CPU, Esq.

How Lawyers and Coders Do Things with Words

James Grimmelmann
Professor of Law, Cornell Tech and Cornell Law School

Brief Summary

Should law be more like software? Some scholars say yes, others say no. But what if the two are more alike than we realize?

Lawyers write statutes and contracts. Programmers write software. Both of them use words to do things in the world. The difference is that lawyers use natural language with all its nuances and ambiguities, while programmers use programming languages, which promise the rigor of mathematics. Could legal interpretation be more objective and precise if it were more like software interpretation, or would it give up something essential in the attempt?

CPU, Esq. explodes the idea that law can solve its problems by turning into software. It uses ideas from the philosophy of language to show that software and law are already more alike than they seem, because software also rests on social foundations. Behind the apparent exactitude of 1s and 0s, programmers and users are constantly debating the meanings of programs, just as lawyers and judges are constantly debating the meanings of legal texts. Law can learn from software, and software can learn from law. But law cannot become what it thinks software is -- because not even software actually works that way.

Summary

There is a close parallel between legal texts and computer programs. Both lawyers and programmers work with words that weave the world. But while the natural languages in which lawyers work are messy and ambiguous, the programming languages in which programmers work are precise and unambiguous. Since ambiguity, vagueness, and unconstrained discretion are the bugbears of legal interpretation, software offers an appealing alternative. Or so it seems.

Previous scholarship has explored the relationship between law and computer science by focusing on what software *does* and on how it *decides*. Scholars working in

the architecturalist tradition of Lawrence Lessig and Jonathan Zittrain have written about software as an alternative "modality of regulation" to law: it prevents unwanted behavior rather than punishing it after the fact. And more recently, a burgeoning scholarly community associated with the ACM FAT* (Fairness, Accountability, and Transparency) conference has delved deeply into the biases and pitfalls of algorithmic decision-making as compared with human decision-making on which legal systems have traditionally relied.

This book's focus will be different: it will ask what software *means*. Starting from the observation that adopting a legal text and running a computer program are both authoritative speech acts, *CPU, Esq.* will analyze how lawyers and programmers extract meaning from the texts they work with. It will use concepts from the philosophy of language to understand software, then turn the analogy around, using software to understand law.

It turns out that software provides an imperfect ideal for law, not because it is different and worse, but because it is much the same. Many of the same factors that make legal interpretation indeterminate and contentious are also at work with software. Programming languages are no less dependent on consensus and convention than natural languages; it is just that the social processes that establish shared meaning are a little subtler. While some of the approaches that technologists have devised are interesting and worth emulating, software cannot resolve the fundamental tensions inherent in legal language. Nothing can.

The book will make three principal contributions: to the philosophy of computer science, to legal doctrine, and to the philosophy of law.

Part I will advance the philosophy of computer science. A speech-act-based analysis borrowed from the philosophy of law provides a fresh perspective on the relationship between programmers, programs, and computers. The first four chapters will give a philosophical analysis of syntax and semantics in four types of language: the natural language that people use in everyday life (chapter 1), the legal language of statutes and other legal texts (chapter 2), the formal languages of mathematics and logic (chapter 3), and programming languages (chapter 4). Three further chapters will flesh out the analysis of programs by looking at their different interpreters: programmers (chapter 5), computers (chapter 6), and users (chapter 7).

Part II will clarify doctrinal debates in cases involving software. When programmers do things with software, the legal system needs to understand just what it is they are doing. Five chapters will consider how legal interpreters ought to approach software in settings where it has legal consequences. These questions arise in an interestingly diverse range of contexts, including the First Amendment

(chapter 8), copyright (chapter 9), patent (chapter 10), unauthorized access (chapter 11), and smart contracts (chapter 12).

Part III will turn to the big question for the philosophy of law: whether legal texts could and should be more like software. The first two chapters will deal with legal interpretation; whether computers could and should imitate human legal interpreters (chapter 13) and vice-versa (chapter 14). The second two chapters will deal with legal drafting, first substance — whether legal texts should be rewritten in formal languages (chapter 15) — and then process — whether legal drafting should be more like software development (chapter 16).

Detailed Outline

Part I: Foundations

Lawyers and programmers are modern wizards. A lawyer who puts the right words on paper under the right conditions summons a company into existence, or binds people to act in the future. A programmer who puts the right words into a computer under the right conditions builds a virtual castle, or makes a real airplane fly. These two professions weave the world with words. The question is *how*.

This part systematically compares and contrasts legal language and programming languages. To do so, it also considers two other types of language: natural language (of which legal language is an important subset) and formal logical languages (to which programming languages are closely related). The discussion of programming languages is the longest and most detailed, since it is the one least familiar to the legal-academic community.

Chapter 1: Natural Language

This chapter surveys standard concepts in the philosophy of language. It has two goals. The first is to bring out the basics of speech act theory, which explains how people "do things with words." The second is to survey the major theories of interpretation to identify different possible sources of meaning, including the speaker, the hearer, the relevant linguistic community, and the context of utterance. The distinction between *speaker meaning* (what a speaker intended to communicate) and *sentence meaning* (what an expression means standing on its own), provides a jumping-off point for later chapters.

Chapter 2: Legal Language

This chapter presents standard accounts of how legal language modifies the usual philosophical picture. The first major difference is that legal speech acts are conventional rather than purely communicative. They succeed only when they are made by speakers who possess the authority to make them and are made in the proper form. The second is that once they are successfully made, they have authoritative force: legal rights and obligations have an institutional existence in the world that goes beyond a hearer's recognition of the speaker's intent. These differences considerably complicate the nature of legal speech acts, so part of the chapter is devoted to unpacking the different kinds of things that different kinds of legal texts (statutes, contracts, wills, etc.) can do. It then considers some of the theoretical debates over the proper interpretive standards to use for legal texts. It pays particular attention to textualism — the view that interpretation should focus on understanding the sentence meaning of the text in question (rather than the intent of its author).

Chapter 3: Formal Languages

The development of a rigorous and mathematical concept of formal language was one of the major philosophical achievements of the century from 1850 to 1950, and it provided an essential conceptual foundation for the development of modern programming languages. In addition to presenting a standard modern account of the syntax and semantics of formal languages, the chapter briefly considers how some of these ideas have made their way back into the analysis of natural languages: natural-language processing systems rely on formal representations of syntax, and linguistics has borrowed fruitfully from logical semantics (albeit not with the same austere rigor). The chapter finishes by arguing that there that will always be a gap between the mathematical core of a formal system and the real-world uses that people can make of it. Similar gaps will appear in several ways in the discussion of programming languages.

Chapter 4: Programming Languages

This chapter is the first of four on software. It aims to give the reader an understanding of how programming languages work that is both intuitive and rigorous. It gives an extended example of a toy programming language and a few simple programs — first in a naive and intuitive way, and then again in a more precise way that builds on the formal approaches discussed in the previous chapter.

The first major point of the chapter is to bring out the idea that any program can be given a *naive program meaning* by running it and observing what the computer actually does. I call it "naive" because the concept is deficient in several ways — e.g., the computer might malfunction — and later chapters will provide the necessary corrections.

The second major point of the chapter is to characterize running a program as a speech act. The analysis here closely parallels the speech-act analysis in Chapter 2, but it also requires a digression to address the philosophical objection that a computer cannot "understand" software as speech. As soon as we understand a programmer as a speaker, the sentence meaning/speaker meaning distinction immediately shows that the programmer may intend something by the program other than what it actually does. This leads to the concept of *naive programmer meaning* — again, it is "naive" because it will require refinement.

An important running theme in this chapter and the next three is that technical systems are also social. A programming language, for example, has mathematically unambiguous semantics only to the extent that a community of programmers agree on the standard which defines those semantics.

Chapter 5: Programmers

Chapter 4 looked at what it means to run a program. Chapter 5 turns its attention to the programmers who write them. First, it sharpens up the concept of programmer meaning by looking closely at the debugging process. A better version of programmer meaning is w*hat a program would do if it were correctly specified and bug-free*. This concept depends on the working knowledge of a community of programmers, but it is also one that programmers intuitively apply on a daily basis. One programmer reading another's code can form a shared understanding of what the program should do well enough to debug it. This part of the chapter pays particular attention to formal verification —basically, the application of logical methods to try to bring program and intention into closer correspondence. As Chapter 3 foreshadowed, there is fundamental limit to program verification: a program can be proven correct according to a specification, but the specification need not match what the programer wants, or the real world.

The chapter then considers ways in which a program can carry *incidental meaning* to other programmers who read it rather than execute it. Some of this is derivative of program and programmer meaning (a reader may correctly understand what what the program will do or was intended to do), but some of it is independent of the program's function (source code comments can contain any text at all). The familiar distinction between source code and object code turns out to be

social rather than technical: source code is distinguished precisely by its ability to convey meaning to other programmers. This section also provides a tour of amusing code examples that show the wide variety of ways people can communicate with source code, including ASCII art and jokes in comments.

Chapter 6: Hardware

Chapter 5 went up from a program to the programmer. This chapter goes down from a program to the computer. It starts by working through the basic distinction between type and token: a given program can be encoded in many different possible physical instantiations.

The chapter works in detail through two issues, which closely parallel the discussion in chapter 5. First, just as the source code/object code distinction collapsed technically but was pragmatically useful, the hardware/software distinction is technically ill-founded but meaningfully captures a difference in how programmers work with computer systems. Hardware is what is fixed in a given context; software is what is mutable. Second, just as software can be formally verified, so can hardware. The reasons are a little different but the conclusion is the same: formal methods can greatly improve hardware quality, but perfection is impossible because abstract reasoning can never completely model the behavior of a physical system.

Then, just as Chapter 5 refined the idea of programmer meaning, this chapter refines the idea of program meaning. A more useful concept of program meaning takes as its model of computation not an actual physical piece of hardware, but an abstract model of the "hardware" on which a program runs. This model must be settled by social convention, but once it is, it provides a fixed context for interpreting programs. Thus, a better definition of functional meaning is as *what a program would do if executed on ideal hardware that never malfunctions*.

Chapter 7: Users

Programmers and computers do not exhaust the audiences for programs' meaning. Users also experience programs, and this chapter describes how.

Most importantly, when a program runs, part of its program meaning may be to cause a further communicative act to take place, one for which a user is the audience. This pattern is important enough that it requires a concept of *user meaning* to capture. Part of the chapter is a survey of some of the complex ways in which user meaning can be expressed. For one thing, the program/data distinction collapses: from a certain point of view, a .docx file is a program that can be run on the Microsoft Word virtual machine to produce user meaning through text. For

another, programs can be interactive, so an account of how they produce meaning needs to go beyond single utterances. This section draws on work in human-computer interaction (HCI) and game studies to illustrate some of the richness of software-mediated aesthetics.

The chapter also draws on HCI to make the point that users are typically not programmers and do not want to be. Modern computing interfaces walk a difficult tightrope between giving users clear and predictable controls, and doing what the users intuitively want. Both goals are important, but it can be hard to satisfy both at once. For example, the rise of verbal interfaces to software assistants like Siri and Alexa shows how one major task of modern interaction designers is to hide the immense underlying complexity of the systems they let users control: the creators of these interfaces have reasoned that it is better to be vague and ad hoc than to force users to express themselves precisely in a formal language.

Part II: Legal Doctrine

This part applies the concepts from Part I to concrete examples. Each chapter describes how a different body of substantive law asks legal interpreters to understand the legal effects of software. The running theme of this part is that the answer to "What does this program mean?" depends on why the question is being asked. Different legal contexts call for different interpretive approaches.

Chapter 8: Code as Speech

Is software protected by the First Amendment? The courts which have held that it is have accepted arguments grounded in incidental meaning — software is a medium for people to communicate ideas to other people. But one frequently sees stronger arguments for categorical protection which are grounded in program meaning, and which take the idea of software as "speech" to computers literally. Speech-act analysis, however, shows why the former argument is persuasive and the latter is not: program meaning is not the sort of communication covered by the First Amendment. Instead, software is and should be covered to the extent that it conveys user, programmer, or incidental meaning. These are different vectors, and disentangling them clarifies the different social roles played by software.

Chapter 9: Software Copyright

When is a program copyrightable? The canonical doctrinal answer is, "When it contains sufficient creative expression that does not merge with the idea of the software's function." On the one hand, this expression can come from incidental

meaning: source code comments, variable naming, and other decisions that are incidental to the code's program meaning. On the other, it can come from user meaning: an MP3 has the "function" of playing music, but that music may be copyrightable. Neither program nor programmer meaning, however, is a proper subject of copyright. (Note that this is a different answer than in the previous chapter: programmer meaning can support First Amendment protection but not copyright.)

Chapter 10: Software Patent

When is software patentable? Software patents are a difficult case because patent law rests on an invention-vs.-embodiment distinction that collapses when a description of the invention is also a functional artifact that can practice it. *Per se* arguments against software patents rest on treating the software/hardware division as a natural fact, which it is not. But the consequences of treating all software as patent-eligible may be bad enough that the line is pragmatically justified, even if it is not philosophically rigorous. To the extent that software is patentable, it is primarily program meaning that matters: doctrines such as utility and indefiniteness systematically exclude programmer and incidental meaning from protection. The chapter also includes a short discussion of design patent, which is focused on user meaning but also wrestles with the software/hardware division.

Chapter 11: Unauthorized Access

Computer-misuse laws, which prohibit "access" to a computer without "authorization." Frequently, whether use was authorized or unauthorized turns on the code itself: did it allow or disallow what the user did, or should the user have understood that what they were doing was allowed or disallowed in light of what the computer did? The problem can be understood as one of choosing between program, programmer, and user meaning and of specifying the context in which users are expected to ascertain that meaning. This may seem like a silly distinction: why would anything other than user meaning matter to users? But many misuse cases (and related cases such as ones involving gambling machines) involve software bugs, which allow the user to do something not intended by the computer's owner. To the extent that the user is expected to have understood that a particular use was forbidden *on the basis of interacting with the software*, unauthorized-access law effectively states that programmer meaning trumps program meaning.

Chapter 12: Smart Contracts

In a "smart contract" (which may or may not be a legal contract) the parties agree to have some of their obligations determined by the output of a program. The same general approaches as in the previous chapter are available, but the details are more complicated. First, whereas authorization to use a computer is unilateral, more parties are typically involved in setting up a smart contract, which makes programmer meaning more difficult to define. Second, the interface between legal contract and smart contract is richer: a legal contract can "wrap" a smart contract and specify interpretive rules, and a smart contract can (intentionally or inadvertently) trigger a legal contract as well. This means that the parties themselves may have attempted to specify the interpretive rules to be used, which raises difficult questions of the extent to which their requests should be respected.

Part III: Legal Theory

This part turns to the bigger questions: should law be more like software? In some places, the answer is "yes": programmers have a great deal of wisdom in preventing, detecting, and fixing bugs, from which lawyers could learn. In others, the answer is "no": the vision of software as objective, unambiguous, and apolitical rests on serious misconceptions of how software works in the real world. What looks like natural, mathematical perfection is actually the result of complex and contentious social processes — not unlike the rule of law.

Chapter 13: Artificial Intelligence

This chapter considers the stunning recent rise of big data, machine learning, and black-box artificial intelligence techniques. It argues that replacing human interpreters with computer programs is possible in theory but deeply inadvisable in the near term — and for the same reason. A close look at the arguments for and against using AIs to make legal decisions shows that the most persuasive such arguments turn not on the technical question of the formal correctness of the AI algorithms but on the social question of whether they can articulate their decisions in terms that are comprehensible to and acceptable to the relevant human community. Finally, the chapter includes discussion of corpus linguistics and other statistical and machine-learning techniques as applied to interpretation: they are obviously problematic, but in ways that any interpretive method is problematic.

Chapter 14: Legal Interpretation

The computer as interpreter provides a powerful foil for theories of legal interpretation. This chapter reconsiders the debates over textualism from Chapter 2. One way of thinking about the debates over textualism is as a choice of interpreters: are judges more like *computers* who are expected to carry out legislative directions predictably and precisely, or are they more like *other programmers* who are expected to find and fix bugs in legislative directions? Another related way of framing the question is as a choice between natural and formal language: to what extent are legal texts communicative, so that they take their meanings from linguistic usage, and to what extent are they conventional, so that they take their meanings from stipulated rules of usage?

A close look at program interpretation offers lessons on both sides. On the one hand, program meaning shows that nearly discretionless and nearly contextless interpretation is possible. Formal languages interpreted according to precisely specified semantics are in fact incredibly common in the world. On the other hand, programmer meaning shows that programmers frequently regard program meaning as the wrong answer. Sometimes a program is simply buggy. There are no definitive answers for law, because ultimately the choice of interpretive method is a question of positive law and the arguments for different methods are primarily normative. But a comparison with the world of programming clears away some of the underbrush of overbroad categorical arguments.

Chapter 15: Programming Law

Is the ideal law a computer program? The book's argument that a program's meaning is the result of conventions adopted by people puts this question in a new light. To commit legal decisions to software is not to remove them from the social and political realm of human decision-making. It is simply to mediate the social and political decisions in a different way.

More specifically, the chapter considers two overlapping ways that laws themselves — and not merely their interpretation — might be made more like software. The first is to rewrite statutes and regulations in a programming language, so that their meaning becomes unambiguous and their interpretation could be committed to computers. This idea closely resembles the way in which smart contracts commit interpretation and enforcement to computers, so the discussion here picks up where Chapter 12 left off. Many of the same problems immediately arise. Phrasing law's rules in a formal language of software and committing their interpretation to computers enables greater precision of

expression in some respects, but deeply impoverishes it in others. One crucial distinction is that statutes are unilateral and binding, which raises greater concern about fair notice — particularly when the statutes themselves contain the kinds of bugs to which software is prone.

Second, the chapter considers the possibility that — whether or not statutes themselves are expressed in natural or formal language — they might someday be drafted by AIs, perhaps even modifying parties' legal obligations in real time. Here, the analysis continues the discussion of AI interpreters in Chapter 13. Such systems ought to be judged by what they do and how they publicly justify it, rather than by how they work.

Chapter 16. Legal Drafting

This chapter compares and contrasts how lawyers draft legal documents and how programmers write software. It considers first the textual and structural aspects of drafting/coding: definitions, cross-references, modularity, etc. Then it turns to the processes programmers use, including agile methods, pair programming, and testing. It finishes the survey by looking at the toolchains lawyers and programmers use: editing software, version control, etc. It then bemoans the comparative lack of sophistication of almost everything in the legal drafting process: programmers have much better tools and use them far more effective. It discusses ways in which legal drafting is due for a revolution, with particular emphasis on what better testing methods could achieve.

Audience

This will be an academic book, but not exclusively a *legal* academic book. It will directly engage with the scholarly literature in technology law, philosophy of law, philosophy of language, and philosophy of computer science. I hope that it will be of interest to scholars working in law, computer science, philosophy, linguistics, STS, communications, and information science. I will assume no particular background in any of these fields, and will explain all necessary concepts from first principles. In particular, I will explain the relevant computer technology and the practices of programmers in careful detail. I would like for this book to be a canonical reference relied on by legal scholars when they give their own accounts of programming and software. The book should be appropriate for seminars in technology law, legal theory, and philosophy.

Manuscript

I expect the sixteen substantive chapters to average approximately 7,500 words. I anticipate about 5,000 words of introduction and conclusion, for a total of about 125,000 words. The manuscript will include endnotes formatted according to the authoritative legal-academic style guides, the Blue and Indigo Books.

Related Scholarship

There are no books, academic or popular, on the linguistic parallels between legal texts and software. There is, however, a substantial literature on each pair in this triplet: language and law, law and software, software and language. A major task of this book is to juxtapose these three lines of scholarship and bring out some of their related insights.

There are two bodies of scholarship on law and computer science that I see as models for *CPU, Esq.*. Neither of them deals with the linguistic connection that this book will explore, but both of them strike me as good examples of what legal theory can bring to this space. A third contains some interesting work but approaches the issues from such a different perspective that there is not much overlap.

One is the architecturalist tradition, which describes how technical architectures shape behavior and promote (or inhibit) innovation. Scholarship in this tradition focuses primarily on the *effects* of different architectures, and on how those architectures can be *regulated*. It has very little to say about legal interpretation or programs' meaning. *CPU, Esq.* will engage briefly with the architecturalist tradition in Part III, because some kinds of rules are more amenable to expression in software than others. The leading architecturalist monographs are Lawrence Lessig, *Code: And Other Laws of Cyberspace* (Basic Books 1999), Yochai Benkler, *The Wealth of Networks* (Yale 2006), Jonathan Zittrain, *The Future of the Internet — And How to Stop It* (Yale 2008), and Barbara van Schewick, *Internet Architecture and Innovation* (MIT 2010). Lessig and Zittrain's books are crossovers written with popular audiences in mind; Benkler's and van Schewick's are resolutely, austerely academic. *CPU, Esq.* will aim for a middle ground: my ideal is to combine the analytic precision of Benkler and van Schewick with the accessible prose of Lessig and Zittrain. Other books more loosely in this tradition include Primavera de Filippi and Aaron Wright, *Blockchain and the Law* (Harvard 2018) and Woodrow Hartzog, *Privacy's Blueprint* (Harvard 2018).

The other significant relevant body of books on law and computer science is more recent. It explores the ways in which algorithmic decision-making —

especially based on machine learning over large datasets — can differ from human decision-making and some of the unsettling legal issues that delegation to algorithms can raise. Notable books here include Frank Pasquale's *The Black Box Society* (Harvard 2015), Cathy O'Neil, *Weapons of Math Destruction* (Crown 2016), Virgina Eubanks, *Automating Inequality* (St. Martins 2018), and Safiya Umoja Noble, *Algorithms of Oppression* (NYU 2018). Again, *CPU, Esq.* will engage with this literature in Part III, but its concerns are different than mine. These books are about the institutions that deploy and rely on software; *CPU, Esq.* takes a much closer look at software itself.

A third body of law-and-computer-science scholarship is devoted to making law computable: building working computer systems that can assist with legal research or legal reasoning. The emphasis here is on the artificial intelligence, natural language processing, and data science needed to create reasonable software models of legal arguments or legal databases. Kevin D Ashley, *Artificial Intelligence and Legal Analytics* (Cambridge 2017) is a thorough survey of the field, and Michael A. Livermore & Daniel N. Rockmore, *Law as Data: Computation, Text, and the Future of Legal Analysis* (SFI Press 2019) is a recent collection of cutting-edge work. Part III will briefly engage with this line of work, but from a fairly abstracted point of view: what is it that these systems are actually doing?

There are some interesting analyses — diverse enough that I hesitate to call them a "line" or a "tradition" — of attempts to engage with the modeling problem: what is the nature of the relationship between a digital representation of a thing and the thing itself? The classic text here is Terry Winograd and Fernando Flores, *Understanding Computers and Cognition* (Addison-Wesley 1987), which argued strongly and persuasively for the existence of gaps between representation and reality of the sort that I discuss in chapters 3–6 and especially 7. (The book also has an overview of speech act theory, but applies it to user interactions rather than to programs themselves.) Some more recent works that hit on similar themes are Meredith Broussard, *Artificial Unintelligence* (MIT 2018), David Auerbach, *Bitwise* (Pantheon 2018), and David Golumbia, *The Cultural Logic of Computation* (Harvard 2009). I plan to put them in conversation with the scholarship on digital representations of primary legal materials, such as Peter M. Tiersma, *Parchment, Paper, Pixels* (Chicago 2010) and George S. Grossman, *Legal Research: Historical Foundations of the Electronic Age* (Oxford 1994).

The literature on the philosophy of language is of course immense, and so is the literature on legal interpretation. Interest in the intersection of the two is more recent. Books here include Brian G. Slocum, *Ordinary Meaning* (Chicago 2016), *Philosophical Foundations of Language in the Law* (Andrei Marmor and Scott Soames

eds.) (Oxford 2011), Andrei Marmor, *Interpretation and Legal Theory* (Hart 2005), Andrei Marmor, *Social Conventions: From Language to Law* (Princeton 2014), Timothy Endicott, *Vagueness in Law* (Oxford 2001), Lawrence Solan, *The Language of Judges* (Chicago 1993), Lawrence Solan, *The Language of Statutes* (Chicago 2010), and portions of *The Oxford Handbook of Language and Law* (Lawrence M. Solan and Peter M. Tiersma eds.) (Oxford 2012). Some of these shade over into linguistics and law; none of them deal with programs or computers as interpreters. My discussion of speech acts will draw on the classic texts of the field, including Kent Bach and Robert M. Harnish, *Linguistic Communication and Speech Acts* (MIT 1982), John R. Searle, *Speech Acts: An Essay in the Philosophy of Language* (Cambridge 1970), John R. Searle, *Expression and Meaning: Studies in the Theory of Speech Acts* (Cambridge 1985), J.L. Austin, *How to Do Things with Words* (Harvard, 2d ed. 1975), and Paul Grice, *Studies in the Way of Words* (Harvard 1989). Computers, I think it is fair to say, were not on their minds. Andrei Marmor, *The Language of Law* (Oxford 2014) is a recent entry in the literature on speech acts and law and will be a starting point for some of my analysis.

CPU, Esq. will cut through the philosophy of computer science at an unusual angle. There is no way to avoid touching on the classic Big Issues, such as whether computers can "think," but the questions I am interested in tend to be less explored. There are articles and book chapters on many of the questions I will analyze — such as the distinction between software and hardware, and the nature of specification — but few monographs deal with them in any detail. The closest would be Timothy Colburn, *Philosophy and Computer Science* (Routledge 1999), Brian Cantwell Smith, *On the Origin of Objects* (MIT 1996), Luciano Floridi, *Philosophy and Computing: An Introduction* (Routledge 1999), and *The Blackwell Guide to the Philosophy of Computing and Information* (Luciano Floridi ed.) (Blackwell 2004), but all of these cover so much ground that they can devote only a few shorter segments to the issues *CPU, Esq.* will cover. William Rapoport's ongoing draft textbook on the philosophy of computer science is an exception; its chapters on the nature of computer programs are closely on point and include a useful bibliography.

One important exception is a book from sociology, Donald MacKenzie's *Mechanizing Proof* (MIT 2001), which gives a detailed historical survey of formal methods and software verification. MacKenzie is exceptionally thorough and careful in presenting various arguments about whether and how proofs and programs correspond to the world, and in thinking through what a "proof" and a "program" actually are. MacKenzie's concerns are largely internal to computer science and software engineering; I think a legal audience would benefit from

seeing these ideas translated into their terms of art and applied to their own problems.