Series Foreword

Software is deeply woven into contemporary life—economically, culturally, creatively, politically—in manners both obvious and nearly invisible. Yet while much is written about how software is used, and the activities that it supports and shapes, thinking about software itself has remained largely technical for much of its history. Increasingly, however, artists, scientists, engineers, hackers, designers, and scholars in the humanities and social sciences are finding that for the questions they face, and the things they need to build, an expanded understanding of software is necessary. For such understanding they can call upon a strand of texts in the history of computing and new media, they can take part in the rich implicit culture of software, and they also can take part in the development of an emerging, fundamentally transdisciplinary computational literacy. These provide the foundation for Software Studies.

Software Studies uses and develops cultural, theoretical, and practice-oriented approaches to make critical, historical, and experimental accounts of (and interventions via) the objects and processes of software. The field engages and contributes to the research of computer scientists, the work of software designers and engineers, and the creations of software artists. It tracks how software is substantially integrated into the processes of contemporary culture and society, reformulating processes, ideas, institutions, and cultural objects around their closeness to algorithmic and formal description and action. Software Studies proposes histories of computational cultures and works with the intellectual resources of computing to develop reflexive thinking about its entanglements and possibilities. It does this both in the scholarly modes of the humanities and social sciences and in the software creation/research modes of computer science, the arts, and design.

The Software Studies book series, published by the MIT Press, aims to publish the best new work in a critical and experimental field that is at once culturally and technically literate, reflecting the reality of today's software culture.

Foreword: Software as a Mode of Thinking—An Introduction

John Rajchman

The modern digital computer is the invention of two distinguished mathematicians, Alan Turing and John von Neumann, working in the heyday of a rich debate about numbers and logic and a grand search for the "laws of thought," which they then tried to introduce—perhaps it would be better to say "translate"—into the workings of a new kind of machine, the computer.

The story of this invention has often been told, that of Bletchy Park and The Institute for Advanced Study. Born of the urgencies of war, often elaborated in secrecy in government facilities against a formidable foe, and mobilizing its own science-tech sector, the invention would assume new forms after the war. It would become part of an ever-expanding "military industrial complex," with us now as much as ever, with our giant global Internet companies, surveillance, hacktivism, cybersecurity, smart cities, and infrastructures. In the process, "platforms" themselves would pass from mainframe to PC to smartphone, increasing in speed, efficiency, and reach, and leading to the operations of our great number-crunching algorithms in finance, politics, and social media. The invention of the computer by these two great mathematicians, carried on in military secrecy, in short, has led to an enormous complex in government and economics alike, touching on many aspects of the ways we think and live.

But what role did actual programming play in this history? How did the "translations" of various activities into this complex itself evolve, assuming new forms and functions? What role might programming yet play in the matter of "digital intelligence" today? Such is the complex problem this new study of the origins and nature of software sets out to raise and, in the first place, to formulate. How can we do the history of software itself: What exactly is it? How should we study it? In particular, what ever happened to the great logicist dream of discovering "laws of thought," which inspired Turing and von Neumann, extended at the time in striking ways by Gödel and Hilbert? In what ways did this logicist background help foster a picture of thinking or

"smartness" itself, conceived as a matter of following rules in a finite number of steps, independent of human interaction of any kind—a picture still very much alive today in digital culture, popular as well as more sophisticated? What would it mean to see programming instead, from the start, as belonging to a different, more materially rooted history—more like cooking up ways of doing things, inventing "recipes" for integrating our smart machines into the larger demands of politics, war, finance, commerce, and everyday life? What role in particular might the arts play in this expanded history? What would it mean to see the "digitalization" of artistic media as part of it? Could we adapt what Bruno Latour calls "translation" for these purposes, challenging the idea, already found with "the computer," that digital intelligence arises by simply creating artificial forms for human activities? How then does such transformative translation work in the case of software? In what ways do the translations that software helps bring about introduce something new, for which no human model previously existed, challenging our very ideas of natural or nonartificial human activity? What role, in short, has software played in the ways we talk about and see things—and therefore act?

In fact, even to talk about the translations effectuated by computer programming, we often lack a preexisting vocabulary. The very words we are accustomed to using—"computer," "program," even "algorithm"—of necessity draw on predigital languages and practices, in genealogies we can now retrospectively examine. We see this, for example, in the case of "media" and "media studies" of the sort exemplified by Lev Manovich, part of the larger disciplinary framework through which we talk about and see digitalization. The terms "media" and "medium" were in fact drawn from the arts and journalism, then television, areas that themselves are being transformed by the rise of smart machines. In the arts today, for example, we see a movement away from the very ideas of media and medium, still important for a group like Radical Software, in favor of something more like "ecology of images." How might we start to chart these developments, studying software as a complex, evolving mode of thinking, within divisions of knowledge and artistic practices? Raising all these questions at once, this patient study, six years in the making, becomes a search for method and a plea for new ways of thinking, and the role that software practice might yet play in them.

What, then, is software as a mode of thinking? *The Software Arts* exposes an apparent paradox. While the grand logicist dream that led to the invention of the computer has long lost its philosophical hold on us, in many ways its ghost lives on in digital culture, in the very idea of smartness it helped introduce, in ways Warren Sack starts to trace: Turing machines, artificial intelligence, "cognitivism." He finds one turning point in Noam Chomsky, whose search for innate syntactic structures in language would lead to attempts to relocate such logic in the brain, or universal "neural cognition." To see software instead as a mode of thinking is to reverse the question—not whether our

brains work like computers but how the ways our computers are in fact programmed end up programming our brains to fit their operations. If, then, more generally the aura of mathematical logic still hangs over our very idea of digital smartness, it is precisely there that it matters to question it. The dream of the "laws of thought" today is to be found not so much in the realm of language, where it was once hotly debated, as in the digital world in which we now live and think, and the problem it poses is not about language alone but also about artifice and nature, fact and fiction, cognition and labor. To see software as a complex mode of thinking, to exorcise the ghost of logicism in our picture of it, in short, is to insert its operations into a larger history of such questions.

We see this in the sorts of historical approaches to which Warren Sack turns in his search for methods. In looking back to grammar or rhetoric as precursors of "software thinking," he draws more or less explicitly on Foucault's attempt to analyze the great philosophical preoccupation of his day with language as a complex "discursive event," which helps change our very image and manner of thought. But in adapting this approach to software studies, he shifts from a focus on linguistic form toward practical questions of intervening in environments at once artificial and natural, or where the two are interrelated in new ways. He thus encounters a later phase in Foucault's own work—his genealogy of bio-power, administration, and the role of numbers in it. How, then, has software or software thinking figured in that? In his discussions of big data and the very idea of an algorithm, Sack turns to these matters.

Ian Hacking went on to develop Foucault's study of bio-power in a striking way, pointedly asking how to do the "history of statistics." Warren Sack's question—how to do the history of software—might be seen as one continuation, which in effect asks how and where the two histories of statistical administration and software as a mode of thinking came together in the big data number crunching of today. Hacking distinguished between "styles of reasoning" and "methods of inference." More than a simple logic of inference, statistics is a materially rooted style of reasoning, with many unforeseen consequences, in particular for the very ideas of law and chance, and the new roles they would start to have in arts and letters. Numbers and numbering would play a key role in this new practice. Hacking talks of an "avalanche of numbers," which would pose new questions about who "counts" in society and who does the counting. The role of numbers in this manner of reasoning is of course quite different from their role in the methods of logic, famously posed by Frege and Russell, or even later by Wittgenstein. Hacking thinks it marks a shift from a mechanical universe (a clock wound up by God) to a "stochastic" universe, where probability replaces certainty and the "game of thought" starts to be played in the new ways, which came to be associated in France with Nietzsche and Mallarmé's search for an "untamed" chance that no roll of the dice can ever abolish.

How, then, did computers or computing enter into this administrative territory, introducing a new "translation" in it? What role does it play, for example, in Cambridge Analytica and Facebook's "commercial surveillance"? When we look back, for example, at the opening of the Stasi files in what was thought to be postcommunist Germany, we see a predigital phase of administration—bureaus, paper files, archives—which now seems quite quaint. Today's "bureaus" are all increasingly on our screens. But does this shift—this software translation—also introduce something new, a sort of shift in administrative reason? The East German case, of course, was part of the sort of Soviet "brainwashing" Orwell feared might be coming to Britain. But in our current "post–Cold War" situation, are we confronted with something new that neither Orwell nor even Foucault lived to see? What role does software play in it? What makes data big is not sheer quantity but rather the invention of software enabling inventories at great speed that extract certain kinds of information (or facts) about us—our "friends" and "likes," our buying and voting habits—as well as the rise and fall of stocks. Perhaps, to talk like Foucault, we could look at how such software thinking helps "constitute" us as buyers, voters, or stockholders, in the process turning electoral politics into a new domain of manipulation and interference. Using a term from Peirce, Sack proposes to see in big data a sort of algorithmic "abduction." But, in some ways, Peirce still belongs to the earlier nineteenth-century universe of chance, as Ian Hacking argues in his book on the topic. Perhaps our problem is no longer Hacking's shift from a mechanical to a stochastic universe, to which artists and thinkers responded at the end of the nineteenth century; perhaps now, in an age of big data, we need new analyses and new forms of artistic intervention.

What, then, is software as a mode of thinking? How should we do its history, and how might this history contribute to the larger questions of digital intelligence today? We come to the last aspect of this enterprise. To further such study, to pose such questions, Sack argues, we need to overcome entrenched divisions in knowledge itself, dividing "humanistic" from "technical" or "scientific" culture, finding new ways of drawing from each and initiating new kinds of collaboration. His own itinerary offers one example. After studying with Donna Haraway at Santa Cruz, he would pursue his studies at the Media Lab at MIT, where he started work as well with Visual Arts and the History of Technology. For his thesis, he would write code to visualize large-scale Internet "conversations," arguing that they worked in ways that go beyond anything found in linguistic theories of speakers or speaking—a project that already showed an ambition to go beyond the more lucrative military or commercial uses of software usually undertaken at the Media Lab, instead using software to participate in a larger debate, artistic and philosophical, crossing academic boundaries.

This new space of exchange and analysis across the arts and sciences is one that must itself be invented. That is the suggestion that emerges from this study. It is not simply a matter of getting the two "communities," intellectual-artistic and military-commercial, to talk more with each other, but rather to encourage the creation of a new space of discussion, for the problem of digital intelligence today is not simply one of the two cultures, scientific and humanistic. (It is not clear that the "intelligence" of software nerds is very mathematical at all; it is in fact drawn from many other sources.) The question it poses is not in the first place a matter of machines and us, artifice and nature, regarding who is in control. (The history of software is a history of modes of thinking that are at once artificial and natural, scientific and artistic.) The problem of digital intelligence is rather how to invent new ways of working together, outside the confines within which our thinking is now kept, and for that no simple return to a bookish "humanities" will suffice. The workings of our smart machines need to be analyzed at once in the arts and the sciences in ways and through methods and means that help create new relations between them. Only then will the reigning idea of smartness be replaced by something more like a kind of collective intelligence, working and thinking together across many domains and disciplines. The force and originality of this study is to show how software—software as a mode of thinking—has a key role to play in this process.

1 Introduction

Computers are language machines.
—Paul N. Edwards

The computer revolution can be envisioned as a rewriting of the world. This book is an examination of computerization as a work of rewriting or, more specifically, as translation. Increasingly, in academia, industry, and government, ideas are exchanged as software rather than as printed prose documents. Software now constitutes a new form of logic, rhetoric, and grammar, a new means of thinking, arguing, and interpreting.

This book argues that computing grew out of the arts. This argument will be a provocation for some, especially for those who see a bright line dividing the "two cultures"[1] of the arts and the sciences. For others, the argument will not seem provocative at all. Important computer scientists have argued that computing is not a science, software is a literature, and computer programming is a kind of essay writing. For those who see no clear distinction between the arts and the sciences, this book will be an old saw with some new teeth.

Prior to the "scientific revolution"[2] of the seventeenth century, there were no scientists in Europe. There were no professional organizations of science. Most studies that we would now identify as scientific were conducted under the name of "natural philosophy." Natural philosophers most frequently published their works in Latin, in which the word "scientia" meant something broader than the modern cognate of "science," something more like the general term "knowledge." The professionalization of engineering occurred even later. At that time and before, education and inquiry were carried out in the mechanical arts and in the liberal arts. This book argues that the software arts—like science and engineering—are the fruit of a coupling of the liberal and the mechanical arts. To demonstrate this argument, the approach taken is partly historical. By tracing the genealogy of computing back to events before or during the

initial professionalization of science and engineering, it becomes clear that computing grew out of the arts.

The Software Arts is also a reading of the texts of computing—code, algorithms, and technical papers—that emphasizes continuities between prose and programs.[3] Historically, it is possible to say that this position was first sketched out in the seventeenth century in proposals to develop artificial, philosophical languages that were used to knit together the liberal arts (e.g., logic, grammar, and rhetoric, the liberal arts of language) and the mechanical arts (e.g., those practiced by artisans in workshops producing pins, stockings, locks, guns, and jewelry).[4] In brief, these artificial languages became what we know today as computer programming languages. The claim is that contemporary, artificial languages have shaped and been shaped by the arts and have rearticulated the relationship between the liberal arts and the mechanical arts—an assembly we currently call art, design, the humanities, and technology.

Programming languages are the offspring of an effort to describe the mechanical arts in the languages of the liberal arts. Writing software is a practice of writing akin to the activity of novelists, playwrights, screenwriters, speechwriters, essayists, and academics in the arts and the humanities. Consequently, contemporary education, research, industry, and technology development all need to change to better recognize how the arts sit at the center of computing.

Apple's Artists

In 1995, Apple cofounder Steve Jobs said, "Part of what made the Macintosh great was that the people working on it were musicians, poets and artists and zoologists and historians who also happened to be the best computer scientists in the world.... And they brought with them, we all brought to this effort, a very liberal arts attitude."[5] Long after the introduction of the original Macintosh computer, Jobs was still describing the liberal arts as an Apple competitive advantage. At the launch of a new model of the iPad tablet computer, Jobs said, "It is in Apple's DNA that technology alone is not enough—it's technology married with liberal arts, married with the humanities, that yields us the results that make our heart sing."[6] In this book, I will argue that Jobs was right: the arts and the humanities are at the heart of computing.[7]

In the United States, Jobs's comments are remarkable today, when the arts and humanities are under siege with demands that students receive preprofessional training instead of a fine arts or liberal arts education.[8] The increasing disregard for a liberal arts education is misguided.[9] If Jobs was right, education needs to change. Computing

education needs to be redesigned to recognize its rightful place in the liberal arts, and the humanities disciplines of the contemporary liberal arts need to be extended to acknowledge their position at the heart of the computer revolution.[10]

If Jobs was right, it also becomes possible to imagine how computing research and development can be pursued as forms of arts research and humanities scholarship. With this insight, the path to the next "insanely great"[11] computer technology widens to become a great expressway accommodating a much larger and more diverse group of fellow travelers.

To emphasize the centrality of the arts would almost certainly help the computer industry with its long-standing diversity problems. At least that is the thinking that drove the summer 2014 diversity campaign in which Apple described itself this way: "From the very beginning, we have been a collective of individuals. Different kinds of people from different kinds of places. Artists, designers, engineers and scientists, thinkers and dreamers. An intersection of technology and the liberal arts. Diverse backgrounds, all working together."[12]

Computing and the Arts

Beyond Steve Jobs and Apple are a number of important computer scientists who have also put the arts at the center of computing. For example, Harold Abelson, Gerald Sussman, and Julie Sussman wrote a programming textbook for their undergraduate students at the Massachusetts Institute of Technology. Their textbook, *The Structure and Interpretation of Computer Programs*, embodies this alternative vision of computing. The authors state in their introduction:

> Underlying our approach to this subject is our conviction that "computer science" is not a science and that its significance has little to do with computers. The computer revolution is a revolution in the way we think and in the way we express what we think. The essence of this change is the emergence of what might best be called procedural epistemology—the study of the structure of knowledge from an imperative point of view, as opposed to the more declarative point of view taken by classical mathematical subjects. Mathematics provides a framework for dealing precisely with notions of "what is." Computation provides a framework for dealing precisely with notions of "how to."[13]

By distinguishing classical mathematics from computation—"what is" as distinguished from "how to"—the authors articulate a position of "procedural epistemology," but rather than coining the new phrase "procedural epistemology," they could simply have said that computing is an art. "Art" in its original sense means how-to knowledge—as used in phrases such as "martial arts" and "arts and crafts."

In their book, Abelson, Sussman, and Sussman emphasize one aspect of epistemology: that computing constitutes a new way of thinking. Computer scientist Edsger Dijkstra stated the case like this: "[Computers] have had a great impact on our society in their capacity of tools, but in that capacity their influence will be but a ripple on the surface of our culture, compared with the much more profound influence they will have in their capacity of intellectual challenge without precedent in the cultural history of mankind."[14]

This form of research and education, with a focus on the implications for cultural history, has been pursued with a mixture of methods that weave together ideas from the arts, the humanities, and mathematics. Donald Knuth, Professor Emeritus of the Art of Computer Programming at Stanford University, advocates a method he calls "literate programming": "Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.... The practitioner of literate programming can be regarded as an essayist, whose main concern is with exposition and excellence of style."[15]

Knuth sees programming as an art and as literature. Practitioners of "literate programming" and "procedural epistemology" are essayists, writers, and expositors. For Steve Jobs, Donald Knuth, Edsger Dijkstra, Harold Abelson, Gerald Sussman, and many other important computer scientists in the world (e.g., many who have won the Turing Award, the analog of the Nobel Prize for computer science), computing is part and parcel with the liberal arts. By arguing that the arts are at the heart of computing, I am arguing neither a radical nor a marginal point.

Computing and Engineering

Unfortunately, even though this argument has been made repeatedly and with great authority, it remains institutionally marginalized and generally unpopular. Institutionally—in both education and industry—the winning arguments have positioned computing either within the sciences or as a form of engineering. As a result, in universities, most computing departments are positioned in schools of science or engineering and away from schools of the arts and humanities.

While these "winning" arguments have been reified in the shaping of institutions, if we look closely at the arguments as originally stated and as pursued to date, we see that they are based on undefined terms and nonobvious and unstable analogies between computing and the subjects and objects studied and produced by science and engineering disciplines. Casting computing in a new disciplinary mold is frequently,

at least initially, not commonsensical. For example, in a remark at the first Software Engineering Conference, convened in 1968 by the North Atlantic Treaty Organization (NATO) Science Committee, an analogy was drawn between software production and engineering: "The phrase 'software engineering' was deliberately chosen as being provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines that are traditional in the established branches of engineering."[16]

In an article on the history of software engineering, Michael Mahoney writes the following about this opening gambit of the conference:

> The phrase was indeed provocative, if only because it left all the crucial terms undefined. What does it mean to "manufacture" software? Is that a goal or current practice? What, precisely, are the "theoretical foundations and practical disciplines" that underpin the "established branches of engineering"? What roles did they play in the formation of the engineering disciplines? Is the story the same in each case? The reference to "traditional" makes the answer to that question a matter of history—analyzing how the fields of engineering took their present form and searching for historical precedents, or what we have come to refer to as "roots."[17]

As Mahoney goes on to show in this article and subsequent scholarship, these terms remain undefined, decades after this first conference.[18]

One of the participants at the first Software Engineering Conference of 1968, Alexander D'Agapeyeff, had the following complaint: "Programming is still too much of an artistic endeavor."[19] The presupposition inherent in this complaint is, of course, that programming is already an art but that the aspiring software engineers would like it to be otherwise.

Indeed, "software engineering" remains an unrealized goal despite its current institutional success. Software engineers would like their discipline to be accepted as a form of engineering, but they are repeatedly unsure that it is. As Mahoney wrote in the introduction to his article, "It is … not hard to find doubts about whether its current practice meets those criteria and, indeed, whether it is an engineering discipline at all.… [It has been declared that] 'Software engineering is not yet a true engineering discipline, but it has the potential to become one.' From the outset, software engineering conferences have routinely begun with a keynote address that asks, 'Are we there yet?'"[20]

Is Computing a Science?

In 1967, a year before the first Software Engineering Conference, Allen Newell, Alan Perlis, and Herbert Simon published a letter in the journal *Science* arguing that computing is not (just) engineering but is also a science: "Professors of computer science are

often asked: 'Is there such a thing as computer science, and if there is, what is it?' The questions have a simple answer: Wherever there are phenomena, there can be a science to describe and explain those phenomena. Thus, the simplest (and correct) answer to 'What is botany?' is, 'Botany is the study of plants.' And zoology is the study of animals, astronomy the study of stars, and so on. Phenomena breed sciences. There are computers. Ergo, computer science is the study of computers."[21]

This letter had tremendous persuasive power and, arguably, launched the founding of many university computer science departments. It incorporates a number of tropes—rhetorical techniques—to press the case. Consider, for instance, the opening statement, "Professors of computer science are often asked…." In 1967, there were very few professors of computer science, because the first university computer science department had been founded only five years earlier, in 1962, at Purdue University.[22] All three of the letter writers were affiliated with Carnegie Mellon University (then called Carnegie Institute of Technology), where one of the first computer science departments in the country, after Purdue's, was founded in 1965 by the letter writers. Indeed, Alan Perlis had moved to Carnegie Tech from Purdue and was the first head of its Computer Science Department.[23] So, at the time, their opening line would have been akin to three of the first astronauts writing "astronauts are often asked…."

While this letter may have been the first word on the topic, it was hardly the last. In their 1976 Turing Award lecture, "Computer Science as Empirical Inquiry," Allen Newell and Herbert Simon group together geology, astronomy, and economics as "empirical disciplines" and then compare them as a group to computer science.[24] But geology, astronomy, and economics are not similar in any obvious way. Moreover, their intersection certainly does not provide a clear set comparable to an emerging fourth discipline, like computer science.

Newell and Simon elaborate on these unlikely comparisons with further far-fetched analogies: "Each program that is built is an experiment. It poses a question to nature, and its behavior offers clues to an answer."[25] Unaddressed by Newell and Simon is the poetic license they employ to anthropomorphize "nature" (a term they capitalize later in the article) into a being that can answer questions; the metaphorical notion that textual productions, like programs, are "built" rather than written; and the copular statement that makes building into a form of experimenting. That these kinds of loose and fanciful analogies carried the day and convinced many that the study of computing is a science may be puzzling but was—and, undeniably, still is—a winning rhetoric: "computer science" is more than analogical apposition; it is a large, growing, and well-funded discipline.

Simon tried to write the definitive word on this by publishing a book on the topic, *Sciences of the Artificial* (with three editions, in 1969, 1981, and 1996), but he never

definitively settles the question, Can a science focus on phenomena and objects that are not naturally occurring?

Computational Thinking as the Science of All Sciences

Some have even argued that computing is not just *a* science but *the* science of all sciences. This cadre of computer scientists has been influential in promoting the belief that computer science—or what they call "computational thinking"—is the queen of the sciences, as philosophy and mathematics once were.[26]

"Computational thinking," promoted by the US National Science Foundation;[27] the website code.org, supported by a coalition of largely corporate concerns;[28] and other educational initiatives deploy a discourse of computation-as-thinking so abstract that it seems to apply to everything but refers to nothing in particular.

Nevertheless, thinking is always thinking about something, and it is always thinking with other people.[29] One might even do well to question the very notion of thinking in general. So, we need to ask, with whom is "computational thinking" done and about what? Science studies scholar Bruno Latour exhorts us to rethink thinking as work, co-work. "We neither think nor reason," Latour writes, "Rather, we work on fragile materials—texts, inscriptions, traces, or paints—with other people."[30] So, what is the work of "computational thinking"? Who does it? And for whose benefit?

Gender Diversity

Even if we could resolve these questions about "computational thinking," it is clear that it should not be taught as computer science is taught today. Diversity is one of the most important motivations for a new approach. Unfortunately, computer science—as a field—has largely failed in its efforts at inclusion and diversity. According to one of the leading professional organizations for computer science, the Computing Research Association, women constituted only 14 percent of graduates receiving bachelor's degrees from American computer science departments in 2013.[31] These figures are especially worrisome because they appear to be headed in the wrong direction. For example, according to the US Department of Education, 37 percent of the computer science graduates in 1984 were women.[32] In other words, women used to make up over one-third of each year's computer science graduates, and thirty years later they represented only one-seventh.[33]

The problems of diversity in education can only be exacerbating the diversity problems of industry. Recent surveys of gender and ethnic diversity at the top computer

technology companies show that only 20–40 percent of the workforces of the most powerful computer technology companies are made up of women and that the vast majority of employees are white.[34] Disclosures by large technology companies concerning the diversity of their workforce are relatively new, but year-to-year data (e.g., from 2014 to 2015) show almost no progress.[35]

The fine arts, design, and the liberal arts—at least in the university—do not have the same diversity problems as science and engineering. I am motivated by the vision that computing can be taught and practiced as an art (or a set of arts). Pursuit of the software arts could reveal new horizons by opening the field to those in the arts and the humanities who now only think of themselves as users and not as makers of software.

Peter Denning, the former president of the main professional organization for computer science, the Association for Computing Machinery (ACM), wrote an article in 2013 that starts like this: "Computer science has for decades been ripped by an old saw: Any field that calls itself a science, cannot be science. The implied criticisms that we lack substance or hawk dubious results have been repeatedly refuted. And yet the criticism keeps coming up in contexts that matter to us."[36] What is appalling about Denning's opening line is not the idea that computing might not be a science but rather the presupposition that anything that is not a science must be a field that produces dubious results and lacks substance. Perhaps computing is not a science but nevertheless is a field of substance that produces rigorous, solid results!

Computing and Numbers

Common sense has it that—at their core—computers are made of numbers: ones and zeros. But consider an irrational number like the square root of two or, worse yet, a transcendental number like π. We can choose to represent π with a procedure that never terminates or as an infinite series of digits that begins 3.1415926….[37] For the computer, we are obliged to truncate this infinite series; we must decide how many bits of memory will be devoted to it.

If numbers were "native" to computers, we would not have to worry about how to approximate them. But, in actual practice, even if extremely precise calculations are needed, we tend to represent these infinite series with only 128 bits of computer memory.[38] Certainly, in principle, one could fill the entire disk and memory of a given computer with just the digits of one number, but even then we would be approximating an infinite series with a finite set of bits. Instead, an entire subdiscipline of computer science—numerical analysis—has been developed over the course of decades to address the problems of representing numbers and of calculating numbers with computers.[39]

If numbers were "native" to computers, no such computer science specialty would be necessary.[40]

My point is that although numbers and operations on numbers—such as arithmetic—can be approximated with a computer, computers are not numerical machines. They are language machines, and numbers are just a very common domain of application. Imagining computers only as powerful calculators confuses the machine itself with a single important application of computer technology.

Let me underline this pitfall with an absurd example. If I use the microwave oven in my kitchen mostly as a means to make popcorn, does that mean that the microwave is essentially a "popcorn machine"?

My point is controversial because it is, of course, a counternarrative to the most commonly told history, in which computers are figured as information technologies and are thus tied to information, quantification, and mathematics. In contrast, in the story I want to tell, computers are a coupling of the liberal arts and the mechanical arts—what today we would call the knowledge of artisans, artists, humanists, and designers. In the first story, computers are the materialization of mathematics and science. In the second story, computers are the manifestation of methods and theories from the arts and humanities.

Computing and the Liberal Arts

If computing can be conceived of as something other than just science or just engineering, what are the alternatives? Ironically, or perhaps characteristically, Alan Perlis—who, as mentioned earlier, argued in 1967 that computing *is* a science and, as a participant at the first Conference of Software Engineering in 1968, helped to articulate a vision of software production as engineering—described, in 1962, how computer programming should be integrated into a liberal arts education.[41] This third approach, computing as an intrinsic part of a liberal arts education, also advanced by Perlis, has more recently been revived by, for example, computational media theorists and practitioners Michael Mateas[42] and Ian Bogost in their respective advocacies for a "procedural literacy." Bogost specifically turns to an examination of the teaching of the language arts, the trivium, in his exploration of what a procedural literacy could mean for education and learning.[43]

Perlis's third path, the idea that computing research and education can be pursued within the liberal arts, is comfortable for computer scientists who are programming language designers, since they find it quite natural to imagine that computing is primarily about the creation and use of language. Alan Perlis was one of the designers of the ALGOL language.[44] Gerald Sussman, cited earlier, was co-designer of the Scheme

programming language.[45] Casey Reas and Ben Fry designed Processing, a programming language to help artists and designers learn how to program.[46] Those who have been especially articulate about computing-as-language-art have included the designers of programming languages produced to teach children and novices to program. Alan Kay co-invented—originally for children—the object-oriented programming language Smalltalk.[47] Mitchel Resnick's Scratch programming language has transformed computing education for young children.[48] Resnick's mentor, Seymour Papert, was the co-designer of a programming language for children called LOGO.[49] Papert was pivotal in tearing down the walls between the language arts, science, and mathematics. In his book *Mindstorms*, Papert put it like this: "Plato wrote over his door, 'Let only geometers enter.' Times have changed. Most who now seek to enter Plato's intellectual world neither know mathematics nor sense the least contradiction in their disregard for his injunction. Our culture's schizophrenic split between 'humanities' and 'science' supports their sense of security. Plato was a philosopher, and a philosopher belongs to the humanities as surely as mathematics belongs to the sciences."[50]

Papert's comments make us recall that the traditional liberal arts, as studied and practiced in the early modern era of Europe, included both the trivium (the arts of language) and the quadrivium (the arts of number). At that time, there was no strict boundary to be drawn between what today we call the arts and the humanities and the sciences.

A Short History of the Liberal Arts

What might a recasting of computing as part of the liberal arts look like? A short history of the liberal arts will show that they have been expanding and diversifying for centuries and that the design of programming languages, the languages of software, are the latest version of a very old dream of the liberal arts—to find just the right words, just the right language.

What are the liberal arts? The *Oxford English Dictionary* (*OED*) defines the liberal arts as:

> Originally: the seven subjects of the trivium (grammar, rhetoric, and logic) and quadrivium (arithmetic, geometry, music, and astronomy) considered collectively (now historical).

American Catholic nun Miriam Joseph wrote a widely read college textbook in which she defined the seven liberal arts as follows:

> The *trivium* includes those aspects of the liberal arts that pertain to mind, and the *quadrivium*, those aspects of the liberal arts that pertain to matter. Logic, grammar, and rhetoric constitute

> the *trivium*; and arithmetic, music, geometry, and astronomy constitute the *quadrivium*. Logic is the art of thinking; grammar, the art of inventing symbols and combining them to express thought; and rhetoric, the art of communicating thought from one mind to another, the adaption of language to circumstance. Arithmetic, the theory of number, and music, an application of the theory of number (the measurement of discrete quantities in motion), are the arts of discrete quantity or number. Geometry, the theory of space, and astronomy, an application of the theory of space, are the arts of continuous quantity or extension.... These arts of reading, writing, and reckoning have formed the traditional basis of liberal education, each constituting a field of knowledge and the technique to acquire that knowledge. The degree bachelor of arts is awarded to those who demonstrate the requisite proficiency in these arts, and the degree master of arts, to those who have demonstrated a greater proficiency.[51]

Many others summarize the trivium as the arts of language and the quadrivium as the arts of number.

Media scholar Marshall McLuhan wrote his dissertation on a history of three of the liberal arts, specifically the trivium.[52] He emphasized the historical centrality and continuity of the liberal arts, citing A. F. Leach: "It is hardly an exaggeration to say that the subjects and the methods of education remained the same from the days of Quintilian to the days of Arnold, from the first century to the mid-nineteenth century of the Christian era."[53] Arguably, in much of Europe, the liberal arts were at the heart of education for a millennium. Yet, as McLuhan's history makes clear, the static picture of the liberal arts projected by Sister Joseph was only a snapshot of a specific historical period. The definition of the arts and their relationships to each other changed dramatically from one era to the next, and thus "grammar," "logic," and "rhetoric" today are not necessarily the same as their historical precedents.

In the United States, many elite institutions are still called liberal arts colleges. Yet, what is called a "liberal arts education" today is no longer the Aristotelian endeavor outlined by Sister Joseph in 1948; nor is it exactly the silhouette A. F. Leach saw in 1911. In early modern Europe, the liberal arts were distinguished from mechanical or manual arts.[54] Today, in some countries, such as the United States, liberal arts colleges do include the fine arts. Elsewhere, however—for example, in France and Germany—art colleges and technical schools are separate from the university. But A. F. Leach's choice of the mid-nineteenth century as a critical moment for the transformation and expansion of the liberal arts throughout Europe and the United States is compelling, because it was then that both industrialization and the rise of the humanities changed the liberal arts by integrating them with the mechanical arts.

The contemporary definition of the liberal arts puts them in opposition to science and technology. I elided from the *OED* citation earlier this crucial sentence: "In later use more generally: arts subjects as opposed to science and technology (now chiefly

North American)." What happened in nineteenth-century American education that seems to have made technology and the liberal arts antonyms but at the same time paradoxically expanded a liberal arts education to include the mechanical arts, engineering, and technology?

In 1936, Henry Seidel Canby wrote a memoir set in the American college, specifically centered on his experience at Yale College. In *Alma Mater: The Gothic Age of the American College*,[55] Canby pointed out that the college had been radically transformed between 1870 and 1910. Industrialization, the rise of the corporation, and the invention of the American research university all played a part in changing college.

Before 1870, college was for an elite few. In the United States of 1870, there were about 50,000 undergraduates.[56] By 1920, there were an order of magnitude more undergraduates, over half a million. Before 1870, undergraduates enrolled in college were primarily pursuing a liberal arts education prior to entry into one of three specialties: divinity, law, or medicine. But during this period of industrialization, new, specialized forms of knowledge were developed to deal with the introduction of a myriad of emerging machines and new forms of production and distribution. Thus, for many, after 1870 a college education was no longer synonymous with a traditional liberal arts education.

The first federal aid for higher education in the United States was the 1862 Morrill Land Grant College Act: "An Act Donating public lands to the several States [and Territories] which may provide colleges for the benefit of agriculture and the Mechanic arts … in order to promote the liberal and practical education of the industrial classes in the several pursuits and professions in life."[57] The Morrill Act provided the founding financial support for many of the great public (and some private) universities of the United States, including the University of California, Berkeley; Purdue (later the site of the first computer science department); University of Wisconsin–Madison; University of Maryland, College Park; Massachusetts Institute of Technology; and Cornell. At the time, it was argued that the industrial era required a new form of college education incorporating "utilitarian" and "democratic" forms of knowledge.[58] Where previously a select group of gentlemen were instructed in the liberal arts before starting careers as clergymen, lawyers, or medical doctors, the Morrill Act signaled that a larger population needed to be educated in some hybrid of the mechanical and the liberal arts so that they might become accountants, engineers, and technical experts of the many, increasingly specialized, disciplines important for industrial capitalism. These new universities created by the Morrill Act were minted in a very different die than the older, originally ecclesiastical, colleges of early America, such as Harvard, Yale, and Princeton.

This thread of change—the expansion of college education to concern new forms of technical expertise[59]—was intertwined with another: the rise of the research university

in the nineteenth century.[60] Prior to the nineteenth century, universities were primarily teaching institutions where professors were paid to teach an established canon of knowledge, not to develop new forms of knowledge and technology. In the nineteenth century, the research university was invented in Germany, especially Prussia. This new model of the university was imported to the United States by then-new universities, such as Johns Hopkins and the University of Chicago, and, after that, was adapted by long-standing institutions, such as Yale. Central to this new model of the university—and a departure from the older models—was the emphasis on empirical scientific research; the change in the duties of professors to pursue research in addition to teaching; and the increasing investment in secular forms of knowledge.[61]

As McLuhan demonstrates in his history of the trivium—the three liberal arts devoted to language—a liberal arts education was, for centuries, tantamount to a Christian religious education. Nevertheless, since this form of education properly started in ancient Greece, even during the early modern period it combined ancient Greek philosophy (especially Aristotelian philosophy) with teachings of the Catholic Church (especially those of Thomas Aquinas). So, the liberal arts have always been interdisciplinary.

The emergence of the humanities from the liberal arts arose from a secularization of this body of knowledge. This third strand of change—secularization—was woven with the influence that the increasing ubiquity of technologies of industrialization and the rise of the research university had on education. Secularization displaced the sacred and, in its stead, centralized the study of the human.

In the introduction to their text *Digital_Humanities*, Anne Burdick, Johanna Drucker, Peter Lunenfeld, Todd Presner, and Jeffrey Schnapp encapsulate this history in one paragraph:

> While the foundations of humanistic inquiry and the liberal arts can be traced back in the West to the medieval *trivium* and *quadrivium*, the modern human sciences are rooted in the Renaissance shift from a medieval, church-dominated, theocratic worldview to a human-centered one.... The wellsprings of humanism were fed by many sources, but the meticulous (and, sometimes, not-so-meticulous) transcription, translation, editing, and annotation of texts were their legacy. The printing press enabled the standardization and dissemination of humanistic cultural corpora while promoting the further development and refinement of editorial techniques. Along with many other scholars, we suggest that the migration of cultural materials into digital media is a process analogous to the flowering of Renaissance and post-Renaissance print culture.[62]

What is left unstated in their paragraph is that the "post-Renaissance" lasted a long time. At Yale College, for instance, nontheological topics of study, especially science, did not become important until the nineteenth century, and even then they had to be carefully

negotiated with the clerical community in charge of the university.[63] Furthermore, the reigning "science" of the time was philology—the study of texts, especially ancient ones. Philology provided the model for newer sciences such as evolutionary biology.[64]

As sketched earlier—despite the assertion of the *Oxford English Dictionary* that the liberal arts are seen to be in opposition to science and technology—we find it difficult to narrate the emergence of contemporary disciplines and forms of knowledge without stumbling on the intertwining of the secular and the theological, the ancient and the modern, the coevolution of the humanities and the sciences, the liberal arts and the mechanical arts. Even a short historical review shows that the opposition between the liberal arts and science and technology is just a prejudice or a political strategy of marginalization to sideline the arts. The real story is instead a story of how cultural, economic, and technological changes arose already entangled with the arts.

Translation and the Liberal Arts

The liberal arts are not a static formation. They have always been constituted and reconfigured through the movement—the translation—of textual knowledge from one form into another. In Rome and then later throughout Europe, the development of the liberal arts was dependent on new practices of translation. Texts from Greek, Arabic, and Hebrew were translated into Latin starting, especially, with Jerome, who was born in 347 CE in contemporary Ljubljana, educated in Rome, and died in Bethlehem in 420 CE. As philosopher Barbara Cassin and the editors of the *Dictionary of Untranslatables* state in their entry "On Translation": "The notion of *translatio* is truly the confluence of the arts of language (grammar, logic, rhetoric) and of theology. In its widest acceptance of meaning, the term *translatio* designates a transfer of meaning, a displacement of signification, from a proper usage to an improper usage."[65]

As the confluence of the liberal arts of language (the trivium) and its primary practice (the translation and interpretation of primary texts in other languages), translation will be both the object of study and the method of analysis pursued in this book. We will examine contemporary forms of translation and how they have been employed to develop new metaphors, new equivalences, and even new identities. Contemporary forms of translation are tied up in various forms of computation, so we will refer to these new practices as the software arts. The software arts are a practice of computing that emerges from a coupling of the liberal arts and the mechanical arts. Just as it is possible to understand science and engineering as practices that emerged from artisans' workshops,[66] the software arts is an understanding of computing as a practice that emerged from the arts and the humanities.

Translation and Information Technologies

Previously, translation, as a practice of the liberal arts, meant a transformation that would, unavoidably, take away or add information to a text or message. In contrast, today's information and communication technologies are frequently developed with the ideal that messages can be moved or translated with no information lost or added. In their conception and design, information and communication technologies are meant to exactly reproduce that which was produced at the other end of the transmission line. In their foundational text *A Mathematical Theory of Communication*, Warren Weaver and Claude Shannon stated the issue like this: "The fundamental problem of communication is that of reproducing at one point either exactly or approximately a message selected at another point."[67] Elsewhere in the same book, Weaver and Shannon point out that, according to their theory, information is accorded no meaning.[68] In short, from an engineering perspective, what we are engaged in when we send email or use the web is a meaningless reproduction that pays no attention to its social and political environment. This meaningless mimicry is considered the fundamental problem of communication.

In engineering terms, information and communication technologies are evaluated according to their ability to push bits losslessly across a fixed-capacity channel, repeating what was recorded at one end of the "wire" exactly and without reference to environment on the other end of the "wire." In classical philosophy, there is a name for this kind of exact repetition. Repetition without knowledge of meaning, context, or association is called "imitation" or, more specifically, "mimicry," or "sophistry" in the writings of Plato. In his Socratic dialogue titled the *Sophist*, Plato states: "Some mimics know the thing they are impersonating; others do not.... [F]or the purposes of distinction let us call mimicry guided by opinion 'conceit mimicry,' and the sort guided by knowledge 'mimicry by acquaintance.'... The art of contradiction making, descended from an insincere kind of conceit mimicry, of the semblance-making breed, derived from image making, ...—such are the blood and lineage which can, with perfect truth, be assigned to the authentic Sophist."[69]

In other words, the criteria for success articulated in information and communication theory are exactly Plato's criteria for the worst kind of ethical and aesthetic failure. They are, according to Plato, the worst behavior of sophistry taken as a virtue, indeed taken to be the fundamental virtue of a working technology.[70] For these reasons, contemporary theories and technologies of information, communication, and computation can be called "sophisticated," a point I will elaborate on in my discussion of rhetoric in chapter 6.

The connection between communication theory and translation's radical transformation under the conditions of computation was already apparent in Warren Weaver's 1949 report titled "Translation."[71] Weaver's purpose was to explore the idea that one might design a computer program to translate texts from one language into another. Those familiar with Shannon and Weaver's text on the theory of communication will not find the following too surprising, but any bilingual person is likely to find Weaver's understanding of translation fantastical. Weaver wrote: "When I look at an article in Russian, I say, 'This is really written in English, but it has been coded in some strange symbols. I will now proceed to decode.'" Weaver wrote this shortly after World War II, when protocomputers were first applied—with great success—to the problem of breaking Germany's military communication codes. In short, for Weaver, it was clear that computers were good for the tasks of decryption, so if a problem could be reconceptualized to look like a decryption problem, then it was probably something a computer could do. Despite skepticism voiced by scientific luminaries of the day,[72] Weaver's "Translation" essay was enormously influential[73] and, arguably, still informs computer scientists' approaches to translation. For example, the statistical approach to decoding that Weaver outlined in his essay constitutes the core of popular work in contemporary machine translation.

Two points of Weaver's text from 1949 are notable. First is the previously unusual idea that machines—specifically computers—can be used to translate texts from one language into another. Second is the radical translation of the very word "translation" that his text performs: Weaver equates translation with the operations of (information lossless) encryption and decryption. These two points will serve as points of contention for our exploration of the software arts.

Any bilingual person with access to the web can go online and test the current viability of Weaver's assertion that translation is just a form of decryption. For instance, find Google's online translation service[74] and try translating a text from one language into another. Unless the text is incredibly simple, the automatic machine translation will be riddled with errors, even gross mistakes. Certainly machine translation has improved dramatically since its inception in 1949, but there is still a gap between machine translation and good translation. Furthermore, there is not just a gap but a chasm between machine translation and transformation of a text in one language into a text in another language without loss or gain of information (i.e., a perfect translation as conceptualized by Weaver and other information theorists).

One of the problematics central to the software arts is to grapple with how computers and networks have come to incorporate this untenable idea of lossless translation and to pursue this by exercising older understandings of translation as they have been developed by scholars of the liberal arts and the humanities.

Translation and Science and Technology Studies

Historian and philosopher of science Michel Serres, in his book *Hermès III: La Traduction*, developed an approach to studying science and technology by means of translation.[75] Serres's method was then extended under the rubric of a "sociology of translation," alternatively, and more commonly, called "actor-network theory" or just ANT.[76]

Philosopher and science studies scholar Bruno Latour regrets the more recent and popular name "actor-network theory":

> At the time [the late 1970s and early 1980s], the word network, like Deleuze's and Guattari's term rhizome, clearly meant a series of transformations—translations, transductions—which could not be captured by any of the traditional terms of social theory. With the new popularization of the word network, it now means transport without deformation, an instantaneous, unmediated access to every piece of information. That is exactly the opposite of what we meant. What I would like to call "double click information" has killed the last bit of the critical cutting edge of the notion of network. I don't think we should use it anymore at least not to mean the type of transformations and translations that we want now to explore.[77]

Readers of Latour's more recent work know that his previous mention of "double click information" was far from casual.[78] In a recent book, he identifies "double click" as one of the fifteen "modes of existence" examined and strongly distinguishes it from other modes of existence, including what he calls the "mode of networks."

A sociology of translation is appropriate to an analysis of software, since computer scientists describe much of their work as translation. Beyond the problematics of machine translation (from one natural language into another), many practical problems are called "translations" in software design and engineering, including those translations alternatively labeled, in computer science, "compilations" or "interpretations" (as in the translation from a high-level programming language into a lower-level language) and "implementations" (as in the translation of an abstract system specification into a piece of working software).

Translation—as we know it in the arts and humanities—is always a force of change. Each translation involves either the loss or the addition of information, or both. Consequently, the result of any translation is not the same as the text translated—it is different. But translation—as it is known in computer science and information theory—is ideally lossless. From this perspective, a perfect translation from the source text to the target is one in which information is neither lost nor gained. A humanities scholar is likely to find the computer science approach to translation naive or idealistic. The computer scientist, conversely, might view the humanities approach as self-defeating since, if a translation can only be a deformation, degradation, or elaboration of the source

text, then an accurate translation is always out of reach. But humanists do not believe translation is impossible, only that any given translation can always be improved. This book explores the clashes that emerge from these differing perspectives on translation.

The Software Arts and the Liberal Arts of Language

Historian of science and technology Paul Edwards, commenting on a famous essay about computer historiography by Michael Mahoney (cited earlier),[79] explained that computer histories generally fall into a very small set of story types. Edwards wrote, "The first genre is an intellectual history in which computers function primarily as the embodiment of ideas about information, symbols, and logic." He continues: "The standard lineage here runs from Plato's investigations of the foundations of knowledge and belief, through Leibniz's rationalism, to Lady Lovelace's notes on Charles Babbage's Analytical Engine and Boole's *Laws of Thought* in the nineteenth century."[80] The lineage runs through the twentieth century and includes Alan Turing's machines, Norbert Wiener's cybernetic theory, the McCulloch-Pitts theory of neurons, and John von Neumann's collaborative work on computers. It is a history I retell in this book, but I retell it here in tension with two other less popular narratives: computer-as-rhetoric and computer-as-grammar.

This book is structured around the trivium, the three language arts of logic, rhetoric, and grammar. As Edwards and Mahoney point out, many computer histories tell the computer-as-logic story. In contrast, the computer-as-rhetoric history is one that emphasizes the role of computer software as a means of persuasion. Computer graphics and computer simulations are two forms of software widely deployed as forms of rhetoric. In contrast with the computer-as-logic and computer-as-rhetoric histories is the narrative of computer-as-grammar. Grammar concerns the rules of language. Grammar rules have long been considered by some to be machines. The story of computer-as-grammar came into focus in the mid-twentieth century, when it was ventured that the rules of language are machines or devices of software. By spinning together these three separate histories of computation, a clear picture can be woven of computing as a coproduction of the liberal arts of language.

The computer-as-logic, computer-as-rhetoric, and computer-as-grammar stories are three different interpretations of computing. These stories intersect somewhat but are also quite different. Their differences are partly explicable by the historical differences between logic, rhetoric, and grammar as three separate fields of study.

Our contemporary usage of the terms "grammar," "logic," and "rhetoric" is symptomatic of an old rivalry between them. As McLuhan describes in his history of the

trivium from ancient Greece to about the time of William Shakespeare, each of these areas of knowledge has been in competition with the others for millennia. When we praise someone for their logic and deride another for their rhetoric, we are voicing the current state of this competition. Clearly, today, logicians are accorded more respect than rhetoricians, since phrases like "empty logic" and "sound rhetoric" seem almost oxymoronic; the adjectives "logical" and "rhetorical" are laudatory and derogatory, respectively. Grammarians are now the leaders in this three-way rivalry, since grammar is institutionalized to the extent that we think nothing of sending our children to grammar school or having them learn grammar from their earliest years in school. We would be inhabiting a very different world if children were sent to rhetoric or logic school—rather than to grammar school—at age seven. Since rhetoric, logic, and grammar have been in a rivalry for ascendancy for centuries, the triptych painted in this book—of computer-as-logic, computer-as-rhetoric, computer-as-grammar—will expose some raw edges between them.

Stakes and Claims

The stakes of this book are threefold: pedagogical, industrial, and epistemological. First, if software is an art, then education needs to change to integrate it into the liberal arts. What do we teach? What do we learn? These are old questions that need to be posed once again in a world where basic literacy is not just a matter of English, Latin, and Greek but also of software. Second, if software can be written in the manner of an artist/humanist, then new avenues of software production beyond engineering and mathematics may be possible—avenues that some, like Steve Jobs, have already traveled. Third, if software is the new lingua franca, then there are a series of ethical and moral questions that must be pursued in conjunction with this epistemological transformation. What counts as knowledge, for whom, and at what cost?[81]

The most general claim of this book is that the software arts is a new name for something that has been going on for centuries: the pursuit of methods to invent and interrogate statements of connection, equivalence, and identity. Today, writing software is essential to both science and engineering. Yes, writing programs is very different from writing prose. Yes, computer languages are distinctively different from natural languages. But, regardless of whether we call software "machines" or "instructions," "objects" or "rules," regardless of whether we call the production of software "building" or "writing," "construction" or "composition," our names for software and its production are metaphors, and once we are working with metaphors, we are working as artists. I argue, along with Steve Jobs, that the arts are at the center of software.

History, Philosophy, and the Software Arts

I have tried to substantiate the constitution of the software arts both historically and philosophically. In what follows, I trace a more detailed history that runs from Denis Diderot, to Adam Smith, to Gaspard de Prony, to Charles Babbage, to Ada Lovelace, to Alan Turing, to today. When Denis Diderot and his collaborators on the eighteenth-century *Encyclopédie* were faced with the task of describing, in some standard manner, all of the processes, operations, and gestures employed in the studios and workshops of diverse artists and artisans throughout France, they needed to forge a language acceptable to the liberal arts and adequately descriptive of the mechanical arts. The encyclopedists did this by first listening to the artists and artisans to learn the language they used to describe their work practices and the machines they employed in their productions. In chapter 3, I call language like this "work and machine language." Then, the encyclopedists had to translate these various work and machine languages into a uniform lexicon and syntax for the *Encyclopédie* entries and a uniform visual language for the images that illustrated the entries. The language of the *Encyclopédie* had to cover everything from the making of stockings to the manufacture of pins. I argue that the encyclopedists' translation between the mechanical arts and the liberal arts eventually constituted the basis for what we know today as programming languages.

Philosophically, the substantiation of the software arts is entwined with this history. Francis Bacon, Gottfried Wilhelm Leibniz, and others were, in the words of semiotician, novelist, and historian Umberto Eco, engaged in a "search for the perfect language": a philosophical language, a "universal characteristic" with which all types of knowledge could be articulated. But none of them attempted a full-scale encyclopedia of knowledge. However, Diderot et al. did attempt such a full-scale catalog and did so with an understanding of how their project responded to Bacon's and Leibniz's aspirations.

The software arts continue with the encyclopedists' efforts to translate the mechanical arts into a language of the liberal arts and vice versa. The lingua francas forged for this project are now programming languages, forms of inscription liminally positioned between prose and machines.

On the Limits of Translation

Programming languages are limited in comparison with natural languages because they, and the programs they comprise, are imperative, independent, impersonal, infinitesimal, inscrutable, and instantaneous in ways that no previous forms of language are or have been. For example, in a programming language, one can "conjugate" only in

the imperative ("fetch," "return," "assign," etc.) and the conditional ("if this then do that"). The subjunctive, even the past tense, is beyond expression in a programming language. Programming languages are "languages" and yet they are "machines."

Given this picture of programming languages, it does not seem too difficult to imagine that translating an analog process into a piece of software is frequently like forcing a large, round peg into a small, square hole. Like any translation, even one between spoken and written languages, a translation into a programming language is an exercise in loss, change, and addition. Stuff gets broken and has to be mended and amended in the process. In the words of the Italian proverb, "tradurre è tradire"; that is, "translation is betrayal."

As elaborated in chapter 2, this understanding of translation—that it is always "lossy," always an agent of change—is commonsensical to most polyglots and to those in the arts and humanities more generally. In a translation between a source language and a target language, there is always some "je ne sais quoi" in the source language text that is impossible to phrase in the target language, so there is always something that needs to be made up and inserted into the target-language text that has no counterpart in the source-language text.

Problems with Perfect Languages

The search for the perfect language, however, carries with it a fantasy of the "lossless" translation. This fantasy is the conviction that we can design a perfect language into which anything and everything can be translated without loss, addition, or change. This fantasy has persisted in mathematics and logic, and it now animates programming language design.

Here then is the spark for a repeated drama. Believers in a perfect language translate X, in a source language, into Y, a text in a "perfect" target language. The believers then declare that "X is Y." But to declare "X is Y," one must either overlook some "je ne sais quoi" that differentiates X from Y or one must deem this difference insignificant. This drama recurs in many of the other chapters.

Once it is declared that "X is Y," there will always be a nonbeliever, someone who does not believe in the perfect language and who will dispute the idea that the difference between X and Y can be overlooked or who will argue that the difference between X and Y is a significant difference. Today, the posited perfect language is frequently a computer programming language; Y is some piece of software written in that language; the believers are computer scientists or software engineers; and the nonbelievers are former stakeholders in X who have now lost control of X because Y has displaced it.

For example, let us imagine that X is the book publishing, distribution, and retailing business as it previously existed and Y is Amazon.com, especially the software that runs the website, but also the various internal software systems that predict readership demand, organize warehouses, route boxes of books between warehouses and from warehouses to customers' houses, and other operations. The nonbelievers in this scenario include the former local bookstore owner and her former regular customers who used to love her reading knowledge and her ability to always have a book they wanted, even before they knew of the book.

When X is a long-standing institution—like the book business—then Y is, in the vernacular of Silicon Valley, an attempt to "disrupt" that institution and replace it with a new institution configured around a piece of software. Amazon.com is an example of this; so is Uber's "disruption" of the taxi business. Bitcoin is an effort to disrupt banking and monetary currencies, Netflix has upended television, and there are many other examples. When many of our social, cultural, economic, and political institutions are configured around software, I say we live under "computational conditions," or, more simply, that our life is a "digital life."

When X is an idea, an area of research and teaching—such as biology, sociology, or physics—then Y is an attempt to replace some key intellectual construct; Y could be a model, a simulation, the "search" in "research," even a theory. The genetic code, for example, is often investigated as if it was a computer code.

Ideologies and Equalities

I consider a set of equivalences and inequalities as constituting a system of ideas or, more concisely, an "ideology" (with no pejorative sense of the term intended). When the set of equivalences and inequalities concerns software, or computers more generally, I call this a "digital ideology." When Y is a piece of software and can be considered a "theory," we have arrived at a strange new place where knowledge can be phrased as a computer program. I call this place a "computational episteme" and interrogate its implications in chapter 7.

Translation of a big idea or a big institution cannot be done with a little piece of software and a grand statement of equivalence, even though it has been tried. When computers were first introduced in the 1950s, journalists naively called them "electronic brains." This ideology of 1950s journalism and popular culture included the assertion "computers=brains." In contrast, in rigorous intellectual climates and in demanding industries, large equivalences need to be substantiated with a large collection of smaller equivalences. To constitute such a collection entails translating many smaller ideas or institutions into many smaller pieces of software.

So, before one asserts that the computer is a brain, one might want to investigate, as Warren McCulloch and Walter Pitts did in "A Logical Calculus of Ideas Immanent in Nervous Activity," published in 1943, whether neural activities can be modeled as a set of logic circuits. If one wants to call a small computer a "smartphone," there is a large set of smaller equivalences that need to be established, for example to render and operationalize "buttons" for dialing a number as a graphical object on a touch screen. Equivalences between two unlike entities are established by translating one into the other.

These translations are frequently predicated on an older set of equivalences established through earlier translation efforts. Thus, for instance, McCulloch and Pitts's logical calculus was based on earlier efforts to translate all of mathematics into logic (e.g., the work of Bertrand Russell and Alfred North Whitehead). These earlier efforts, in turn, developed partially from translations attempted in the converse direction, such as George Boole's nineteenth-century efforts to render Aristotelian logic as a form of algebra or arithmetic.

There are many techniques and methods, especially from mathematical logic and the theory of computation, to equate objects and processes with software and to "prove" or "disprove" an equivalence between one piece of software and another and/or between a piece of digital software and an analog entity. Yet, from the perspective of the arts and the humanities, "proofs" of equivalence are not irrefutable but rather are only arguments or demonstrations that a translation was done rigorously. In chapter 6, on rhetoric, various forms of demonstration with computation are examined.

If we think of these equivalences as "proved" and thus settled once and for all, we are unlikely to think of alternatives or innovate further. If instead we think of these equivalences as the result of a translation effort, we can always ask, "What got lost in translation?" For example, even though Claude Shannon "proved" in his master's thesis that Boolean logic and Boolean circuits are the same thing, if we persist in asking how they are different, we quickly unearth a set of problems circuits have that written logics do not; for example, while designing circuits, we have to worry about heat produced by the electrical current running through the circuit. While modern computers need fans to keep them cool, George Boole's nineteenth-century arithmetic of logic never needed a fan! Differences in materiality are significant differences. This example is discussed in detail in chapter 5, on logic.

Translation as Imperfect

To take this liberal arts attitude about the messiness of translation into the technical literature entails closely rereading pivotal papers of science, mathematics, and engineering to find and evaluate what got lost in translation. Given a proposed equivalence

X = Y, what link was established between X and Y; how was X translated into Y and Y into X? What had to be put to the side or ignored in order to establish the equivalence? Chapter 2, on translation, is essentially a chapter on methodology—an interpretation of actor-network theory—and its employment in following such chains of equivalence in the texts of software.

Bruno Latour wrote a small philosophical work, *Irreductions*, that starts with a declaration as far afield from a digital ideology as one can find. Latour asks, "What happens when nothing is reduced to anything else? … Nothing is, by itself, either reducible or irreducible to anything else. I will call this the 'principle of irreducibility,' but it is a prince that does not govern since that would be a self-contradiction."[82] While a digital ideology might include a myriad of reductions showing how life, the universe, and everything can be digitized or reduced to some form of computer program, Latour's declaration establishes an alternative and opposing manifesto: everything and everyone is singular, nothing is equivalent to anything else, everything is irreducible. However, even if that is the case, things can still be connected or linked together. With enough links between things, we have a network, thus "actor-network theory."

Under the theory of ideology exercised in this book, inspired by Algirdas Julien Greimas, an ideology can be analyzed as a set of equivalences between a number of ideas and/or identities. These equivalences and/or inequalities can be stated in a number of ways, including copular statements (X is Y), equations (X = Y), and—especially important for the digital—assignment statements (X:= Y; i.e., X is assigned the value of Y) and rewrite rules (X → Y; i.e., X is rewritten as Y). To do actor-network theory on an ideology is to question its equivalences: to ask whether they are really definitional or are provisional. In a sense, it is akin to what Socrates did in the agora of Athens by flipping the copular assertions of common sense and turning them into questions of definition: What is courage? What is truth?

Actor-Network Theory and Software Studies

Two independent inspirations for ANT were Greimasian semiotics[83] and sociologist Harold Garfinkel's ethnomethodology.[84] The work of ethnomethodologists has been dominated by ethnographic concerns and methods—observing and writing down the oral and physical interactions between people. In contrast, Greimasian semiotics was originally developed to study texts of literature.

ANT has been applied to the equations and equivalences of many areas of science and technology but only rarely to the texts of software. When they have been interested in software, the way that ANT practitioners, in particular, and STS researchers,

more generally, have pursued the study of computer science and software engineering has been heavy on ethnography and light on semiotics. Consequently, there are a number of wonderful ethnographic studies of various software development projects in which we learn a lot about the programmers and how they interact, but these studies tend to leave out of focus the texts of software: What exactly was the code being written by the programmers? What were the technical papers being read and written by the project group members, and how did the ideas proposed in the papers make their way—or not—into the goals and accomplishments of the programmers' productions, the code and its commentary?

This book illustrates how ANT, the sociology of translation, can be used as a method for looking at the texts of software (including code and technical papers). It is a necessary complement to the STS work that has been done to understand the people producing the software; that is, to, as one of Bruno Latour's books is subtitled, "follow scientists and engineers through society."[85] We need to study the texts of software as well as follow the people who produce it.

One would think that computer historians would have already written a lot about software, but much of computer history has focused on hardware, and only recently has software become an object of study. One might equally imagine that philosophers and historians of logic would have written a lot about software, but they rarely investigate contemporary software texts. Thus, in the lacunae of STS, computer history, and the history and philosophy of logic, there is an opening for a new kind of scholarship that focuses on the texts of software. This new kind of scholarship is the emerging field of software studies, and this book can be seen as a contribution to it.

Close Readings

Simply put, this book is a close reading of key texts of computer science and its history. For example, in chapter 4, when we read closely Donald Knuth's features of an algorithm—perhaps the key term for computer science—we find a circularity and several logical inconsistencies; for example, Knuth claims that algorithms can be defined independently of any programming language, yet all of his algorithms are defined in terms of a specific programming language of his own design. In chapter 2, reading Alan Turing's paper on the definition of Turing machines and papers by his students and colleagues popularizing the idea, we find that central to Turing's original paper is a negative result—that there are tasks that the computer cannot be programmed to perform. Yet, in its popular reception, we find an enthusiasm for the idea that a computer can be programmed to do anything and everything. In chapter 5, by closely reading texts

on logic and computation, we find—directly at the roots of software—a long-standing and strangely circular project to make logic into arithmetic and arithmetic into logic. This circular project has profoundly reshaped logic and has shaped computing since its inception. In chapter 6, on rhetoric, we follow the radical transformation of what it used to mean to demonstrate a point in a rigorous argument into the very different "demos" of today, the demos we see in Silicon Valley and in the arguments of "big data" practitioners. I call these demos "abductive demonstrations" and contrast them with the previously dominant forms of deductive and inductive demonstrations. In chapter 7, in an examination of the work of Noam Chomsky—as well as his teachers, colleagues, and students—we encounter the nascence of the notion that theories can be "devices," specifically devices rendered in software. When theories become devices, we do not ask *why* questions, we ask *how* questions: we ask whether the theories work. But we should also ask for whom they work and at what cost.

In sum, when we read the technical texts of software closely, we frequently find their popularizations are miles away from what the original texts actually say. Culturally, economically, politically, socially, and technically, this would not matter if software were a marginal concern left to a few specialists with no power and no influence, but today software looms large in research and teaching and in the everyday lives of people all over the world.

The Organization of the Book

The proposal of this book is that we read and write software as an extension of the liberal arts, specifically the trivium, the three language arts of the liberal arts. Logic, grammar, and rhetoric (in the guise of the "demo") are all instantly recognizable as intrinsic to software and central to computer science. The book is organized around these three liberal arts but with the first four chapters—this introductory chapter and then chapters on translation, language, and algorithms—devoted to introductory materials. In chapter 5, the first of the trivium, logic, is examined. Following that is a chapter on rhetoric, and then a chapter on grammar.

The composition of the book is self-similar, with the same kind of arguments made for the book as a whole and also at the scale of the chapter. I closely read a piece of computer science, looking for its instabilities and contradictions and seeking clues to its historical precedents. I then chase down those historical precedents to see what lost or neglected alternatives existed that could serve as improvements or correctives to the computer science of today. The instabilities and contradictions usually result from efforts to reduce everything to mathematics or logic or, more specifically, to digitize,

to turn everything into a form of arithmetic, to collapse it into a concern of calculation. In contrast, the historical alternatives I have found come from the arts and the humanities.

I have been writing this book with two readerships in mind. One is a set of cultural workers, artists, and scholars of culture interested in examining what software might have to offer in terms of theories, methods, or tools. The other is a group of computer scientists and software engineers whose work is bound up with cultural production—game design, social media, or streaming video. My message to all readers is that culture and computing are knotted together, and one way we can understand their entanglements is by closely reading the texts of software—code and technical books and papers. For each chapter of the book, I have a hope for the reader.

To begin, if this chapter, the introduction, has worked as I wished, the reader is willing to entertain the possibility that although computing can be seen as science (e.g., computer science) and as engineering (e.g., software engineering), it can also be seen as an art, or a collection of arts: the software arts.

Chapter 2, on translation, is offered as a "methods" chapter. Translation is known to the scholar and to the computer scientist, but each is familiar with a very different flavor of it. The main example discussed in the chapter is a set of texts from the beginnings of the theory of computation, concerning Alan Turing's machines, Alonzo Church's lambda-calculus, and popularizations of the Church-Turing thesis that claim that there are no limits to what a computer can do. This popularization is not true. Turing's original publication shows definitively that computers do have limits. By reading popularizations of the texts of software as a series of translations—from the most technical to the most popular—I show how the popular reception of a technical text can result in a fantasy that contradicts the findings of the original publication. My hope is that the reader will see how to reframe a popularization as a series of translations from the technical literature "out" into the wilds of popular culture and then back again—into the technical literature. The methodology presented is an amendment and an extension to actor-network theory, also well known in the field of science and technology studies as the sociology of translation. This contribution to actor-network theory (or ANT for short) is the main methodological contribution of the book.

In chapter 3, on language, I argue that computers are not information technologies and that the operations of computing are not the functions of mathematics. To expand on these assertions, I narrate a history of programming languages that starts in the artisans' workshops of eighteenth-century France. I trace a history of the division of labor as it was practiced in these studios and workshops; as it was transcribed by economist Adam Smith in the first chapter of his book *The Wealth of Nations*; as Gaspard Prony

revised Smith's ideas to organize large-scale calculations with human computers for postrevolutionary France; and as Charles Babbage devised a machine to embody Prony's methods in his Analytical Engine. Ada Lovelace, who in 1843 used the operations of the Analytical Engine to write the first computer program, called this a "science of operations" and contrasted it with mathematical logic. Lovelace wrote, "The science of operations…is a science of itself, and has its own abstract truth and value; just as logic has its own peculiar truth and value, independently of the subjects to which we may apply its reasonings and processes."[86] My hope is that through this history the reader will come to understand the huge gap that separates logic and mathematics from computation, and the affinities shared between computation and the kind of work that is accomplished in the arts.

In their original form, algorithms, the topic of chapter 4, were simply a new way to do arithmetic, which arrived in Europe when merchant capitalism was a dominant force. With the rise of industrial capitalism and in today's age of financial and linguistic capitalism, arithmetic has gone from being economically important to becoming the beating heart of the economy. Concomitant with this rise of arithmetic in commerce and industry was its transformation into a hegemonic form of knowledge from a circumscribed liberal art where it was one of the quadrivium, the arts of number, which also includes geometry, astronomy, and music. Originally spurred by the challenges of mathematics, over the course of the twentieth century, mathematicians, linguists, logicians, and economists reduced huge swaths of intellectual terrain to arithmetic, to calculation, in a move called "arithmetization." Arithmetization presaged what we now call "digitization" and "convergence." These moves to centralize calculation have been environmental disasters for many fields—as calamitous for thought as monocrop agriculture has been to the Earth's air, water, and soil. The hope for this chapter is that the reader, by following a history of the algorithm from early modern Venice to the computer algorithms of today, will be given the means to stop marveling at the power of algorithms and instead begin to think beyond the current ideological limits of algorithms. To do this, one needs to understand the industrial and intellectual forces that have been applied for well over a century to translate the language arts into the liberal art of arithmetic and, once understood, to reverse these translations.

The focus of chapter 5 is the liberal art of logic. The computer-as-logic story is often told to emphasize the many contributions of the language art of logic to the development of the computer. I, too, retell that story, but I include two details that are frequently occluded: materiality and history. It complicates the computer-as-logic story if one admits that computers of today—with power supplies, screens, and circuits—are materially very different from older works of logic printed on paper, so material details

are usually sidelined when telling the computer-as-logic story. Historical specificity is also usually marginalized in the common narrative because logic, as it is articulated today in technical venues, is not very old. First-order predicate logic is an invention of the twentieth century. Consequently, narrating how logic begat computers becomes too complicated if one acknowledges that the logic of yesterday was a completely different animal than the logics of today. By examining a history of logic, the material specificity of logic circuits, and some of the software design techniques of logic programming, my hope is that the reader will gain insights into how one can take a seemingly unitary, monolithic, technical topic—logic—and break it down into its many disparate parts. There is no such thing as Logic, with a capital "L," only a multitude of logics, all spelled with lowercase letters.

Chapter 6 is on the second art of the trivium: rhetoric. Aristotle tells us that the strongest rhetoric is closely tied to logical demonstration. This chapter traces a history of demonstration. The history of the "demo" starts in ancient Greece, when definitive demonstration was a matter of deduction as practiced in geometry. Euclid's demonstration or deductive demonstration is displaced by inductive demonstration in the seventeenth century, during the "scientific revolution." Inductive demonstration, which we can call Robert Boyle's demonstration, was made necessary when arguments began to be based on empirical data and not just derived from statements taken to be obviously true. Today, arguments are made on the basis of so much data—"big data"—that no one person could possibly read it all, much less observe its collection. This has necessitated the invention of yet another form of argumentation, which I term "abductive demonstration" or, alternatively, Solomonoff's and Kolmogorov's demonstration. The latest form of rhetorical demonstration is actually a kind of data compression, otherwise known as machine learning. My point is concordant with media theorist Jonathan Sterne's idea that we should now be concerned with compression rather than representation.[87] One could say that the founding document of contemporary machine learning was Ray Solomonoff's 1960 publication "A Preliminary Report on a General Theory of Inductive Inference."[88] Employing a theorem in Solomonoff's paper, Soviet mathematician Andrey Kolmogorov articulated a theory of complexity. The Kolmogorov measure of complexity of a collection of data is the length of the shortest computer program that can generate the data as output. Machine-learning algorithms are designed to accept a collection of data and then search the space of computer programs to find the shortest one that is applicable. We are led to believe that big data sets are aptly characterized by the outputs produced by machine-learning algorithms, but we have no way of checking the data, so I call this an abductive demonstration because philosopher Charles Peirce, who first articulated abduction in its contemporary sense, said that

abduction is a form of guessing. Alternatively, we might say that abduction is a form of interpretation, a practice well known to the arts and the humanities. The chapter proceeds from older means of making a point to the newest forms of persuasion. I hope to provide the reader with ways to both question and compose software-based arguments.

Chapter 7 is on the third of the three liberal arts of language: grammar. For a long time, grammar was a political project prosecuted as pedagogy in order to homogenize written and spoken language of empires. Later, it was deployed in an analogous manner to consolidate nation-states. Grammar was initially predominantly prescriptive. Then, in the late nineteenth and early twentieth centuries, grammar was reframed by linguists desiring to describe how language is actually used. With linguist and semiotician Ferdinand de Saussure, grammar became descriptive. When it did, its locus moved from textbooks into machines—both mechanical and imagined mechanisms of the brain. By the mid-twentieth century, linguistics had joined forces with the mathematical formalism championed by David Hilbert. This resulted in a transformation of linguistics to exclude meaning from its object of study. In the words of Noam Chomsky, "The study of meaning and reference and of the use of language should be excluded from the field of linguistics."[89] Instead, Chomsky and his followers pursue linguistics in the form of meaningless syntactic manipulations ultimately articulated as computer programs. After Chomsky, grammar machines became software, and claims were made that software could constitute a theory of language. This represented a huge shift in intellectual culture. When a computer program, a piece of software, can be a theory, we have entered what I will call the "computational episteme." In a computational episteme, software is taken for theoretical insight, and meaning is pushed to the margins. These conditions are strange and challenging. I hope that the reader will see that one way to make sense in a computational episteme—to revive meaning—is to act as an artist, to engage in the software arts.