taking in a new word at the end and removing one from the beginning, giving a sense of movement across a symbolic textual structure. The listing below in the Haskell programming language attempts to represent this structure, and can be considered “successful” in that when this code is interpreted, the original text is produced. It includes the list of words and an algorithm for progressively scanning across the list. The subtleties of the original—a short first line and a concluding colon—take up the majority of the code, and the result is not much shorter than the original. It is different, however, in that it makes structure explicit that is only implicit in the score, describing more than the surface text. In this respect the code is perhaps closer to the poem as it was spoken than to the poem as written, with the intonation of the reader relating structural information that can only be inferred from the written text. The code is a kind of paralinguistic phrasing, and on interpretation produces its own transcription with this phrasing removed. It oscillates between written and spoken forms.

```
poem = recite sounds
sounds = words "dll rrrrr beeeee bö fümms bö wö tää zää Uu"

recite xs | xs == sounds = (unwords $ take 4 xs) ++ "\n" ++ recite (tail xs)
      | length xs == 6 = slice xs ++ ":\n"
      | otherwise = slice xs ++ ",\n" ++ recite (tail xs)
  where slice xs = (unwords $ take 6 xs)
```

Evidently sound poetry may be nonlexical but is not entirely ambiguous. Schwitters makes this clear in an accompanying text that declares the poem should be performed using German intonation, emphasizing our earlier comments about the shape of the words in the mouth. For without a method of interpreting the text into sound, whether explicit or implied, this could not be considered a sound poem as such. The method of interpretation is external to the text, but at the same time a fundamental part of the experience of the work. The same relationship of code and interpreter is apparent with computer code; a running program exists as a combination of the code with a language interpreter. Indeed it is possible to write one piece of source code that conforms to the grammatical rules of several quite different programming languages, and that behaves differently in each. It is also possible to write source code that when interpreted outputs source code in different languages—and this is a standard process in the interpretation of C code. The example further demonstrates that a computer program can undermine the distinction between its function as a score and its performance, in similar ways to the sound poem (but in quite different ways to conventional musical scores).

This chapter aims to extend this discussion about the relation between spoken language and program code, but in particular to speculate on vocable code and the ways in which the body is implicated in coding practices. Through the connection to speech, programs can be called into action as utterances that have wider political resonance; they express themselves as if possessing a voice.

Coding language

To attempt to understand a “text” like a sound poem requires recognition that all languages consist of closed systems of discrete semiotic elements, and that meaning is organized out of differences between elements that are meaningless in themselves.⁶ Things can only be understood in terms of the organizational structure of which they are a part; there is an abstract system (langue/competence) that generates the concrete event (parole/performance) in semiotic terms.⁷ The extreme of this position is the view that a text is entirely autonomous from the act of writing—writing writes, not writers—as if all writing is active like a computer program or has a kind of agency beyond its inscription.⁸ It is as if every work carries its source code with it through the formal

logic that underpins its behavior, whether emanating from a computer or from a book or speech.⁹ In this respect it speaks for itself.

Grammars

To understand in more detail more how a script performs, one common starting point is the linguistic concept of transformational grammar, derived from Noam Chomsky's *Syntactic Structures* of 1957. Chomsky assumes that somehow grammar is given in advance, hard-wired or preprogrammed: that humans possess innate grammatical competence that is presocial (and one of the controversies of Chomsky's system is its separation of consciousness from the outside social world). Thus he explains the deep-seated rules by which language operates and how principles and processes are set by which sentences are generated. In procedural terms, a natural language can be simply described as a "finite state Markov process that produces sentences from left to right."¹⁰ In mathematics and other formalized systems, the logic adheres to similar grammars that generate their own grammatical sequences. Chomsky's conclusion is that grammar is independent of meaning, and that the separation from semantics is essential to the study of syntactic structure.

Clearly syntax is also fundamental for the structure of a statement in a computer language, such as with the use of parentheses to produce subclauses while retaining the overall flow of logic. Conditional structures such as loops are commonplace in setting out instructions to perform repeated tasks, usually stopped under a certain condition, and loops rest within loops in intricate ways, including parenthesis within parenthesis (as with the appearance of { and }). With loops, simple sentences are able to reproduce themselves recursively, as in the example by Douglas Hofstadter: "The sentence I am now writing is the sentence you are now reading."¹¹ The referent "the sentence" is understandable only within the overall context of the words that make a sentence, this being one of the key structural elements of written language as a whole. Evidently it would be a mistake to think that grammars are simply devices for generating sentences, and much experimental software artwork has been developed with this principle in mind, partly in response to the overreliance on the syntactical aspects of coding and to address issues of meaning production at source.¹² Whereas artists simply had to engage with programming in early computer arts or generative arts practice, the lack of this necessity now (due to the wide availability of scripting languages with libraries of functions) allows for wider issues to be engaged related to semantic and social concerns.

This view also recognizes the wider material apparatus of programming and the many agents of production involved in the process, which would include the engineers who design the machines and the workers who build them as well as the programmers who write the programs.¹³ Meanings are produced through the interactions of these agents at all levels of operation. In "There Is No Software," Kittler

argues that software obscures such operations, and as a result confusion arises between the use of formal and everyday languages. Programming languages have supplemented natural languages, suggesting the need for new literacies that include both natural and artificial languages.¹⁴ These languages form a “postmodern tower of Babel,” as he puts it, and this is why: “We simply do not know what our writing does.”¹⁵ Kittler is partly referring to the ways that graphical interfaces dispense with the need for writing and hide the “machine” from its users. The confusion Kittler refers to is implemented at the level of hardware itself, wherein “so-called protection software has been implemented in order to prevent ‘untrusted programs’ or ‘untrusted users’ from any access to the operating system’s kernel and input/output channels.”¹⁶ If that was at issue in the early 1990s when these words were written, the problem of access to the code that underpins the system hardware is now exaggerated with contemporary developments in service-based computing; now users hardly know at all what their programs do nor have access to them, as they are closed off almost entirely from their machines.

Behind this crucial issue of access to information is the history of the sharing of source code, itself rooted in the history of the UNIX operating system and its precarious position between the promises of the public domain and commercial enterprise, corresponding to the differences between free software and open-source development.¹⁷ The former champions the idea of freedom in resistance to proprietary software by keeping software in the public domain (associated with Richard Stallman and the free software movement), while the latter takes the view that open-source development will lead to better implementation and therefore offers economic benefit (associated with Eric Raymond and his free-market approach). According to Raymond, UNIX is open as it works across different computer platforms, and as such it is the “closest thing to a hardware-independent standard for writing truly portable software.”¹⁸ In supporting multiple program interfaces and flexibility, it provides access to the hidden depths of the machine. Furthermore, it emanates from a “bottom-up” folk tradition in which expertise comes from the culture itself, exemplified in principles of transparency and discoverability, and indeed in the software engineering ethos to “keep it simple, stupid!”¹⁹

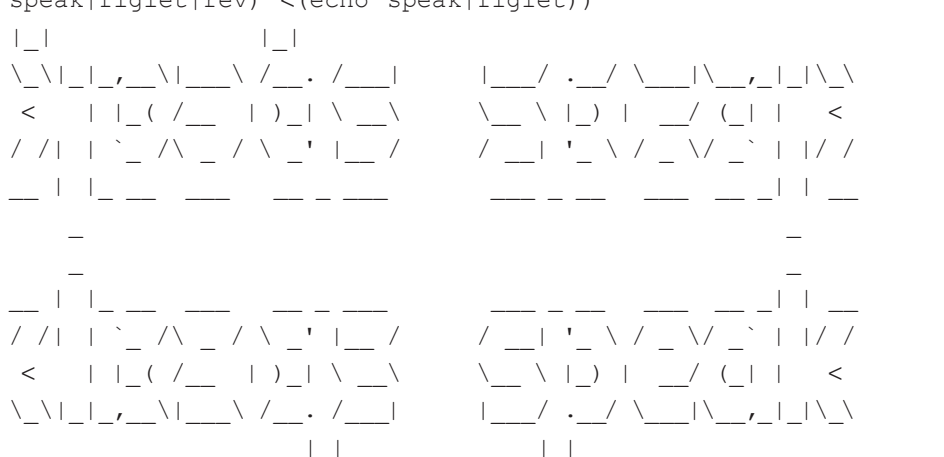
Also with UNIX in mind, but arguing against Kittler’s determinism, Florian Cramer has drawn on terms from Roland Barthes’s *S/Z* to make the distinction between “readerly” and “writerly” texts.²⁰ Rather than the readerly properties of a GUI operating system that hides the underlying program code, Cramer claims the command line user interface of UNIX is writerly, in terms of its openness and in encouraging the reader to become an active producer of “scriptable” code. For Cramer, these writerly aspects of coding break down the false distinction between the writing and the tool with which the writing is produced (and in terms of the computer, between code and data). He cites a 1998 essay by Thomas Scoville, “The Elements of Style: UNIX as Literature,” to insist on the writerly aspects of programming by explaining how writers

use language: “No literary writer can use language merely as a stopgap device with which to compose an artwork that is not in itself language—so, like in a recursive loop, literature writes its own instrumentation.”²¹ This is a kind of writing that writes itself; all writing is a kind of program in this sense.

Notation and execution are collapsed into one thing, as in the Fluxus performance score of La Monte Young’s *Composition 1961 No. I, January I*, “Draw a straight line and follow it,” that Cramer uses as an example of software art (interestingly, an example that does not use a computer).²² Another example might be Alvin Lucier’s composition *I am sitting in a room (for voice on tape)* (1969),²³ a script read by a performer and repeated 32 times as a loop. The score explains the concept like a program: at each iteration, the sound is recorded, played back, and then rerecorded, and hence becomes a representation of the resonance of the room itself as well as the body of the speaker. The instruction is executed, but the work is more about the irregularities of speech (including Lucier’s stutter) and its inherent musicality, to reveal that speech is so much more than the articulation of words.²⁴ But also the words themselves can be said to perform, as they execute their instructions recursively.

The point is that neither the score nor the program script can be detached from its performance, and in this context this is what makes a program like speech. The following example renders the word “speak” as ASCII characters, which are reversed horizontally and vertically to create a symmetrical arrangement. However, the character forms somewhat mask the symmetry from the human eye, and the reader might need to turn to the code to verify the act of speech.

```
example@speakingcode:code$ cat <(paste <(echo
speak|figlet|tac|rev) <(echo speak|figlet|tac)) <(paste <(echo
speak|figlet|rev) <(echo speak|figlet))
```



Notation

Rather than include writing systems such as program code, most critical work in this area has contrasted spoken and written forms of language. But much of the subjective

expression of speech is hidden when it is transcribed in written form, and there are many simple techniques, which we tend to take for granted, that try to capture the special qualities of speech with various degrees of success. Approaches to quotation are particularly interesting in this connection, and what became the common practice of using the markup convention of quotation marks to distinguish the voice of the character in the text from that of the writer. Quotation or exclamation marks emphasize that the words are not simply printed but are spoken in written form. Similarly in coding practices, “commenting out” is used to distinguish the voice of the program from that of the programmer.

```
# This is the voice of the programmer
echo "This is the voice of the program"
```

Yet the program interpreter only listens to the program and not the programmer; and significant comments, existing as natural language read by programmers, are ignored by interpreters, as with the numerous instances of the word “fuck” that can be found in the comments of the Linux kernel. Here is one example:

```
alex@quantas:/tmp/linux-3.0-rc5$ find -name '*.c' |xargs grep -i fuck
./lib/vsprintf.c: * Wirzenius wrote this portably, Torvalds fucked it
up:-)
```

Comments are not the only aspects of programs ignored during interpretation, as the majority of the text may be discarded and optimized away in the early stages. Carefully chosen variable names are replaced with numbers, and explanatory two-dimensional spatial arrangements are compressed into contiguous, single-dimensional sequences. Nor is the color highlighting seen in programming interfaces saved to source files. Together these aspects of programs are known as “secondary notation,” a pejorative term suggesting that code’s primary purpose is to be executed by a computer, and only secondarily to be understood by a human. However, in terms of human understanding, the names given to abstractions, their spatial arrangement, and their appearance on the page are as important as the logic contained within. If secondary notation is removed from a program then confusion results; and with human aspects of code removed, all that remains are monotonous sequences of syntax, bereft of meaning at the level of programming or wider human understanding of the processes at work.

The interpreter ignores natural words, but they still remain in the original source code, giving the opportunity for programmers to express themselves beyond the internal actions of the computer. In his technical manual “Oracle PL/SQL: Guide to Oracle8i Features,” Steven Feuerstein controversially placed technical examples in the context of war crimes.

```
CREATE OR REPLACE PROCEDURE update_tragedies (warcrim_ids
IN name_varray, num_victims IN number_varray) IS BEGIN
```



```
FOR indx IN warcrim_ids.FIRST .. warcrim_ids.LAST LOOP
UPDATE war_criminal SET victim_count = num_victims (indx)
WHERE war_criminal_id = warcrim_ids (indx); END LOOP; END;25
```

This demonstrates the effectiveness of secondary notation, but as Oracle is used primarily in large-scale businesses, including much of the global financial infrastructure, Feuerstein's examples of excessive CEO compensation, union busting, and war crimes in a mainstream published book about technology were met with an outcry.²⁶ A similar approach to combining technical and political commentary can be seen in the following extract from Harwood's *Perl Routines to Manipulate London*, porting William Blake's poem "London" (of 1792) into the Perl programming language.²⁷ In both the original version and the adaptation, statistics and the modulation of populations are used for social comment:

```
local @SocialClasses = qw(
    RentBoy YoungGirl-Syphalitic-Inoculator
    CrackKid WarBeatenKid ForcedFeatalAbortion
);
```

The interpreter will instead "perceive" an abstract, tokenized version, somewhat like:

```
t_local @v1 = [t1, t2, t3, t4, t5]
```

Much of the meaning of the program and how it might be understood is simply ignored by the computer. The program may extract the original text "RentBoy" from the pointer "t1," but the context of the container name "SocialClasses" will have been lost, and the program itself will have no notion of what the word "RentBoy" means.

In terms of the literate application of the programming language Perl (developed by Larry Wall in 1987), Harwood is extending an established aesthetic practice referred to as "Perl poetry," the practice of writing poems that can be compiled as Perl code, made possible by the semantic qualities of Perl. Wall was emphasizing the point that code has expressive qualities, and the need for programmers to "express both their emotional and technical natures simultaneously."²⁸ The technique is effective as it manages to combine both primary and secondary notation, both syntactical and semantic registers, and the tension between human and machine interpretation is made painfully evident (so much so that you might imagine yourself to hear the screams).

```
local %InfantsScreamInFear;
# WoeOfEveryMan is the main method for constructing the
# InfantScreamInFear structure:
# If we find that say that the hash %{RentBoy} is defined
# with values then we add this to the hash %{InfantScreamInFear}
```



```

sub WoeOfEveryMan
  foreach my $Class (@SocialClasses){
    warn "New class = $Class\n";
    $InfantsScreamInFear->{Class} = %{$Class} if {Class};
  }
}

# In every cry of every Man,
# In every Infants cry of fear,
# In every voice: in every ban,
# The mind-forg'd manacles I hear

```

The interplay of the body of the code, the programmer's comments, and the human-machine reader express how hardware and software, text and code, are "embodied." The body is of course registered in the content (in the codework itself), in the narrator's body (the comments and secondary notation), but also in the bodies of all those humans involved in the production process, including the reader's body. This embodiment issue is something that N. Katherine Hayles asserts in *Writing Machines*: "Literary texts, like us, have bodies, an actuality necessitating that other materialities and meanings are deeply interwoven into each other."²⁹ Imaginative and political possibilities emerge from this recognition.

Indeterminism

As in speech, it is clear that there is a dynamic relation between what exists and what is possible, between past and present states, between concealing and revealing possibilities corresponding to the layers within computation itself. Part of this relates to digitalization as opposed to the use of analog systems, in which elements are not continuous but discrete units or objects. Hayles clarifies this sense of structure: "The act of making discrete extends through multiple levels of scale, from the physical process of forming bit patterns up through a complex hierarchy in which programs are written to compile other programs."³⁰ The hierarchical aspect is important in understanding how code is organized in deliberate ways, and thus how it can also be understood and theorized. Hayles explains how human speech is analog, emanating from a continuous stream of breath that forms phonemes that are more discrete. Writing makes these units yet more discrete through inscription. Programmers oscillate between these modes at the level of the human-computer interface, translating between machine behavior and human perception,³¹ with the human made somewhat computerized and the computer somewhat anthropomorphized. Compiling and interpreting, although similar, are also distinct across human and machine platforms. Therefore it would be a mistake to simply conflate natural language and human intelligence with program code. This is also an issue that Ong refers to when he argues that although some computer languages resemble human languages, they are unlike

them as they do not emanate from the unconscious but directly out of consciousness.³² Although this may or may not be strictly the case, the point is that both modes interact in ways we do not fully comprehend.

The human dimension is registered in a multiplicity of ways in communication systems, as for instance in conceptual metaphor theory,³³ in which meaning can be understood as largely structured through spatial relationships relative to the body, unbound by syntax. Speech grounds language in the voice, and orientational metaphor grounds semantics in the body. It follows that computer software cannot have access to systems of meaning without at least some kind of reference to bodily relationships. This embodiment issue has become crucially important for understanding the interactive experience of computation, extending the limits of HCI (human-computer interaction) traditions.³⁴ Similarly emergent fields such as “tangible computing” and “social computing” strive to account for the ways in which humans interact with physical objects that are increasingly “augmented” with computational processes and interactions with the environment.³⁵ Programmers bring bodily meaning to their work by applying models of human perception, and by trying to account for the ways that other social bodies are drawn into the process of meaning production.

In opposition to this tendency, Edsger Dijkstra understands the world of programs and of people as totally separate. He thinks that “whereas machines must be able to execute programs (without understanding them), people must be able to understand them (without executing them). These two activities are so utterly disconnected—the one can take place without the other.”³⁶ Yet computers embedded in the fabric of the human environment, increasingly with actuators and reacting sensors, require the encoding of interactions that are meaningful to humans.³⁷ Even computers have some sense of their own conditions, as for instance in the way that a standard laptop can now detect when it is being dropped, or getting too hot. Perhaps the most radical physical action that a computer can carry out is simply turning itself off; not quite self-immolation, but all transitory data is lost. The following example *monitor.upstart* is a configuration file for Linux systems. When the computer starts up, it checks the temperature, and if this is higher than the long-term global average of 14 degrees Celsius, it shuts the computer down again. As the operation of the computer itself creates heat, and its feed of electricity is dependent on the production of greenhouse gases, it expresses interaction on a global scale.

```
start on startup
exec if [ `perl -pe 's/\D+(\d+).*/\1/' \
        /proc/acpi/thermal_zone/*/temperature` -gt 14 ] \
    then shutdown now \
    fi
```

As software becomes ubiquitous it becomes ever more connected to external processes, and programs no longer encode pure logic but human social behavior too. Such

approaches recognize that systems are embedded in larger language systems where meanings are produced through social practices. In other words, computer programs have bodies in the sense that other materialities and meanings are deeply interwoven, and these necessarily exist as part of wider social relations.

Coding speech

Clearly speech emanates from the body; but even more fundamentally voice has been understood as a precondition of consciousness itself. Johann Gottfried Herder, writing at the end of the eighteenth century, helped establish that voice and speech were distinct terms (even if in dubious ways that marginalized certain types of people outside of “normal” language).³⁸ To Herder, marking a distinction from other animals, speech happened when the voice took on the inflection of the human institution of language.³⁹ Thus, once the human voice “spoke,” it was understood to signify higher human reason and a world of signification, as opposed to a mere voice that might simply comprise a collection of nonlexical vocables. Yet it would seem that computer speech simulation is rather too rational, further complicating the distinctions.

Machines

So what would it take for a machine to speak convincingly? The problem of how to account for the voice is a recurring theme, indicative of the energies that lie outside of rational systems of representation and the human fascination with simulations of the strange noises that lay at the root of language. In the eighteenth century, Wolfgang von Kempelen’s speech machine was allegedly the first that allowed the simulation not only of some speech sounds but of whole words and short sentences. Initiated in 1769, it was further developed in response to a call from the Imperial Academy of Arts and Sciences in St. Petersburg in 1780 offering a prize to construct a machine that could reproduce vowel sounds and explain their properties.

In his 1791 book *Mechanismus der menschlichen Sprache nebst Beschreibung einer sprechenden Maschine* (The mechanism of the human speech with the description of a speaking machine), Kempelen describes “die Sprech-Maschine” (the speaking machine) in close detail so that others might be able to reconstruct and improve it (thus distributing his plans under the terms and conditions of what we would now consider to be open-source hardware).⁴⁰ The machine comprised bellows like lungs to pump air into the voice box simulating a windchest, while a reed controlled the release of air alongside movable parts corresponding to lips, palate, tongue, and nostrils. The mouth was modified by hand to produce speech in combination with a series of valves. The speech machine was presented in a format that invited the public to suggest some words to be repeated by the machine. At the end, the machine was explained in detail, demonstrating that the intention was not simply to provide a mystifying spectacle

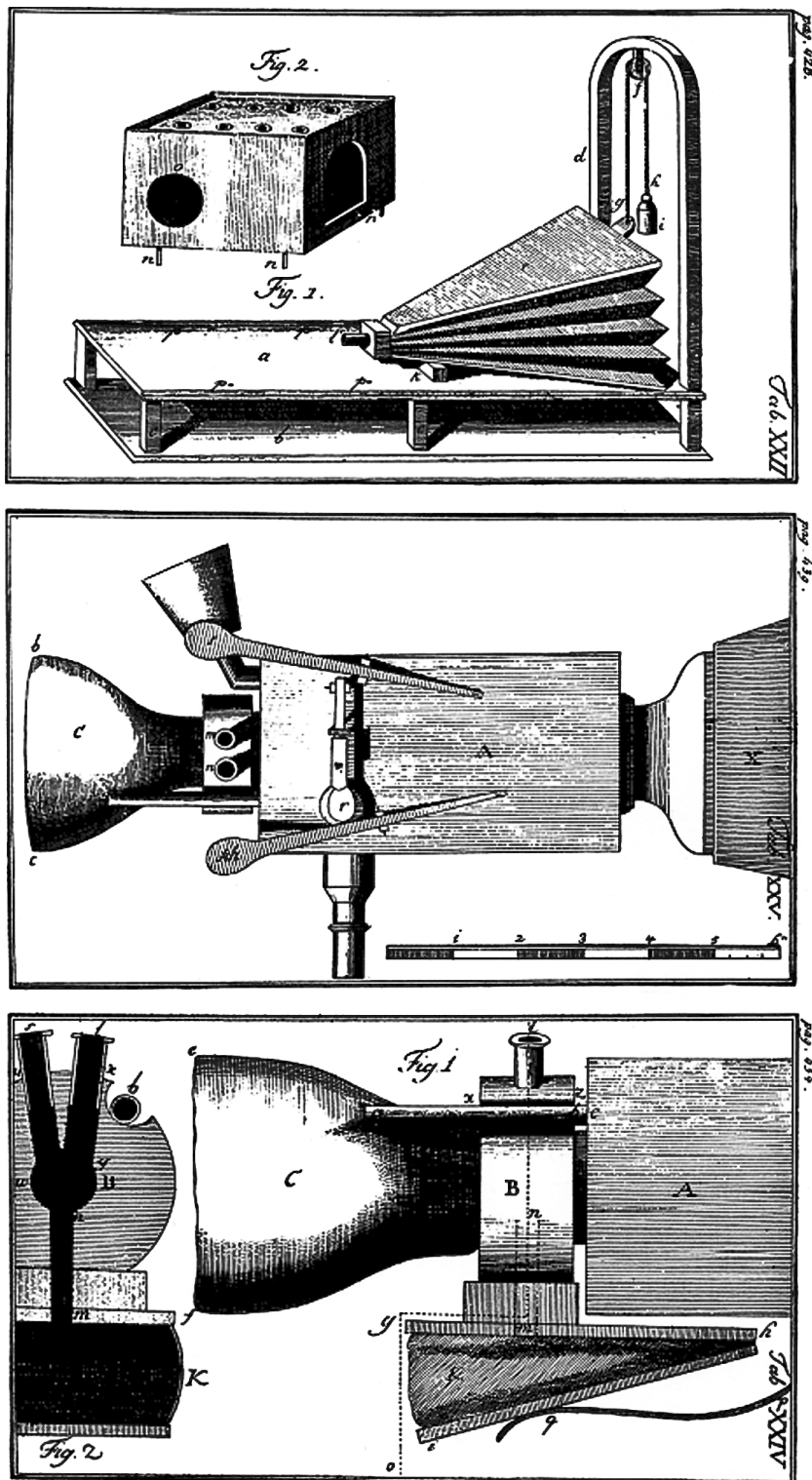


Figure 1.1

Wolfgang von Kempelen, "Sprech-Maschine" (1791), first developed in 1769. Image in public domain.

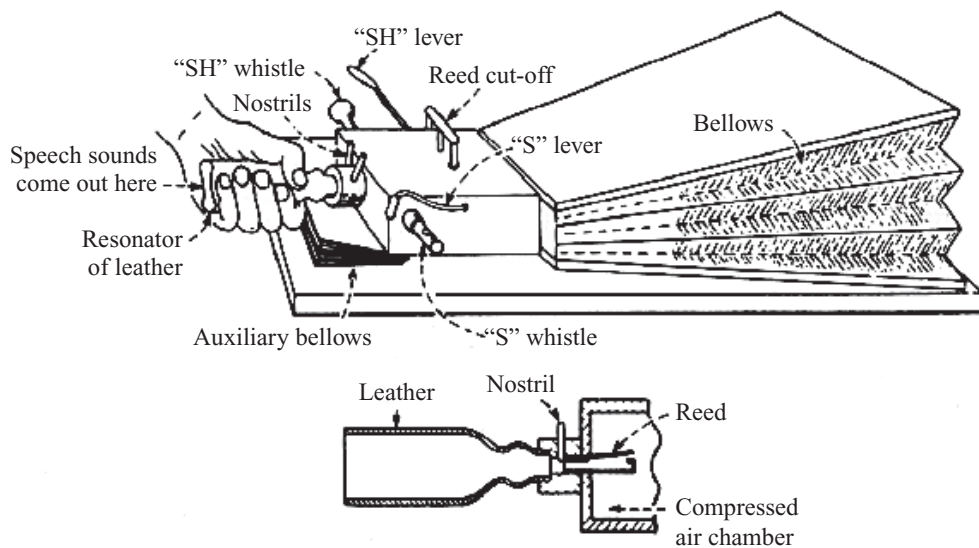


Figure 1.2

Reconstruction of Kempelen's talking machine, attributed to Sir Charles Wheatstone (1879). Image in public domain.

but to be of educational value, and more concretely, as Kempelen indicates, to give the deaf an instrument by which to produce speech. The idea was to simulate a voice in its fullest sense, following principles of free expression, to give voice to those without a voice, following in the spirit of what Kempelen refers to as "the basic tenets of society."⁴¹ Significantly the machine did not simply reproduce the human speech organs but also attempted sound synthesis.⁴²

There are far too many other historical attempts to capture speech to mention; but for the purpose of the argument being developed, it is worth stressing the enduring fascination of a machine that could capture human qualities and the many attempts to move beyond the limitations of the Latin alphabet. Phonography, developed in the early nineteenth century, was one such technique, in which each language sound was given a shorthand mark, Isaac Pitman even claiming (in 1854) his phonetic shorthand to be an exact "picture of speech" itself. Others who concentrated on the physiology of speech, such as Alexander Melville Bell (in 1849), described the "actual movements of the organs of speech"; Bell subsequently referred to his work as "visible speech," based on a universal notation system able to reproduce every dialect and language, the symbols of which were not alphabetical but physiological.⁴³

These examples are among the myriad ways in which inventors of machines have tried to represent the complexities of what is spoken. This is also something that written languages have attempted using symbols; the Chinese language makes a notable example with its vast number of characters (the K'anghsi dictionary of Chinese in AD 1716 contained 40,545 characters and took about twenty years to learn,

something that a computer might do more efficiently).⁴⁴ Another relevant example is Hangul, the main alphabet in use in what are now North and South Korea. Hangul was invented in the fifteenth century; not only is it phonetic, but the shapes of the consonants are based on the shape the mouth makes when the corresponding sound is made.⁴⁵ This grounding of symbols in the body also reinforces the idea that language might be recursive, undermining the distinction between the spoken word and the way it is produced (in a similar way to literature writing its own instrumentation, as previously mentioned).

These few examples also serve to emphasize the inadequacies of the Latin alphabet to capture dialects and phonetic diversity: making “the art of recording speech almost impossible” according to George Bernard Shaw, a position that he exemplifies in his play *Pygmalion*.⁴⁶ In the narrative, the main character Eliza Doolittle, the cockney flower girl, is the subject of a bet by a professor of phonetics that he can teach her gentility through the acquisition of proper speech. The title makes reference to the Greek myth in which Pygmalion, a sculptor, falls in love with a statue he carves, and Venus grants it the breath of life.⁴⁷

The Pygmalion myth stands as a useful analogy for the idea of breathing life into machines. The chatterbot Eliza produced by Joseph Weizenbaum between 1964 to 1966, named after Eliza Doolittle, is a disarmingly simple example based on similar motivations: to simulate a believable exchange with a human conversant. It uses primitive natural-language processing to simulate a conversation with a therapist, producing human-like responses by implementing a simple script based on key words and language patterns through which responses are generated. Here is an example:

```
I am the psychotherapist. Please, describe your problems.
> Hello world.
Why do you say hello world?
> It's the orthodox way to begin when using a new computer language.
You have your mind on computers, it seems.
> Sometimes I think I am a computer.
You think too much about computers.
> Thinking computers, thinking too much about computers?
You should try taking your mind off of computers.
> Well you seem to dwell on the subject yourself too.
What makes you believe that?
> The evidence is in the above text!
Maybe your life has something to do with this.
```

Without involving complex algorithms, it can generate responses that appear to make some degree of sense. Yet it is interesting to note how the illusion of conversation follows an extremely reductive model of human expression, and the fantasies of machine intelligence seem to be similarly founded on reductive logic. At the same

time, natural-language processing programs and other chatterbots offer good examples of the speechlike procedures mentioned thus far, as well as the apparent impossibility of duplicating actual speech.

Intelligence

To demonstrate believability, a machine would be required to possess some kind of intelligence that reflects the capacity for human reasoning, in parallel to turning mere voice sounds into proper speech that expresses human gentility. In a paper of 1950, “Computing Machinery and Intelligence,” Alan Turing made the claim that computers would be capable of imitating human intelligence, or more precisely the human capacity for rational thinking. He set out what become commonly known as the “Turing test” to examine whether a machine is able to respond convincingly to an input with an output similar to a human’s.⁴⁸ The contemporary equivalent, CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart), turns this idea around, so that the software has to decide whether it is dealing with a human or a script.⁴⁹ Perhaps it is the lack of speech that makes this software appear crude by comparison, as human intelligence continues to be associated with speech as a marker of reasoned semantic processing.

In his essay “Minds, Brains, and Programs” from 1980, John Searle refutes the Turing test because machines fall short in understanding the symbols they process. His observation is that the syntactical, abstract or formal content of a computer program is not the same as semantic or mental content associated with the human mind. The cognitive processes of the human mind can be simulated but not duplicated as such. Searle develops his thought experiment known as the “Chinese Room argument” as follows: “Suppose that I’m locked in a room and given a large batch of Chinese writing. Suppose furthermore (as is indeed the case) that I know no Chinese, either written or spoken. . . . To me, Chinese is just so many meaningless squiggles.”⁵⁰ Given linguistic instruction, Searle imagines that he becomes able to answer questions that are indistinguishable from those of native Chinese speakers, but insists, “I produce the answers by manipulating uninterpreted formal symbols. As far as the Chinese is concerned, I simply behave like a computer; I perform computational operations on formally specified elements. For the purposes of the Chinese, I am simply an instantiation of the computer program.”⁵¹

Searle’s position is based on the linguistic distinction between syntax and semantics as applied to the digital computer or Turing machine as a “symbol-manipulating device,” where the units have no meaning in themselves (a position that follows from semiotics). Even if it is argued that there is some sense of intentionality in the program or a degree of meaning in the unit, it is not the same as human information processing, and this sense of agency is what Searle calls “as-if intentionality.” In the Chinese Room, Searle becomes an instantiation of a computer program, drawing on a database

of symbols and arranging them according to program rules. The point to be stressed is that formal principles alone remain insufficient to demonstrate human reason; the Chinese language is a particularly complex kind of invention that challenges human capabilities (so that machines might usefully supplement them), but it also combines images and speech in ways that confound formal logic.

Also referring to the Chinese spoken by the “living reading machine,” Ludwig Wittgenstein clarifies that a genuine speech act is more than just a series of sounds arranged in sequence, or talking without thinking. To him, “The sentence, as it were, plays a melody (the thought) on the instrument of the soul.”⁵² If the machine is considered to lack a soul, it is because it manipulates symbols but does not understand them or produce meanings in them. Wittgenstein likens this to the workings of a pianola, translating marks and following patterns rather than expressing intentionality as such: “The living reading machine produces as output solutions to arithmetical problems, texts spoken aloud, proofs of logical theorems, notes played on a piano, and suchlike. These ‘machines’ may be born, like an idiot savant, or trained.”⁵³

Although there are similarities with Searle’s position and his description of “as-if intentionality,” Wittgenstein’s argument is different in that when a word is spoken it refers to “the whole environment of the event of saying it. And this also applies to our saying that someone speaks like an automaton or parrot.”⁵⁴ The meanings of words are not derived from an inherent logical structure alone that manipulates symbols into particular sequences (like a program), but also from their social usage. This reveals a problem with the Searle experiment in that it is based on a description of the workings of the CPU (central processing unit) and not of the larger environment that has been stressed in earlier sections of this chapter, including the body and social relations.

Yet to Searle, the argument is posed rather differently. To him, in this specific case, the distinction between humans and machines is obsolete, as humans are already biological machines. It is possible for an artificial machine to think in principle, but only in simulation; for it to duplicate actual thought, it would also need to reproduce consciousness itself.⁵⁵ This may have proved impossible thus far, but the problem for Searle lies in the fact that computational processes do not simulate machines far enough in terms of energy.⁵⁶

Embodiment

What is required to duplicate the human speech act more convincingly is a conscious body, and a model of human perception that is socially grounded sufficiently to be able to derive meaning and thus stake a claim of intelligence. For now, the position remains that machines still do not simulate human intelligence particularly convincingly, but it is also clear how willing humans are to anthropomorphize machines, especially those that appear to demonstrate the ability of speech. Speech seems to

encapsulate, indeed embody, the complexities of meaning production, and therefore it comes as no surprise that there are repeated attempts and failures to simulate its procedures and melodies.

The various attempts to synthesize speech using computer systems seem to stress the point, although most attempts have involved taking recordings of small samples of real speech and concatenating them together. The sounds sampled are those made not when the vocal tract is held in a particular position (phonemes), but when the tract moves between two positions (diphones). An alternative to this, and closer to the principles of the speaking machine, is articulatory speech synthesis, in which a simulation is run of the lungs pushing air through the vocal cords, their oscillations producing air pressure waves shaped by the tongue, nose, teeth, and lips, each contributing their own smacks, grunts, whistles, and so on. For instance, Praat software, a speech analysis tool that can help visualize spoken language, attempts to simulate speech patterns.⁵⁷ In order for the synthesizer to say something, the articulatory synthesis package explains a number of necessary steps in biotechnical terms:

We are going to have the synthesizer say [əpə]. We need a Speaker and an Artword object.

1. Create a speaker with Create Speaker . . . from the New menu.
2. Create an articulation word of 0.5 seconds with Create Artword. . . .
3. Edit the Artword by selecting it and clicking View & Edit.
4. To set the glottis to a position suitable for phonation, use the ArtwordEditor to set the *Interarytenoid* activity to 0.5 throughout the utterance. You set two targets: 0.5 at a time of 0 seconds, and 0.5 at a time of 0.5 seconds.
5. To prevent air escaping from the nose, close the nasopharyngeal port by setting the *Levator-Palatini* activity to 1.0 throughout the utterance.
6. To generate the lung pressure needed for phonation, you set the *Lungs* activity at 0 seconds to 0.2, and at 0.1 seconds to 0.
7. To force a jaw movement that closes the lips, set the *Masseter* activity at 0.25 seconds to 0.7, and the *OrbicularisOris* activity at 0.25 seconds to 0.2.
8. Select the Speaker and the Artword and click Movie; you will see a closing-and-opening gesture of the mouth.
9. Select the Speaker and the Artword and click To Sound . . . (see Artword & Speaker: To Sound . . .).
10. Just click OK; the synthesis starts.

This example may express something of the energy that Searle identified as missing in previous simulations, but despite some promising results the vocal tract has so far proved too complex to fully understand in order to produce plausible speech, certainly in real time. Even the introduction to the Praat software admits its limitations: “Praat is a powerful tool from which you can learn a lot on your own, but if you want to improve your spoken English, we recommend you work with a qualified ESL instructor.”

Artificial voices largely remain based on concatenative synthesis, in which emotional aspects are impossible to fully control and simulate. Articulatory synthesis stress, for instance, might be simulated as physical tightness, whereas with concatenative synthesis an actor's voice would have to be recorded making every combination of possible sounds while pretending to be stressed. This thinking can be seen in the everyday speech synthesis software that is now regularly bundled with operating systems, later combined with a range of sample voices and most recently with authentic-sounding breaths between sentences.⁵⁸ Nevertheless the concern here is not with speech synthesis per se, but rather with the impulse for the various attempts to simulate speech, which have largely failed despite massive increases in computational speed. The human nervous system and brain remain a massively concurrent system, and modern software struggles to run more than one interacting process at a time. In plain speech it is possible to pronounce two things at the same time, and this is something that the experiments of Schwitters would affirm. The enduring problem of how to invent a machine that can replicate the complexities of the human mouth and vocal cords attests to the power of speech but also to the continued difficulties of its duplication.

Code act

In charting the interactions of speech, writing, and the actions of code (and borrowing from Grusin and Bolter's idea of "remediation"), Hayles explains how new practices borrow and reinterpret previous technologies recursively.⁵⁹ Speech and writing both influence program code and are changed by program code. Taking inspiration, or rather a point of departure, from Ferdinand de Saussure (on speech) and Jacques Derrida (on writing), Hayles aims to examine the conceptual system in which code is embedded and the activity of coding. The significant fact for Hayles is not whether speech is subordinate to writing (the position that Derrida takes) or whether writing is derived from speech (de Saussure's position), but that code exceeds both in addressing both humans and machines. She is drawn more to the technological materialism of Kittler and his attention to the detail of code, with his insistence on the central importance of changes in voltage, treating signifiers as voltages and the signified as "interpretations that other layers of code give these voltages."⁶⁰ Further layers of translation from machine code to higher-level languages result in a chain of relations between signifier and signified based on the ability of the machine to recognize the difference between zero and one. Hayles considers code to determine actions with little ambiguity, although she does admit to the existence of noise with higher-level languages. For her purpose, she has problems with de Saussure's "dematerialized view of speech" and Derrida's "linguistic indeterminacy,"⁶¹ as neither seems adequate to describe computational processes and actions. Yet what Hayles appears to overlook,

in her reliance on Kittler's technomaterialism, is her earlier insistence that machines have bodies too. If code undermines the distinctions between speech and writing and exceeds them, it is because it is a special kind of human-machine writing that makes things happen; in other words, it acts like speech.

Speech act

In addition to the work of Searle and Wittgenstein, "speech act theory" derives from John Langshaw Austin's *How to Do Things with Words* (1955), delivered as a series of lectures examining ordinary linguistic usage and utterances.⁶² In the first lecture, Austin establishes that as well as providing descriptions, questions, and commands, sentences can do something as opposed to just saying something: "to utter the sentence . . . is not to *describe* my doing of what I should be said in so uttering to be doing or to state that I am doing it: it is to do it."⁶³ A sentence or utterance of this type he calls a "performative," to indicate how it performs an action. In addition, Austin introduces the term "operative" to indicate how saying something can make it happen, as for instance in the practice of law, where an utterance serves to make an instrumental effect.

His concern was to examine in what ways to say something is to do something, how *in* saying something we do something, and how *by* saying something we do something. In saying something, Austin writes, we may perform "locutionary acts" (in providing the meaning of something, thus making a certain reference to something), "illocutionary acts" (in saying something with a certain force, such as informing, ordering, warning), and "perlocutionary acts" (in achieving certain effects by saying something, such as persuading someone of something), among other types.⁶⁴ The perlocutionary act marks a distinction between the action and its consequences, and in this sense the consequences almost become the act itself, as with the following example in which a light exists as a diode built into the hinge of a laptop, to inform us of computational activity. By entering the operating system via the command line interface, the program represents the light as a file, and when it sends the string "0 blink" to that file, the light flashes. In this sense, it speaks through the file to the light.

```
# In the assertion "I speak,"
# I do something by saying these words;
# moreover, I declare what it is I do while I do it
echo "0 blink" > /proc/acpi/*/led # flash a light
```

Speech acts come close to the way program code performs an action, like the instruction addressing the file. Programs are operative inasmuch as they do what they say, but moreover they do what they say at the moment of saying it. What distinguishes the illocutionary act is that it is the very action that makes an effect: it says

and does what it says at the same time. Such utterances are not conventional but *performative*. In the analogy to code, what becomes useful is the recognition that speech produces an enormous variety of articulations that resist computational analysis, as described in the previous section. As the many attempts to build speech machines also indicate, speech is highly complex and not simply reducible to instrumental techniques and algorithms. The physiognomy of breath and muscle control, the tongue, lips, and larynx, and other cultural factors of language and dialect all add to the difficulty of simulation. Any spoken utterance is always locutionary in that it is an act of saying certain words, and of making movements with vocal cords and breath. Human gestures, such as those made by hands (aside from sign languages), can also be considered performative speech acts.⁶⁵ Adding a prosodic inflection to speech can give it a totally different meaning, perhaps only intended for certain listeners or contexts. There is necessarily a connection between saying something and physical action in a general sense. However, actions do not happen simply as a consequence of locution, as these further depend on outside factors.

Vocable Synthesis

Citing Schwitters's *Ursonate* as inspiration, Alex McLean's *Vocable Synthesis* (2008) is a system for improvising polymetric rhythms with vocable sounds.⁶⁶ He refers to the common use of vocable words in various musical traditions too: words (whether written, spoken, or sung) that are used for their sounds but not for wider meaning production. Vocables may be used to describe sounds, for example in the *bol* syllables of Indian classical music and the chanting or "mouth music" of the *canntaireachd*, the ancient Gaelic method of notating classical bagpipe music. Although far from standardized, *canntaireachd* operates through a combination of definite syllables (vowels represent the notes and consonants the embellishments, but this is not always the case) to enable the learner to recollect the tunes and easily transmit them orally. The following example of written *canntaireachd* is the ground (*urlar*) of *The Cave of Gold*, attributed to Donald Mor MacCrimmon, circa 1610:

Heinbodrie heunbodro, heinbodrie bitri betre, heinbodrie heunbodro, heinbetre odrierarierin,
(repeat)

Heinbitri hereinve, heinbodro heororo, heinbitri hereinve, heinbetre odrierarierin,
Heinbitri hereinve, heinbodro heororo, heinbodrie heunbodro, heinbetre odrierarierin.⁶⁷

In *Vocable Synthesis*, the sequence of letters making up a word is translated to a sequence of movements within a simulation of a drumstick hitting a drum skin. For example, the vocable word *kopatu* translates to "Hit loose, dampened drum outwards with heavy stiff mallet, then hit the middle of the drum with a lighter mallet while tightening the skin slightly and finally hit the edge of the skin with the same light mallet while loosening and releasing the dampening."⁶⁸ With vocable symbols, the

idea is that the user/performer relates complex movements to his or her own voice, and therefore is arguably better able to intuitively understand and manipulate them musically. In this sense, the symbols speak to us. Below is an example session from the logs of *Babble* (2008),⁶⁹ an online version of *Vocable Synthesis*, showing a common pattern of the user writing text which, in response to the unconventional sounds that result, degrades through a series of edits into nonlexical rhythmic structure. As with *canntaireachd*, the idea is that the user is able to relate this to their voice, and therefore to intuitively understand and manipulate the symbols.

```
hola tarola
you don't know but that's ok, you might find me anyway
oooooooo uuu uiuiuu aeaeae youaeae
ololololoolo ululu uiluilulu alelalealele ylululalelele
```

The user interface for *Vocable Synthesis* is built upon the GNU Readline library that allows editing of text in the UNIX command line (and its writerly qualities). The performer types a rhythm as a line of text, rather like a score for sound poetry. The sound itself is rather difficult to describe; you have to hear it—and of course it is performed in real time, like *Ursonate* itself. The program performs; and this is better considered to be a performative utterance or articulation, as it is not merely a demonstration of complex auditory signals. Through the analogy to speech, some core issues are also expressed that have been discussed in the chapter thus far, such as the poetic aspects of code and the ways humans communicate their understanding of code in terms of conversational forms and through social relations,⁷⁰ not least when it is performed as part of a live-coding event.

Excess

Although programming languages are clearly not spoken as such, they express particular qualities that come close to speech and even extend our understanding of speech. Computer code has both a legible state and an executable state, namely both readable and writable states at the level of language itself. This is precisely the point to stress in considering code for its speechlike qualities. The act of coding might be seen as the translation between a problem expressed in human terms of speech and one expressed in a way the computer can interpret, between ambiguous and complex expression and formal logic—or between loose and strict thinking. Clearly, performative utterances are linked to actions in a general sense, but, following Austin, it is interesting to think about how utterances express an internal act, almost like an intention to act: “the outward appearance is a description, true or false, of the occurrence of the inward performance.”⁷¹ Yet the force of the utterance is only understood in the context of its totality, the “total speech situation” as Austin calls it, inasmuch as it exceeds itself. This is not simply a question of understanding the

context of the speech but of its excess, where context is lost or turns in on itself recursively.⁷²

Confronting the notion that machines merely do what they are programmed to do, the understanding of programming as a performative speech act extends the unstable relation between the activities of writing, compiling, and running of code as a set of interconnected actions (as the practice of live coding demonstrates so well). Indeed, saying words or running code or simply understanding how they work is not enough in itself. What is important is the relation to the consequences of that action, and that is why the analogy of speech works especially well—although ultimately the phrase “speech act” may not be sufficient for the purpose.⁷³

The various attempts to capture the voice in speech machines and software reveal the futility of the endeavor. In resisting the forces of rationality, it challenges normative communication and synthetic technologies of the general economy, resonating more with the notion of excess.⁷⁴ That the voice cannot be fully captured by computation is very much the point, as it provides inspiration for new ways of working with code and examining “code acts,” to reveal other possibilities and motivations.