What Is Computer Science?

William J. Rapaport

Department of Computer Science and Engineering,
Department of Philosophy, Department of Linguistics,
and Center for Cognitive Science
University at Buffalo, The State University of New York
Buffalo, NY 14260-2500

rapaport@buffalo.edu
http://www.cse.buffalo.edu/~rapaport/

January 27, 2017

Abstract

A survey of various proposed definitions of 'computer science', arguing that it is a "portmanteau" scientific study of a family of topics surrounding both theoretical and practical computing. Its single most central question is: What can be computed (and *how*)? Four other questions follow logically from that central one: What can be computed *efficiently*, and how? What can be computed *practically*, and how? What can be computed *physically*, and how? What *should* be computed, and how?

The Holy Grail of computer science is to capture the messy complexity of the natural world and express it algorithmically.
— Teresa Marrin Nakra, quoted in [Davidson, 2006, p. 66].

1 Philosophy of Computer Science

In 2004, I created a course on the philosophy of computer science [Rapaport, 2005b];[1] a draft of a textbook based on the course is available online [Rapaport, 2017]. The book is intended for readers who might know some philosophy but no computer science, those who might know some computer science but no philosophy, and even those who know little or nothing about both. So, we begin by asking what *philosophy* is (primarily aimed at the computer-science audience), and, in particular: What is "the philosophy of *X*"? (where *X* = things like: science, psychology, history, and, of course, computer science).

I take the focal question of the philosophy of computer science to be: *What is computer science?* To answer this, we need to consider a series of questions, each of which leads to another: Is computer science a science, a branch of engineering, some combination of them, or something else altogether? To answer these, we need to ask what science is and what engineering is.

Whether science or engineering, computer science is surely *scientific*, so we next ask what it is a (scientific) study *of*. *Computers*? If so, then what is a computer? Or is computer science a study of *computation*? If so, then what is computation? What is an algorithm? ([Rapaport, 2017, Ch. 8] is a close, line-by-line reading of sections of [Turing, 1936].) Algorithms are said to be procedures, or recipes, so what is a procedure? What is a recipe? What is the Church-Turing Computability Thesis (that our intuitive notion of computation is completely captured by the formal notion of Turing-machine computation)?[2] What is "hypercomputation" (i.e., the claim that the intuitive notion of computation goes beyond Turing-machine computation)?

Computations are expressed in computer programs, which are executed by computers, so *what is a computer program*? Are computer programs "implementations" of algorithms? If so, then what is an implementation? What is the relation of programs and computation to the world?[3] Are programs (scientific) theories? What is the difference between software and hardware? Are programs copyrightable texts, or are they patentable machines? Ontologically, they seem to be both texts and machines, yet legally they cannot be both copyrightable and

[1] See syllabus and supporting documents at http://www.cse.buffalo.edu/~rapaport/510.html

[2] See [Soare, 2009, §12] on this name.

[3] As discussed in [Smith, 1985]; see also [Rapaport, 2015].

patentable [Newell, 1986]. Can computer programs be verified [Fetzer, 1988]?

We then turn to issues in the philosophy of AI, focusing on the Turing Test and the Chinese Room Argument [Turing, 1950], [Searle, 1980].

Finally, we consider two questions in computer ethics, which, when I created the course, were not much discussed, but are now at the forefront of computational ethical debates: (1) Should we trust decisions made by computers? [Moor, 1979]—a question made urgent by the advent of automated vehicles. And (2) should we build "intelligent" computers? Do we have moral obligations towards robots? Can or should they have moral obligations towards us?

And, along the way, we look at how philosophers reason and evaluate logical arguments.[4]

Although these questions arise naturally from our first question (What is computer science?), they do not exhaust the philosophy of computer science. Many topics are *not* covered: the nature of information, social and economic uses of computers, the Internet, etc. However, rather than aiming for universal coverage, I seek to provide a foundation for further discussion: Neither the course nor the book is designed to answer all (or even any) of the philosophical questions that can be raised about the nature of computer science, computers, and computation. Rather, they provide background knowledge to "bring students up to speed" on the conversations about these issues, so that they can read the literature for themselves and perhaps become part of the conversations by contributing their own views. The present paper is a synopsis of [Rapaport, 2017, Chapter 3], based on my Barwise Prize talk at the APA.[5]

[4] Computer Science Curricula 2013 covers precisely these sorts of argument-analysis techniques under the headings of Discrete Structures [DS]/Basic Logic, DS/Proof Techniques, Social Issues and Professional Practice [SP] (in general), and SP/Analytical Tools (in particular). Many other CS2013 topics also overlap those in the philosophy of computer science. See http://ai.stanford.edu/users/sahami/CS2013/

[5] I am grateful to Thomas M. Powers and Richard M. Rubin for comments and discussion at the 2017 APA Eastern Division session where I presented the material in the present essay. And I am especially grateful to the American Philosophical Association Committee on Philosophy and Computers for this distinct honor, which recognizes "contributions to areas relevant to philosophy and computing" (http://www.apaonline.org/?barwise). Because of my philosophy interests in philosophy of mind, I was inspired—by [Hofstadter, 1980]'s review of [Sloman, 1978], which quoted Sloman (p. 5) to the effect that a philosopher of mind who knew no AI was like a philosopher of physics who knew no quantum mechanics—to study AI at SUNY Buffalo with Stuart C. Shapiro. This eventually led to a faculty appointment in computer science at Buffalo. Along the way, my philosophy colleagues and I at SUNY Fredonia published one of the first introductory logic textbooks to use a computational approach [Schagrin et al., 1985]. At Buffalo, I was amazed to discover that my relatively arcane philosophy dissertation on Meinong was directly relevant to Shapiro's work in AI, providing an intensional semantics for his SNePS semantic-network processing system [Shapiro and Rapaport, 1987], [Shapiro and Rapaport, 1991]. And then I realized that the discovery of quasi-indexicals ('he himself', 'she herself', etc.; [Castañeda, 1966]) by my dissertation advisor,

2 Preliminary Questions

However, before investigate what computer science is, it's worth asking some preliminary questions.

2.1 What Is the Name of this Discipline?

Should we call the discipline 'computer science' (which seems to assume that it is the *science* of a certain kind of *machine*), or 'computer *engineering*' (which seems to assume that it is *not* a science, but a branch of engineering), or '*computing* science' (which seems to assume that it is the science of what those machines do), or 'informatics' (which suggests that it is a mathe*matic*al discipline concerned with *infor*mation)?

In this essay—but only for convenience—I call it 'computer science'. However, by doing so, I do not presuppose that it is the science of computers. Think of the subject as being called by a 15-letter word 'computerscience' that may have as little to do with computers or science as 'cattle' has to do with cats. Or, to save space and suppress presuppositions, just think of it as "CS".

2.2 Why Ask what CS Is?

There are both academic and philosophical motivations for trying to define CS.

2.2.1 Academic Motivations

There is the *political* question of where to locate a CS department: In a college, faculty, or school of *(arts and) science*? Of *engineering*? Or in its *own* college, faculty, or school (perhaps of informatics, along with communications and library science)?

There is the pedagogical question of what to teach in an introductory course: Programming? Computer literacy? The mathematical theory of computation? Or

Hector-Neri Castañeda, could repair a "bug" in a knowledge-representation theory that Shapiro had developed with another convert to computer science (from psychology), Anthony S. Maida [Maida and Shapiro, 1982]. This work was itself debugged with the help of yet another convert (from English), my doctoral student Janyce M. Wiebe [Rapaport et al., 1997]. My work with Shapiro and our SNePS Research Group at Buffalo enabled me to rebut [Searle, 1980] using "syntactic semantics" [Rapaport, 1986], [Rapaport, 1988], [Rapaport, 1995], [Rapaport, 2000], [Rapaport, 2012]. Both of these projects, as well as one of my early Meinong papers [Rapaport, 1981], led me, together with another doctoral student (Karen Ehrlich) and (later) a colleague from Buffalo's Department of Learning and Instruction (Michael W. Kibby), to develop a computational and pedagogical theory of vocabulary acquisition from context [Rapaport and Kibby, 2007], [Rapaport and Kibby, 2014].

an introduction to several different branches of CS, including, perhaps, some of its history?

And there is the *publicity* question: How should a CS department advertise itself so as to attract good students? How should the discipline advertise itself so as to encourage primary- or secondary-school students to consider it as something to study in college or to consider it as an occupation? How should it advertise itself so as to attract more women and minorities to the field? How should it advertise itself to the public at large, so that ordinary citizens might have a better understanding of what CS is?

Different motivations may yield different definitions.

2.2.2 Philosophical Motivations

The *philosophical* question concerns what CS "really" is. Is it like some other academic discipline (mathematics, physics, engineering)? Or is it *sui generis*?

To illustrate this difference, consider two very different comments by two Turing-award–winning computer scientists (as cited in [Gal-Ezer and Harel, 1998, p. 79]): Marvin Minsky, a founder of artificial intelligence, once said:

> Computer science has such *intimate relations* with so many other subjects that *it is hard to see it as a thing in itself.*
> [Minsky, 1979, my italics]

On the other hand, Juris Hartmanis, a founder of computational complexity theory, has said:

> Computer science *differs* from the known sciences so deeply that it has to be viewed as *a new species among the sciences.*
> [Hartmanis, 1993, p. 1; my italics] (cf. [Hartmanis, 1995, p. 10])

3 Two Kinds of Definitions

3.1 An Extensional Definition of CS

As with most non-mathematical concepts, there are probably no necessary and sufficient conditions for being CS. At best, the various branches of the discipline share only a family resemblance. If no intensional definition can be given in terms of necessary and sufficient conditions, perhaps an extensional one can: Perhaps CS is simply whatever computer scientists do: "Computing has no nature. It is what it is because people have made it so" [Mahoney, 2011, p. 109].

So, what do computer scientists do? Ordered from the most to the least abstract, this might range from the abstract mathematical theories of computation, computational complexity, and program development; through software engineering, operating systems, and AI; to computer architecture, chip design, networks, and social uses of computers. But this is less than satisfactory as a *definition*.

3.2 Intensional Definitions

In the absence of necessary and sufficient conditions or an extensional definition, we can ask what the *methodology* of CS is: Is it a methodology used elsewhere? Or is it a new methodology? And then we can ask what its *object* of study is: Does it study something that other disciplines also study? Or does it study something new? And is its object of study unique to CS?

As for methodology, CS has been said to be:

- an *art form*
  ([Knuth, 1974a, p. 670] has said that programs can be beautiful),
- an *art and science*
  ("Science is knowledge which we understand so well that we can teach it to a computer; and if we don't fully understand something, it is an art to deal with it. ... [T]he process of going from an art to a science means that we learn how to automate something" [Knuth, 1974a, p. 668]),
- a *liberal art* [Perlis, 1962, p. 210], [Lindell, 2001]
  (along the lines of the classical liberal arts of logic, math, or astronomy),
- a branch of *mathematics* [Dijkstra, 1974],
- a *natural science* [McCarthy, 1963], [Newell et al., 1967], [Shapiro, 2001],
- an *empirical study* of the artificial [Simon, 1996b],
- a combination of *science and engineering*
  [Hartmanis, 1993], [Hartmanis, 1995], [Loui, 1995],
- just *engineering* [Brooks, 1996],
- or—generically—a "study"

But a study of what? Here is an alphabetical list of some of the objects that it "traffics" in (to use [Barwise, 1989]'s term): algorithms, automation, complexity, computers, information, intelligence, numbers (and other mathematical objects), problem solving, procedures, processes, programming, symbol strings.

It is now time to look at some answers to our title question in more detail.

4 CS Is the Science of Computers

Allen Newell, Alan Perlis, and Herbert Simon argued that CS is exactly what its name suggests:

> Wherever there are phenomena, there can be a science to describe and explain those phenomena. ... There are computers. Ergo, computer science is the study of computers. [Newell et al., 1967, p. 1373].

This argument is actually missing a premise to the effect that the science of computers (which the first two premises imply the existence of) is CS and not some other discipline.

[Loui, 1987, p. 175] has objected to the first premise, noting that there are toasters, but no *science* of toasters. Another objection to the first premise, explicitly considered by Newell, Perlis, & Simon, is that science studies only *natural* phenomena, but that computers are *non*-natural *artifacts*. They replied that there are also sciences of artifacts. But one could respond in other ways: Where is the dividing line between nature and artifice, anyway? Are birds' nests artificial? As [Mahoney, 2011, p. 159ff] observes, not only are artifacts part of nature, we use them to study nature; indeed, nature itself might be computational in nature (so to speak).

Another objection that they consider is to the missing premise, that the science of computers is not CS but some other subject: electrical engineering, or math, or, perhaps, psychology. They reply that CS overlaps each of these, but that no single discipline subsumes *all* of CS. Of course, this reply assumes that CS itself *is* a cohesive whole, which the extensional characterization in §3.1 seems to belie. One of my department's deans once suggested that CS would eventually dissolve: The computer engineers would rejoin the EE department, the complexity theorists would join the math department, my AI colleagues might go into psychology, I would go back into philosophy, and so on. (In much the same way, microscopy dissolved into microbiology, optical engineering, etc. [Boorstin, 1983, p. 376]; see further discussion in [Rapaport, 2017, §3.5.3]).

The most significant objection that they consider is that CS studies something besides computers, namely, algorithms. Their reply is also significant: They change their definition! They conclude that CS is the science of computers *and surrounding phenomena, including algorithms*.

5 CS Studies Algorithms

Donald Knuth starts his definition, largely without any argument other than a recitation of its history, roughly where Newell, Perlis, & Simon end theirs: "[C]omputer science is ... the study of *algorithms*" [Knuth, 1974b, p. 323]. He cites, approvingly, a statement by the computer scientist George E. Forsythe that the central question of CS is: What can be automated? (On that question, see §14.1.1.1, below.)

Knuth goes on to point out, however, that you need computers in order to properly study algorithms, because "human beings are not precise enough nor fast enough to carry out any but the simplest procedures" [Knuth, 1974b, p. 323]. Are computers really necessary? Do you *need* a compass and straightedge to study geometry? (Hilbert probably didn't think so.) Do you *need* a microscope to study biology? (Watson and Crick probably didn't think so.) On the other hand, "deep learning" algorithms do seem to need computers in order to determine if they will really do what they are intended to do, and do so in real time [Lewis-Kraus, 2016]. (We'll return to this in §11.)

So, just as Newell, Perlis, & Simon said that CS is the study of computers *and related phenomena* ***such as algorithms***, Knuth says that it is the study of algorithms *and related phenomena* ***such as computers***! Stated a bit more bluntly, Newell, Perlis, & Simon's definition comes down to this: CS is the science of *computers and algorithms*. Knuth's definition comes down to this: CS is the study of *algorithms and computers*. Ignoring for now the subtle difference between "science" and "study", what we have here are extensionally equivalent, but intensionally distinct, definitions. Shades of the blind men and the elephant!

To be fair, however, some ten years later, [Knuth, 1985, pp. 170–171] backed off from the "related phenomena" definition, more emphatically defining CS as "primarily the study of algorithms", because he "think[s] of algorithms as encompassing the whole range of concepts dealing with well-defined processes, including the structure of data that is being acted upon as well as the structure of the sequence of operations being performed", preferring the name 'algorithmics' for the discipline. He also suggested that what computer scientists have in common (and that differentiates them from people in other disciplines) is that they are all "algorithmic thinkers" [Knuth, 1985, p. 172]. (We'll return to this notion in §13.4.)

6 CS Studies Information

Others say "A plague on both your houses": CS is *not* the study of computers *or* of algorithms, but of *information*: [Forsythe, 1967, p. 3, my italics] said that CS is "the art and science of representing and processing *information* and, in particular, processing information with the logical engines called automatic digital computers." Peter J. Denning defined it as "the body of knowledge dealing with the design, analysis, implementation, efficiency, and application of processes that transform *information*" [Denning, 1985, p. 16, my italics]. Jon Barwise said that computers are best thought of as "information processors", rather than as numerical "calculators" or as "devices which traffic in formal strings ... of meaningless symbols" [Barwise, 1989, pp. 386–387]. And [Hartmanis and Lin, 1992, p. 164] define CS this way:

> What is the object of study [of CS and engineering]? For the physicist, the object of study may be an atom or a star. For the biologist, it may be a cell or a plant. But computer scientists and engineers focus on information, on the ways of representing and processing information, and on the machines and systems that perform these tasks.

Presumably, those who study "the ways of representing and processing" are the scientists, and those who study "the machines and systems" are the engineers. And, of course, it is not just information that is studied; there are the usual "related phenomena": Computer science studies how to *represent* and (algorithmically) *process* information, as well as the *machines* and systems that do this.

Simon takes an interesting position on the importance of computers as information processors [Simon, 1977, p. 1186]: He discusses two "revolutions": The first was the Industrial Revolution, which "substitut[ed] ... mechanical energy for the energy of man [sic] and animal". The second was (were?) the Information Revolution(s), beginning with "written language", then "the printed book", and now the computer. He then points out that "The computer is a device endowed with powers of utmost generality for processing symbols." So, *pace* Barwise, the computer is an information processor *because* information is encoded in symbols.

But here the crucial question is: What is information? The term 'information' as many people use it informally has many meanings: It could refer to Claude Shannon's mathematical theory of information [Shannon, 1948]; or to Fred Dretske's or Kenneth Sayre's philosophical theories of information [Dretske, 1981], [Sayre, 1986]; or to several others. (For a survey, see [Piccinini, 2015, Ch. 14].)

As I noted in §1, the philosophy of information is really a separate (albeit closely related!) topic from the philosophy of computer science. But, if ‘information’ isn’t intended to refer to some specific theory, then it seems to be merely a vague synonym for ‘data’ (itself a vague term!). As Michael Rescorla observes, “Lacking clarification [of the term ‘information’], the description [of “computation as ‘information processing’ ”] is little more than an empty slogan” [Rescorla, 2015, §6.1].

And Gualtiero Piccinini has made the stronger claim that computation is distinct from information processing in *any* sense of ‘information’. He argues, e.g., that semantic information *requires* representation, but computation does *not*; so, computation is distinct from semantic information processing [Piccinini, 2015, Ch. 14, §3].

7 CS *Is* a Natural Science (of *Procedures*)

Then there are those who agree that CS is a natural science, but not of computers, algorithms, *or* information: Stuart C. Shapiro agrees with Newell, Perlis, & Simon that CS is a science, but he differs on what it is a science of, siding more with Knuth, but not quite: “Computer Science is a *natural science* that studies *procedures*” [Shapiro, 2001, my italics]. Procedures are not natural *objects*, but they are measurable natural *phenomena*, in the same way that events are not (natural) “objects” but are (natural) “phenomena”. On this point, [Denning, 2007] cites examples of the “discovery” of “information processes in the deep structures of many fields”: biology, quantum physics, economics, management science, and even the arts and humanities, concluding that “computing is now a natural science”, not (or no longer?) “a science of the artificial”. So, potential objections that sciences only study natural phenomena are avoided.

For Shapiro, procedures include, but are not limited to, algorithms. Whereas algorithms are typically considered to be precise, to halt, and to produce correct solutions, the more general notion allows for variations on these themes: (1) Procedures (as opposed to algorithms) may be imprecise, such as in a recipe. Does CS really study things like recipes? According to Shapiro (personal communication), the answer is ‘yes’: An education in CS should help you write a better cookbook, because it will help you understand the nature of procedures better! ([Sheraton, 1981] discusses the difficulties of writing recipes.) (2) Procedures need not halt: A procedure might go into an infinite loop either by accident or, more importantly, on purpose, as in an operating system or a program that computes the infinite decimal expansion of π. (3) Nor do they have to produce a correct solution: A chess procedure does not always play optimally.

And CS *is* a *science*, which, like any science, has both theoreticians (who study the limitations on, and kinds of, possible procedures) as well as experimentalists.

And, as [Newell and Simon, 1976] suggest in their discussion of empirical results (see §8, below), there are "fundamental principles" of CS as a science. Newell & Simon cite two: (1) The Physical Symbol System Hypothesis (a theory about the nature of symbols in the context of computers) and (2) Heuristic Search (a problem-solving method). Shapiro cites two others: (1) the Computability Thesis and (2) the Boehm-Jacopini Theorem that codifies "structured programming" [Böhm and Jacopini, 1966].

Moreover, Shapiro says that computer science is *not just* concerned with algorithms and procedures that manipulate abstract information, but also with procedures that are linked to sensors and effectors that allow computers to operate in the real world. Procedures are, or could be, carried out in the real world by physical agents, which could be biological, mechanical, electronic, etc. Where do computers come in? According to Shapiro, a computer is simply "a general-purpose procedure-following machine". (But does a computer "follow" a procedure, or merely "execute" it?)

Several pleas for elaboration can be urged on Shapiro: Does his view de-emphasize the role of computers in CS, or is it merely a version of the "surrounding phenomena" viewpoint (as with Knuth's view that CS is the study of the phenomena surrounding *algorithms*)?[6] Does the emphasis on procedures (rather than algorithms) lead us into the fraught territory of "hypercomputation" [Rapaport, 2017, Ch. 11]? (We'll return to procedures in §13.3.)

8 CS Is *Not* a *Natural* Science

In 1967, Simon joined with Newell and Perlis to argue that CS was the natural science of (the phenomena surrounding) *computers*. Two years later, in his classic book *The Sciences of the Artificial* [Simon, 1996b, 3rd edition], he said that it was a natural science of *the artificial*: Natural science studies things in the world, but he was careful not to say that the "things" must be "natural"! "The central task of a natural science is ... to show that complexity, correctly viewed, is only a mask for simplicity; to find pattern hidden in apparent chaos" [Simon, 1996b, p. 1]. Indeed, "The world we live in today is much more a[n] ... artificial world than it is a natural world. Almost every element in our environment shows evidence of human artifice" [Simon, 1996b, p. 2]. So, (natural) science can study artifacts; the "sciences of the artificial" are natural sciences.

And then, in a classic paper from 1976, Newell and Simon updated their earlier characterization. Instead of saying that CS is the *science* of (the phenomena surrounding) computers, they now said that it is the "*empirical*" "*study*" of

[6] Knowing Shapiro, I strongly suspect that it is the latter.

those phenomena, “not just the hardware, but *the programmed, living machine*” [Newell and Simon, 1976, pp. 113, 114; my italics].

CS is not a *science* (in the classic sense) on the grounds that it doesn’t always strictly follow the scientific (or “experimental”) method. E.g., often *one* experiment will suffice to answer a question in CS, whereas in other sciences, *numerous* experiments have to be run. However, CS, like science, is *empirical*—because programs running on computers are *experiments*, though not necessarily like experiments in other experimental sciences. In fact, one difference between CS and other experimental sciences is that, in CS, the chief objects of study (the computers and the programs) are not “black boxes”. Most natural phenomena are things whose internal workings we cannot see directly but must infer from experiments we perform on them. But we know exactly how and why computers and computer programs behave as they do (they are “glass boxes”, so to speak), because *we* (not nature) designed and built them. So, we can understand them in a way that we cannot understand more “natural” things. (However, although this is the case for “classical” computer programs, it is not the case for artificial-neural-network programs: “A neural network, however, was a black box” [Lewis-Kraus, 2016, §4]; see the comments about Google Translate in §11, below.)

By “programmed, living machines”, they meant computers that are actually running programs—*not just* the static machines sitting there waiting for someone to use them (computers without programs), *nor* the static programs just sitting there on a piece of paper waiting for someone to load them into the computer, nor the algorithms just sitting there in someone’s mind waiting for someone to express them in a programming language—but *processes that are actually running on a computer*. A *program* might be a static piece of text or the static way that a computer is hardwired. A *process* is a dynamic entity—the program in the “process” of actually being executed by the computer.

However, to study “programmed living machines”, we certainly do need to study the algorithms that they are executing. After all, we need to know what they are doing; i.e., it seems to be necessary to know what algorithm a computer is executing. On the other hand, in order to study an algorithm, it does not seem to be *necessary* to have a computer around that can execute it or to study the computer that is running it. It can be helpful and valuable to study the computer and to study the algorithm actually being run on the computer, but the mathematical study of algorithms and their computational complexity doesn’t *need* the computer. That is, the algorithm can be studied as a mathematical object, using only mathematical techniques, without necessarily executing it. It may be very much more convenient, and even useful, to have a computer handy, as Knuth notes, but it does not seem to be necessary. If that’s so, then it would seem that *algorithms* are really the essential object of study of CS: Both views require algorithms, but only one requires computers. (We'll see a counterargument in §11.)

9 CS Is Engineering, *Not* Science

The software engineer Frederick P. Brooks, Jr., says that CS isn't science—which he calls "analytic"—because, according to him, it is not concerned with the "discovery of facts and laws" [Brooks, 1996]. Instead, he argues that it is "an engineering discipline". Computer scientists are "concerned with *making things*": with physical tools such as computers and with abstract tools such as algorithms, programs, and software systems for others to use; the computer scientist is a *toolmaker*. Computer science, he says, is concerned with the usefulness and efficiency of the tools it makes; it is *not*, he says, concerned with newness for its own sake (as scientists are). So, "the discipline we call 'computer science' " is really the "synthetic"—i.e., the *engineering*—discipline that is concerned with computers.

Here is his argument [Brooks, 1996, pp. 61–62]:

1. "[A] science is concerned with the *discovery* of facts and laws."
2. "[T]he scientist *builds in order to study*;
   the engineer *studies in order to build*.
3. The purpose of engineering is to build things.
4. Computer scientists "are concerned with *making things*, be they computers, algorithms, or software systems".
5. ∴ "the discipline we call 'computer science' is in fact not a science but a *synthetic*, an engineering, discipline."

Let's accept premise 1 for now; it seems reasonable enough.[7]

The point of the second premise is this: If a scientist's goal is to discover facts and laws—i.e., to study rather than to build—then anything built by the scientist is only built for that ultimate purpose. But building is the ultimate goal of engineering, and any studying (or discovery of facts and laws) that an engineer does along the way to building something is merely done for that ultimate purpose. For science, building is a side-effect of studying; for engineering, studying is a side-effect of building. Both scientists and engineers, according to Brooks, build and study, but each focuses more on one than the other. (Does this remind you of the algorithms-vs.-computers dispute in §§4–5?)

[7] I discuss this issue in more detail in [Rapaport, 2017, Ch. 4].

The second premise supports the third, which defines engineering as a discipline whose goal is to build things, i.e., a "synthetic"—as opposed to an "analytic"—discipline. "We speak of engineering as concerned with 'synthesis,' while science is concerned with 'analysis' " [Simon, 1996b, p. 4]. "Where physical science is commonly regarded as an analytic discipline that aims to find laws that generate or explain observed phenomena, CS is predominantly (though not exclusively) synthetic, in that formalisms and algorithms are created in order to support specific desired behaviors" [Hendler et al., 2008, p. 63]. As with his claim about the nature of science in the first premise, the accuracy of Brooks's notion of engineering is a topic for another day.[8] So, let's also assume the truth of the second and third premises for the sake of the argument.

Clearly, if the fourth premise is true, then the conclusion will follow validly (or, at least, it will follow that computer scientists belong on the engineering side of the science–engineering, or studying–building, spectrum). But is it really the case that computer scientists are (only? principally?) concerned with building or "making things"? And, if so, what kind of things?

Moreover, computer scientists *do* discover and analyze facts and laws: Consider the theories of computation and of computational complexity, and the "fundamental principles" cited at the end of §7, above. Computer scientists devise *theories* about how to build things, and they try to *understand* what they build. All of this seems to be more science than engineering.

Interestingly, Brooks seems to suggest that computer scientists *don't* build computers, even if that's what he says in the conclusion of his argument! He says that "Even when we build a computer the computer scientist designs only the abstract properties—its architecture and implementation. Electrical, mechanical, and refrigeration engineers design the realization" [Brooks, 1996, p. 62, col. 1]. I think this passage is a bit confused: Briefly, I think the "abstract properties" *are* the design *for* the realization; the engineers *build* the realization (they don't *design* it) [Rapaport, 1999], [Rapaport, 2005a]. But it makes an interesting point: Brooks seems to be saying that computer scientists only design *abstractions*, whereas other (real?) engineers *implement them in reality.* This is reminiscent of the distinction between the relatively abstract *specifications* for an algorithm (which typically lack detail) and its relatively concrete (and highly detailed) implementation in a computer *program*. Brooks (following [Zemanek, 1971]) calls CS "the engineering of abstract objects": If engineering is a discipline that builds, then what computer-science-qua-engineering builds is *implemented abstractions.*

[8] Covered in [Rapaport, 2017, Ch. 5].

10 Science *xor* Engineering?

So, is CS a science of some kind (natural or otherwise), or is it not a science at all, but some kind of engineering? Here, we would be wise to listen to two skeptics about the exclusivity of this choice:

> Let's remember that there is only one nature—the division into science and engineering, and subdivision into physics, chemistry, civil and electrical, is a human imposition, not a natural one. Indeed, the division is *a human failure*; it reflects *our limited capacity to comprehend the whole.* That failure impedes our progress; it builds walls just where the most interesting nuggets of knowledge may lie. [Wulf, 1995, p. 56; my italics]

> Debates about whether [CS is] science or engineering can ... be counterproductive, since we clearly are *both, neither, and more* ... . [Freeman, 1995, p. 27, my italics]

11 CS as "Both"

Could CS be both science *and* engineering— perhaps the *science* of computation and the *engineering* of computers—i.e., the study of the "programmed living machine"?

It certainly makes no sense to have a computer without a program. It doesn't matter whether the program is *hardwired* (in the way that a Turing machine is); i.e., it doesn't matter whether the computer is a *special*-purpose machine that can only do one task. The program is not separable from the machine; it is built into its structure. And it doesn't matter whether the program is a piece of *software* (like a program inscribed on a universal Turing machine's tape); i.e., it doesn't matter whether the computer is a *general*-purpose machine that can be loaded with different "apps" allowing the *same* machine to do many *different* things. It is simply the case that, *without a program, the computer wouldn't be able to* ***do anything.*** So, insofar as CS is about computers and hence is engineering, it must also be about computation and hence a science (at least, a mathematical science).

But it also makes little sense to have a program without a computer to run it on. Yes, you can study the program mathematically (e.g., try to verify it) or study its computational complexity [Loui, 1996], [Aaronson, 2013], etc.).

> The ascendancy of logical abstraction over concrete realization has ever since been a guiding principle in computer science, which has

> kept itself organizationally almost entirely separate from electrical engineering. The reason it has been able to do this is that computation is primarily a logical concept, and only secondarily an engineering one. To compute is to engage in formal reasoning, according to certain formal symbolic rules, and *it makes no logical difference how the formulas are physically represented, or how the logical transformations of them are physically realized.*
> [Robinson, 1994, p. 12, my italics]

But what good would it be (for that matter, what fun would it be!) to have, say, a program for passing the Turing test that never had an opportunity to pass it? Thus, *without a computer, the program wouldn't be able to actually do anything.* So, insofar as CS is about computation and hence is science, it should (must?) also be about computers and hence an engineering discipline.

So, *computers require programs* in order for the *computer* to do anything, and *programs require computers* in order for the *program* to actually be able to do anything. This is reminiscent of Kant's slogan that "Thoughts without content are empty, intuitions without concepts are blind. ... The understanding can intuit nothing, the senses can think nothing. Only through their union can knowledge arise" [Kant, 1787, p. 93 (A51/B75)]. Similarly, we can say: "Computers without programs are empty; programs without computers are blind. Only through the union of a computer with a program can computational processing arise." A good example of this is the need for computers to test certain "deep learning" algorithms that Google used in their Translate software: Without enough computing power, there was no way to prove that their connectionist programs would work as advertised [Lewis-Kraus, 2016, §2]. So, CS must be both a science (that studies algorithms) and an engineering discipline (that builds computers).

But we need not be concerned with these two fighting words, because, fortunately, there are two very convenient terms that encompass both: 'scientific' and 'STEM'. Surely, not only natural science, but also engineering, not to mention "artificial science", "empirical studies", and mathematics are all *scientific*. And, lately, NSF and the popular press have taken to referring to "STEM" disciplines—science, technology, engineering, and mathematics—precisely in order to have a single term to emphasize their similarities and interdependence, and to avoid having to try to spell out differences among them.[9]

So let's agree for the moment that CS might be *both* science *and* engineering. What about Freeman's other two options: *neither* and *more*?

[9] Nothing should be read into the ordering of the terms in the acronym: The original acronym was the less mellifluous 'SMET'! And educators, perhaps with a nod to Knuth's views, have been adding the arts, to create 'STEAM' (http://stemtosteam.org/).

12 CS as "More"

12.1 CS Is a New Kind of *Engineering*

Michael Loui defines CS as "the theory, design, and analysis of algorithms for processing [i.e., for storing, transforming, retrieving, and transmitting] information, and the implementations of these algorithms in hardware and in software" [Loui, 1987, p. 176]. He argues that CS is "a new species of engineering" [Loui, 1995, p. 1]. He first argues that CS is an engineering discipline on the grounds that engineering (1) is concerned with what *can* exist (as opposed to what *does* exist), (2) "has a scientific basis", (3) is concerned with "design", (4) analyzes "trade-offs", and (5) has "heuristics and techniques". "Computer science has all the significant attributes of engineering"; therefore, CS is a branch of engineering [Loui, 1987, p. 176].

Let's consider each of these "significant attributes": First, his justification that CS is *not* "concerned with ... what *does* exist" is related to the claim that CS is not a natural science, but a science of human-made artifacts. We have already considered two possible objections to this: First, insofar as procedures are natural entities, CS—as the study of procedures—*can* be considered a natural science. Second, insofar as some artifacts—such as bird's nests, beehives, etc.—are natural entities, studies of artifacts can be considered to be scientific.

Next, according to Loui, the "scientific basis" of CS is mathematics. The scientific basis of "traditional engineering disciplines such as mechanical engineering and electrical engineering" is physics. This is what makes it "new"; we'll come back to this.

According to Loui, engineers apply the principles of the scientific base of their engineering discipline to "design" a product: "[A] computer specialist applies the principles of computation to design a digital system or a program" [Loui, 1987, p. 176]. But not all computer scientists (or "specialists") design systems or programs; some do purely theoretical work. And, in any case, if the scientific basis of CS is mathematics, then why does Loui say that computer "specialists" apply "the principles of *computation*"? I would have expected him to say that they apply the principles of *mathematics*. Perhaps he sees "computation" as being a branch of mathematics. Or perhaps he doesn't think that the abstract mathematical theory of computation is part of CS, but that seems highly unlikely, especially in view of his definition of computer science as including the theory and analysis of algorithms. It's almost as if he sees computer *engineering* as standing to computer science in the same way that mechanical or electrical engineering stand to physics. But then it is not computer science that is a branch of engineering.

Let's turn briefly to trade-offs: "To implement algorithms efficiently, the designer of a computer system must continually evaluate trade-offs between resources" such as time vs. space, etc. [Loui, 1987, p. 177]. This is true, but doesn't support his argument as well as it might. For one thing, it is not only system designers who evaluate such trade-offs; so do theoretical computer scientists—witness the abstract mathematical theory of complexity. And, as noted above, not all computer scientists design such systems. So, at most, it is only those who do who are doing a kind of engineering.

Finally, as for heuristics, Loui seems to have in mind rough-and-ready "rules of thumb" rather than formally precise theories in the sense of [Newell and Simon, 1976]. (See §14.1.3, below, for more on this kind of heuristics.) Insofar as engineers rely on such heuristics [Koen, 1988], and insofar as some computer scientists also rely on them, then those computer scientists are doing something that engineers also do. But so do many other people: Writers surely rely on rule-of-thumb heuristics ("write simply and clearly"); does that make them engineers? This is probably his weakest premise.

The second part of Loui's argument is to show how CS is a "new" kind of engineering [Loui, 1995, p. 31]:

1. "[E]ngineering disciplines have a scientific basis".
2. "The scientific fundamentals of computer science ... are rooted ... in mathematics."
3. "Computer science is therefore a *new* kind of engineering." (italics added)

This argument can be made valid by adding two missing premises:

A. Mathematics is a branch of science.

B. No other branch of engineering has mathematics as its basis.

We can assume from his first argument that CS *is* a kind of engineering. So, from that and 1, we can infer that CS (as an engineering discipline) must have a scientific basis. We need premise A so that we can infer that the basis of CS (which, by 2, is mathematics) is indeed a scientific one. Then, from B, we can infer that CS must differ from all other branches of engineering. It is, thus, mathematical engineering.

However, despite these arguments, Loui also says this: "It is impossible to define a reasonable boundary between the disciplines of computer science and computer engineering. *They are the same discipline*" [Loui, 1987, p. 178, my italics]. But doesn't that contradict the title of his essay ("Computer Science Is an Engineering Discipline")?

12.2 CS Is a New Kind of *Science*

Recall that Hartmanis said that "computer science differs from the known sciences so deeply that it has to be viewed as a new species among the sciences" [Hartmanis, 1993, p. 1]. First, Hartmanis comes down on the side of CS being a science: It is a "new species *among the sciences*". A chimpanzee is a different species from a tiger "among the animals", but they are both animals.

But what does it mean to be "a new species" of science? Both chimps and tigers are species of animals, and both lions and tigers are species within the genus *Panthera*. Is the relation of computer science to other sciences more like the relation of chimps to tigers (relatively distant) or lions to tigers (relatively close)? A clue comes in Hartmanis's next sentence:

> This view is justified by observing that theory and experiments in computer science play a different role and do not follow the classic pattern in physical sciences. [Hartmanis, 1993, p. 1]

This strongly suggests that CS is not a *physical* science (such as physics or biology), and Hartmanis confirms this suggestion on p. 5: "computer science, *though not a physical science*, is indeed a science" (my italics; cf. [Hartmanis, 1993, p. 6], [Hartmanis, 1995, p. 11]). The non-physical sciences are typically taken to include at least the social sciences (such as psychology) and mathematics. So, it would seem that the relation of CS to other sciences is more like that of chimps to tigers: distantly related species of the same, high-level genus. And, moreover, it would seem to put computer science either in the same camp as (either) the so*cial sciences or mathematics, or else in a brand-new camp of its own, i.e.,* sui generis.

Hartmanis offers this definition of CS:

> At the same time, it is clear that *the objects of study in computer science are information and the machines and systems which process and transmit information.* From this alone, we can see that CS is concerned with the abstract subject of information, which gains reality only when it has a physical representation, and the man-made devices which process the representations of information. The goal of computer science is to endow these information processing devices with as much intelligent behavior as possible.
> [Hartmanis, 1993, p. 5, my italics] (cf. [Hartmanis, 1995, p. 10])

Although it may be "clear" to Hartmanis that information (an "abstract subject") is (one of) the "objects of study in computer science", he does not share his reasons for that clarity. Since, as we have seen, others seem to disagree that CS is the study of information (e.g., it could be the study of computers or the study of algorithms), it seems a bit unfair for Hartmanis not to defend his view. But he cashes out this promissory note in [Hartmanis, 1995, p. 10], where he says that "what sets [CS] apart from the other sciences" is that it studies "processes [such as information processing] that are not directly governed by physical laws". And why are they not so governed? Because "information and its transmission" are "abstract entities" [Hartmanis, 1995, p. 8]. This makes computer science sound very much like mathematics. That is not unreasonable, given that it was this aspect of CS that led Hartmanis to his ground-breaking work on computational complexity, an almost purely mathematical area of CS.

But it's not just information that is the object of study; it's also information-processing machines, i.e., computers. Computers, however, don't deal directly with information, because information is abstract, i.e., non-physical. For one thing, this suggests that, insofar as CS is a new species of *non-physical* science, it is not a species of *social* science: Despite its name, the "social" sciences deal with pretty physical things: societies, people, speech, etc.

Hartmanis explicitly says that CS *is* a science and is *not* engineering, but his comments imply that it is both. I don't think he can have it both ways. This is remiscent of the dialogue between Newell, Perlis, & Simon on the one hand, and Knuth on the other. Both Loui and Hartmanis agree that computer science is a new kind of something or other; each claims that the scientific and mathematical aspects of it are central; and each claims that the engineering and machinery aspects of it are also central. But one *calls* it 'science', while the other *calls* it 'engineering'. Again, it seems to be a matter of point of view.

A very similar argument (that does not give credit to Hartmanis!) that CS is a new kind of *science* can be found in [Denning and Rosenbloom, 2009]. We'll look at some of what they have to say in §13.1.

13 CS as "Neither"

And now for some things completely different . . .

13.1 CS Has Its Own Paradigm

Hartmanis argued that CS was *sui generis* ***among the sciences***. Denning & Peter A. Freeman offer a slightly stronger argument to the effect that CS is neither science, engineering, nor math; rather CS has a "unique paradigm" [Denning and Freeman, 2009, p. 28].

But their position is somewhat muddied by their claim that "computing is a fourth great domain *of science* alongside the physical, life, and social sciences" [Denning and Freeman, 2009, p. 29, my italics]. That implies that CS is a science, though of a different kind, as Hartmanis suggested.

It also leaves mathematics out of science! In a related article published three months earlier in the same journal, Denning & Paul S. Rosenboom assert without argument that "mathematics … has traditionally not been considered a science" [Denning and Rosenbloom, 2009, p. 28]. Denying that math is a science allows them to avoid considering CS as a *mathematical* science (an option that we explore in [Rapaport, 2017, Ch. 3, §3.10.2]).

In any case, to justify their conclusion that CS is truly *sui generis*, Denning & Freeman need to show that it is not a physical, life, or social science. Denning & *Rosenbloom* say that "none [of these] studies computation per se" [Denning and Rosenbloom, 2009, p. 28]. This is only half of what needs to be shown; it also needs to be shown that CS doesn't study physical, biological, or social entities. Obviously, it does study such things, though that is not its focus. As they admit, CS is "used extensively in all the domains" [Denning and Rosenbloom, 2009, p. 28]; i.e., computation is used by scientists in these domains as a tool.

So, what makes CS different? Denning & *Freeman* give a partial answer:

> The central focus of the computing paradigm can be summarized as information processes—natural or constructed processes that transform information. … [T]he computing paradigm … is distinctively different because of its central focus on information processes. [Denning and Freeman, 2009, pp. 29–30]

This is only a partial answer, because it only discusses the *object of study* (which, as we saw in §6, is somewhat vague).

The rest of their answer is provided in a table showing the methodology of CS (Table 2, p. 29), which comes down to their version of "computational thinking" [Denning and Freeman, 2009, p. 30]. We'll explore what that is in §13.4. Denning & Freeman's version of it is close to what I will present as "synthetic" computational thinking in §14.1.1.1.

13.2 CS Is the Study of Complexity

It has been suggested that CS is the study of *complexity*—not just the mathematical subject of "computational complexity", but complexity in general and in all of nature. [Ceruzzi, 1988, pp. 268–270] ascribes this to Jerome Wiesner [Wiesner, 1958]. But all Wiesner says is that "Information processing systems are but one facet of … communication sciences … that is, the study of … the problems of organized complexity' " (quoted in [Ceruzzi, 1988, p. 269]). But even if computer science is part of a larger discipline ("communication sciences"?) that studies complexity, it doesn't follow that CS itself *is* the study of complexity.

According to Ceruzzi, Edsgar Dijkstra also held this view: "programming, when stripped of all its circumstantial irrelevancies, boils down to no more and no less than very effective thinking so as to avoid unmastered complexity" [Dijkstra, 1975, §4, p. 3]. It is hierarchical structure that "offers a standard way to handle complexity" [Lamport, 2012, p. 16]:

> [P]rograms are built from programs. ... Programs are compilations in another sense as well. Even the smallest sub-program is also a compilation of sub-components. Programmers construct sub-programs by assembling into a coherent whole such discrete program elements as data, data structures, and algorithms. The "engineering" in software engineering involves knowing how to assemble these components to produce the desired behavior.
> [Samuelson et al., 1994, pp. 2326–2327]

The idea that a complex program is "just" a construction from simpler things, each of which—recursively—can be analyzed down to the primitive operations and data structures of one's programming system (for a Turing machine, these would be the operations of printing and moving, and data structures constructed from '0's and '1's) is, first, the underlying way in which complexity can be dealt with and, second, where engineering (considered as a form of construction) comes into the picture.

But, again, at most this makes the claim that *part* of computer science is the study of complexity. CS certainly offers many techniques for handling complexity: structured programming, abstraction, modularity, hierarchy, top-down design, stepwise refinement, object-oriented programming, recursion, etc. So, yes, CS is one way—perhaps even the best way—to *manage* (or *avoid*) complexity, not that it is *the study of* it. What's missing from Dijkstra's argument, in any case, is a premise to the effect that computer science is the study of programming, but Dijkstra doesn't say that, either in [Dijkstra, 1975] or in [Dijkstra, 1976], the document that Ceruzzi says contains that premise. ([Khalil and Levy, 1978], however, do make that claim.)

But [Denning et al., 1989, p. 11] point out that viewing " 'computer science [as] the study of abstraction and the mastering of complexity' ... also applies to physics, mathematics, or philosophy"; no doubt many other disciplines also study complexity. So *defining* CS the study of complexity doesn't seem to be right.

13.3 CS Is the *Philosophy*(!) of Procedures

Could CS be the study of procedures, yet be a branch of *philosophy* instead of science? One major introductory CS text claims that CS is neither a science nor the study of computers [Abelson et al., 1996, "Preface to the First Edition"]. Rather, it is what they call 'procedural epistemology', which they define (italics added) as:

> *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."

And, of course, epistemology is, after all, a branch of philosophy.

"How to" is certainly important, and interestingly distinct from "what is". But this distinction is hard to make precise. Many imperative statements can be converted to declarative ones; e.g., each "*p* :- *q*" rule of a Prolog program can be interpreted either procedurally ("to achieve *p*, execute *q*") or declaratively ("*p* if *q*").

Or consider Euclid's *Elements*; it was originally written in "how to" form: To construct an equilateral triangle (using only compass and straightedge), follow this algorithm [Toussaint, 1993].[10] (Compare: To compute the value of this function (*using only the operations of a Turing-machine*, or: *using only recursive functions*), follow this algorithm.)[11] But today it is expressed in "what is" form: The triangle that is constructed (using only compass and straightedge) by following that algorithm is equilateral: "When Hilbert gave a modern axiomatization of geometry at the beginning of the present century, he asserted the bald existence of the line. Euclid, however, also asserted that it can be constructed" [Goodman, 1987, §4]. Note that the declarative version of a geometry theorem can be considered to be a formal proof of the correctness of the procedural version. This is closely related to the notion of program verification.

But even if procedural language can be intertranslated with declarative language, the two are distinct. And surely CS is concerned with procedures! There is a related issue in philosophy concerning the difference between knowing *that* something is the case (knowing that a declarative proposition is true) and knowing *how* to do something (knowing a procedure for doing it). This, in turn, may be related to Knuth's view of programming as teaching a computer (perhaps a form of knowing-that), to be contrasted with the view of a machine-learning algorithm

[10] http://www.perseus.tufts.edu/hopper/text?doc=Perseus:text:1999.01.0086:book=1:type=Prop:number=1

[11] For further discussion of "to accomplish goal *G*, do procedure *P*", see [Rapaport, 2015].

that allows a computer to learn on its own by being trained. The former can easily gain declarative "knowledge" of what it is doing so that it can be programmed to explain what it is doing; the latter not so easily.

13.4 CS Is Computational Thinking

A popular way to describe CS is as a "way of thinking", that "algorithmic thinking" (about anything!) is what makes CS unique:

> CS is the new "new math," and people are beginning to realize that CS, like math, is unique in the sense that many other disciplines will have to adopt that way of thinking. It offers a sort of conceptual framework for other disciplines, and that's fairly new. . . . Any student interested in science and technology needs to learn to think algorithmically. That's the next big thing.
> —Bernard Chazelle, interviewed in [Anthes, 2006]

Jeannette Wing's notion of "computational thinking" [Wing, 2006, echoing [Papert, 1980]] is thinking in such a way that a problem's solution "can effectively be carried out by an information-processing agent" [Wing, 2010] (see also [Guzdial, 2011]). Here, it is important not to limit such "agents" to computers, but to include humans! It may offer compromises on several controversies: It avoids the procedural-declarative controversy, by including both concepts, as well as others. Her definition of CS as "the study of computation—what can be computed and how to compute it" is nice, too, because the first conjunct clearly includes the theory of computation and complexity theory ('can' can include "can in principle" as well as "can efficiently"), and the second conjunct can be interpreted to include both software programming as well as hardware engineering. 'Study' is nice, too: It avoids the science-engineering controversy.

"[T]o think computationally [is] to use abstraction, modularity, hierarchy, and so forth in understanding and solving problems" [Scott and Bundy, 2015, p. 37]—indeed, computational thinking involves all of those methods cited in §13.2 for handling complexity! Five years before Perlis defined CS as the science of *computers*, he emphasized what is now called computational *thinking*:

> [T]he purpose of . . . [a] first course in programming . . . is not to teach people how to program a specific computer, nor is it to teach some new languages. *The purpose of a course in programming is to teach people how to construct and analyze processes*. . . .
>
> A course in programming . . . , if it is taught properly, is concerned with abstraction: the abstraction of constructing, analyzing, and describing processes. . . .

> This, to me, is the whole importance of a course in programming. It is a simulation. The point is not to teach the students how to use ALGOL, or how to program the 704. These are of little direct value. The point is to make the students construct complex processes out of simpler ones (and this is always present in programming) in the hope that the basic concepts and abilities will rub off. A properly designed programming course will develop these abilities better than any other course. [Perlis, 1962, pp. 209–210, my italics]

Here is another characterization of CS, one that also characterizes computational thinking:

> Computer science is in significant measure all about analyzing problems, breaking them down into manageable parts, finding solutions, and integrating the results. The skills needed for this kind of thinking apply to more than computer programming. They offer a kind of disciplined mind-set that is applicable to a broad range of design and implementation problems. These skills are helpful in engineering, scientific research, business, and even politics![12] Even if a student does not go on to a career in computer science or a related subject, these skills are likely to prove useful in any endeavor in which analytical thinking is valuable. [Cerf, 2016, p. 7]

But Denning finds fault with the notion of "computational thinking", primarily on the grounds that it is too narrow:

> Computation is present in nature even when scientists are not observing it or thinking about it. Computation is more fundamental than computational thinking. For this reason alone, computational thinking seems like an inadequate characterization of computer science. [Denning, 2009, p. 30]

Note that, by 'computation', Denning means Turing-machine computation. For his arguments about why it is "present in nature", see the discussion in §7, above.

(For more on computational thinking, see the homepage for the Center for Computational Thinking, http://www.cs.cmu.edu/~CompThink/.)

[12] And even the humanities [Ruff, 2016]—WJR footnote.

13.5 CS Is AI

> [Computer science] is the science of how machines can be made to carry out *intellectual processes*. [McCarthy, 1963, p. 1, my italics]

> The goal of computer science is to endow these information processing devices with as much *intelligent behavior* as possible.
> [Hartmanis, 1993, p. 5, my italics] (cf. [Hartmanis, 1995, p. 10])

> Computational Intelligence *is* the manifest destiny of computer science, the goal, the destination, the final frontier.[13]
> [Feigenbaum, 2003, p. 39]

These aren't exactly definitions of CS, but they could be turned into ones: CS is the study of (choose one): (a) how to get computers to do what humans can do; (b) how to make computers (at least) as "intelligent" as humans; (c) how to understand (human) cognition computationally.

The history of computers supports this: It is a history that began with how to get machines to do *some* human thinking (certain mathematical calculations, in particular), then more and more. Indeed, the Turing machine, as a model of computation, was motivated by how *humans* compute: [Turing, 1936, §9] analyzes how humans compute, and then designs a computer program that does the same thing. But the branch of CS that analyzes how humans perform a task and then designs computer programs to do the same thing is AI. So, *the Turing machine was the first AI program!*

But, as I will suggest in §14.1, defining CS as AI is probably best understood as a special case of its fundamental task: determining what tasks are computable.

13.6 CS Is Magic

> Any sufficiently advanced technology is indistinguishable from magic.
> —Arthur C. Clarke, http://en.wikipedia.org/wiki/Clarke's three laws

Could it be that the advanced technology of CS is not only indistinguishable from magic, but really *is* magic? Not magic as in tricks, but magic as in Merlin or Harry Potter?

> Computer science is very empowering. It's kind of like knowing magic: you learn the right stuff and how to say it, and out comes an answer that solves a real problem. That's so cool.
> —Euakarn (Som) Liengtiraphan, quoted in [Hauser, 2017, p. 16]

[13] "Understanding the activities of an animal or human mind in algorithmic terms seems to be about the greatest challenge offered to computer science by nature" [Wiedermann, 1999, p. 1].

Brooks makes an even stronger claim than Clarke:

> The programmer, like the poet, works only slightly removed from pure thought-stuff. He [sic] builds castles in the air, creating by the exertion of the imagination .... Yet the program construct, unlike the poet's words [or the magician's spells?], is real in the sense that it moves and works, producing visible outputs separate from the construct itself. ... ***The magic of myth and legend has come true in our time.*** *One types the correct incantation on a keyboard, and a display screen comes to life, showing things that never were nor could be.*
> [Brooks, 1975, pp. 7–8, my emphases].

Of course, the main difference between "the magic of myth and legend" and how computers work is that the former lacks (or at least fails to specify) any causal connection between incantation and result, whereas computation is quite clear about the connection: Recall our emphasis on algorithms (and see the discussion in §14.1.1.2, below).

What is "magic"? One anthropologist defines magic as the human "use of symbols to control forces in nature" [Stevens, 1996, p. 721]. Clearly, programming involves exactly that kind of use of symbols [Rapaport, 2015].

How is magic supposed to work? The anthropologist James G. Frazer [Frazer, 1915] "had suggested that primitive people imagine magical impulses traveling over distance through 'a kind of invisible ether.' " [Stevens, 1996, p. 722]. That sounds like a description of electromagnetic waves: Think of electrical currents running from a keyboard to a CPU, information traveling across the Internet, or text messaging.

According to another anthropologist, Bronisłow Malinowsky,

> The magical act involves three components: the formula, the rite, and the condition of the performer. The rite consists of three essential features: the dramatic expression of emotion through gesture and physical attitude, the use of objects and substances that are imbued with power by spoken words, and, most important, the words themselves. [Stevens, 1996, p. 722, citing Malinowski]

A "wizard", *gesturing* with a "wand", *performs* a "spell" consisting of a *formula* expressed in the *words* of an arcane language; the spell has real-world effects, *imbuing objects with power*.

Abstracting away from "the dramatic expression of emotion", use of a computer involves gestures, perhaps not with a wand, but with a mouse, a trackpad, or a touchscreen: The computer itself can be thought of as "imbued with power" when we issue, perhaps not a spell, but a command, either spoken or typed. And the words (of a programming language, or even English; think: Siri) used by the programmer or user are surely important, so the "rite" criterion is satisfied. Computer programs can be thought of as formulas, and only those programmers who know how to use appropriate programming languages, or those users who have accounts on a computer, might be considered to be in the right "condition".

> Since classical times sophisticated forms of mysticism that relied on written notations and formulas have been called hermetic magic, after Hermes Trismegistus, Greek variant of the Egyptian Thoth, regarded as the god or principle of wisdom and the originator of writing.
> [Stevens, 1996, p. 721]

Clearly, computer programming relies on written notations and formulas. (Of course, it's not clear whether it's "mysticism"!)

> [A symbol] can take on the qualities of the thing it represents, and it can take the place of its referent; indeed, as is evident in religion and magic, *the symbol can become the thing it represents*, and in so doing, the symbol takes on the power of its referent.
> [Stevens, 1996, p. 724, my italics]

We see this happening in computers when we treat icons on a desktop (such icons are symbols) or the screen output of a WYSIWYG word processor (such as a page of a Microsoft Word document) as if they were the very things they represent. Perhaps more significantly, we see this in the case of those computer simulations in which the simulation of something really *is* that (kind of) thing: In online banking, the computational simulation of transferring funds between accounts *is* the transferring of funds; (simulated) signatures on online Word or PDF documents carry legal weight; in AI, computationally simulated cognition (arguably) *is* cognition. (See [Rapaport, 2012, §8] for further discussion.) And an NRC report (cited by [Samuelson et al., 1994, p. 2324, notes 44 & 46; 2325, note 47]) talks about user interfaces as "illusions":

> Unlike physical objects, the virtual objects created in software are not constrained to obey the laws of physics. . . . In the desktop metaphor, for example, the electronic version of file folders can expand, contract, or reorganize their contents on demand, quite unlike their physical counterparts. [Samuelson et al., 1994, p. 2334]

Isn't that magic?

[Newell, 1980, p. 156] says some things about the nature of physical symbol systems (i.e., computers) that have “magical” overtones. The symbols of such a system “stand for some entity”, i.e.:

> An entity X designates an entity Y relative to a process P, if, when P takes X as input, its behavior depends on Y.

Here, I take it that what Newell means is that P’s behavior *really* depends on Y *instead of on* X, even though X (not Y) is P’s input. But that seems to be the essence of magic; it is “action at a distance: The process behaves as if inputs, remote from those it in fact has, effect it” [Newell, 1980, §4.1]. Process P behaves as it does because of a symbolic “spell” cast at a distance from P itself.

So, perhaps computers are not just metaphorically magic (as Arthur C. Clarke might have said); they *are* magic!

14 So, What Is Computer Science?

Our exploration of the various answers suggests that there is no simple, one-sentence answer to our question. Any attempt at one is no better than the celebrated descriptions of an elephant by the blind men: Many, if not most or all, such attempts wind up describing the entire subject, but focusing on only one aspect of it. Recall Newell, Perlis, & Simon’s and Knuth’s distinct but logically equivalent definitions.

CS is the scientific study of a family of topics surrounding both abstract (or theoretical) and concrete (or practical computing)—a “portmanteau” discipline [Carroll, 1871].

Charles Darwin said that “all true classification . . . [is] genealogical” [Darwin, 1872, Ch. 14, §“Classification”, p. 437]. CS’s genealogy involves two historical traditions: (1) the study of algorithms and the foundations of mathematics (from ancient Babylonian mathematics [Knuth, 1972], through Euclid’s geometry, to inquiries into the nature of logic, leading ultimately to the Turing machine) and (2) the attempts to design and construct a calculating machine (from the Antikythera Mechanism of ancient Greece; through Pascal’s and Leibniz’s calculators and Babbage’s machines; to the ENIAC, iPhone, and beyond). So, modern CS is the result of a marriage between (or merger of) the engineering problem of building better and better automatic calculating devices with the mathematical (hence, scientific) problem of understanding the nature of algorithmic computation. And that implies that modern CS, to the extent that it is a single discipline, has *both* engineering *and* science in its DNA. Hence its portmanteau nature.

The topics studied in contemporary CS roughly align along a spectrum ranging from the mathematical theory of computing, at one end, to the engineering of physical computers, at the other, as we saw in §3.2. Newell, Perlis, & Simon were looking at this spectrum from one end; Knuth was looking at it from the other end. The topics share a family resemblance (and perhaps nothing more than that, except for their underlying DNA), not only to each other, but also to other disciplines (including mathematics, electrical engineering, information theory, communication, etc.), and they overlap with issues discussed in the cognitive sciences, philosophy (including ethics), sociology, education, the arts, and business.

14.1 Five Central Questions of CS

In this section, I want to suggest that there are five central questions of CS. The single most central question is:

1. A. **What *can* be computed?**

But to answer that, we also need to ask:

1. B. ***How* can it be computed?**

Several other questions follow logically from that central one:

2. **What can be computed *efficiently*, and how?**

3. **What can be computed *practically,* and how?**

4. **What can be computed *physically,* and how?**

5. **What *should* be computed, and how?**

Let’s consider each of these in a bit more detail.

14.1.1 Computability

14.1.1.1 ***What** Is Computable?* "What can be computed?" (or: "What is computable?") is the central question, because all other questions presuppose it. The fundamental task of any computer scientist—whether at the purely mathematical or theoretical end of the spectrum, or at the purely practical or engineering end—is to determine whether there is a computational solution to a given problem, and, if so, how to implement it. But those implementation questions are covered by the rest of the questions on our list, and only make sense after the first question has been answered. (Alternatively, they facilitate answering that first question; in any case, they serve the goal of answering it.)

Question 1 includes the question of computability vs. non-computability. It is the question that Church, Turing, Gödel, and others were originally concerned with—*Which mathematical functions are computable?*—and whose answer has been given as the Church-Turing Computability Thesis: A *function* is computable if and only if it is computable by a Turing machine (or any formalism logically equivalent to a Turing machine, such as Church's lambda calculus or Gödel's general recursive functions). It is important to note that not all functions are computable. If they were, then computability would not be an interesting notion. (A standard example of a *non*-computable function is the Halting Problem.)

Various branches of CS are concerned with identifying which problems can be expressed by computable functions. So, a corollary of the Computability Thesis is that a *task* is computable if and only if it can be expressed as a computable function.

Here are some examples:

- Is cognition computable? The central question of AI is whether the functions that describe cognitive processes are computable. (This is one reason why I prefer to call AI "computational cognition" [Rapaport, 1995], [Rapaport, 2003].) Given the advances that have been made in AI to date, it seems clear that at least some aspects of cognition are computable, so a slightly more precise question is: How much of cognition is computable? [Rapaport, 2012, §2, pp. 34–35].
- Consider Shannon's 1950 paper on chess: The principal question is: Can we mathematically analyze chess? In particular, can we *computationally* analyze it (suggesting that computational analysis is a branch or kind of mathematical analysis)—i.e., can we analyze it procedurally? I.e., can we play chess rationally?
- Is the weather computable? See [Hayes, 2007].
- Is fingerprint identification computable? See [Srihari, 2010].
- Is final-exam-scheduling computable? Faculty members in my department recently debated whether it was possible to write a computer program that would schedule final exams with no time conflicts and in rooms that were of the proper size for the class. Some thought that this was a trivial

> problem; others thought that there was no such algorithm (on the (perhaps dubious!) grounds that no one in the university administration had ever been able to produce such a schedule); in fact, this problem is NP-complete (http://www.cs.toronto.edu/~bor/373s13/L14.pdf)

This aspect of question 1 is close to Forsythe's famous one:

> The question "What can be automated?" is one of the most inspiring philosophical and practical questions of contemporary civilization.
> [Forsythe, 1968, p. 1025]

Although similar in intent, Forsythe's question can be understood in a slightly different way: Presumably, a process can be automated—i.e., done automatically, by a machine, without human intervention—if it can be expressed as an algorithm. That is, computable implies automatable. But automatable does not imply computable: Witness the invention of the direct dialing system in telephony, which automated the task of the human operator. Yes, direct dialing is computable, but it wasn't a computer that did this automation.[14]

14.1.1.2 ***How** Is It Computable?* The "how" question is also important: CS cannot be satisfied with a mere existence statement to the effect that a problem is computable; it also requires a constructive answer in the form of an algorithm that explicitly shows *how* it is computable.

In a Calvin and Hobbes cartoon,[15] Calvin discovers that if you input one thing (bread) into a toaster, a different thing (toast) is output. Hobbes wonders what happened to the input. It didn't disappear, of course, nor did it "magically" turn into the output:

> Everything going on in the software [of a computer] has to be physically supported by something going on in the hardware. Otherwise the computer couldn't do what it does from the software perspective—**it doesn't work by magic.** But usually we don't have to know how the hardware works—only the engineer and the repairman do. We can act as though the computer just carries out the software instructions, period. **For all we care, as long as it works, it might as well *be* magic.**
> [Jackendoff, 2012, p. 99, original italics, my boldface]

Rather, the toaster *did something* to the bread (heated it). That intervening process is the analogue of an algorithm for the bread-to-toast function. Finding "intervening processes" requires algorithmic thinking, and results in algorithms that specify the transformational relations between input and output. (Where behaviorism focused only on inputs and outputs, cognitive psychology focused on the intervening algorithms [Miller et al., 1960].)

[14] "Strowger Switch", https://en.wikipedia.org/wiki/Strowger switch

[15] http://www.gocomics.com/calvinandhobbes/2016/03/09, originally published 12 March 1986.

So, just as, for any x, there can be a philosophy of x, so we can ask, given some x, whether there is a computational theory of x. Finding a computational solution to a problem requires "computational thinking", i.e., algorithmic (or procedural) thinking (see §13.4, above).

Computational thinking includes what I call the four Great Insights of CS (http://www.cse.buffalo.edu/~rapaport/computation.html):

1. The *representational* insight:
   Only 2 nouns are needed to represent information
   ('0', '1').

2. The *processing* insight:
   Only 3 verbs are needed to process information
   (*move*(left or right), *print*(0 or 1), *halt*)

3. The *structural* insight:
   Only 3 grammar rules are needed to combine actions
   (sequence, selection, repetition)

4. The "closure" insight:
   Nothing else is needed.
   (This is the import of the Church-Turing Computability Thesis.)[16]

Computational thinking involves both synthesis and analysis:

Synthesis: Given a problem P,

1. express P as a mathematical function F_P
   (or a collection of interacting functions;
   i.e., give an input-output specification of P);
2. try to find an algorithm A_{F_P} for computing F_P
   (i.e., for transforming the input to the output;
   then try to find an efficient and practical version of A_{F_P});
3. implement A_{F_P} on a physical computer.

Note the similarity of synthetic computational thinking to David Marr's analysis of information processing [Marr, 1982] (see discussion in [Rapaport, 2015]).

[16] The exact number of nouns, verbs, or grammar rules depends on the formalism. E.g., some presentations add 'read' or 'erase' as verbs, or use recursion as the single rule of grammar, etc. The point is that there is a very minimal set and that nothing else is needed. Of course, more nouns, verbs, or grammar rules allow for greater ease of expression.

Analysis:

Given a real-world process P
(physical, biological, psychological, social, economic, etc.),
try to find a computational process A_P that models (describes, simulates, explains, etc.) P.

Note that, once found, A_P can be re-implemented; this is why computers can (be said to) think! [Rapaport, 2000]

14.1.2 Efficient Computability

Question 2 is the question studied by the branch of computer science known as computational complexity theory. Given an algorithm, one question is how much time it will take to be executed and how much space (memory) it will need. A more general question is this: Given the set of computable functions, which of them can be computed in, so to speak, less time than the age of the universe or less space than the size of the universe. The principal distinction is whether a function is in the class called P (in which case, it is "efficiently" computable) or in the class *NP* (in which case it is not efficiently *computable* but it *is* efficiently "verifiable"):[17]

> Even children can multiply two primes, but the reverse operation—splitting a large number into two primes—taxes even the most powerful computers. The numbers used in asymmetric encryption are typically hundreds of digits long. Finding the prime factors of such a large number is like trying to unmix the colors in a can of paint, ... "Mixing paint is trivial. Separating paint isn't." [Folger, 2016, p. 52]

Almost all practical algorithms are in *P*. By contrast, one important algorithm that is in *NP* is the Boolean Satisfiability Problem: Given a molecular proposition of propositional logic with n atomic propositions, under what assignment of truth-values to those atomic propositions is the molecular proposition true (or "satisfied")? Whether $P = NP$ is one of the major open questions in mathematics and CS; most computer scientists both hope and believe that $P = NP$ [Fortnow, 2013].

[17] Technically, *P* is the class of functions computable in "*P*olynomial time", and *NP* is the class of functions computable in "*N*on-deterministic *P*olynomial time".

14.1.3 Practical Computability

Question 3 is considered both by complexity theorists as well as by more practically-oriented software engineers. Given a computable function in P (or, for that matter, in *NP*) what are some *practically* efficient methods of actually computing it? E.g., under certain circumstances, some sorting algorithms are more efficient in a practical sense (e.g., faster) than others. Even a computable function that is in *N*P might be practically computable in special cases. And some functions might only be practically computable "indirectly" via a "heuristic": A *heuristic for problem p* can be defined as an *algorithm* for some problem *p'*, where the solution to *p'* is "good enough" as a solution to *p* [Rapaport, 1998, p. 406]. Being "good enough" is, of course, a subjective notion; [Oommen and Rueda, 2005, p. 1] call the "good enough" solution "a *sub-optimal* solution that, hopefully, is arbitrarily close to the *optimal*." The idea is related to Simon's notion of bounded rationality: We might not be able to solve a problem *p* because of limitations in space, time, or knowledge, but we might be able to solve a different problem *p'* algorithmically within the required spatio-temporal-epistemic limits. And if the *algorithmic* solution to *p'* gets us closer to a solution to *p*, then it is a *heuristic* solution to *p*. But it is still an algorithm. (For more on heuristics, see [Romanycia and Pelletier, 1985], [Chow, 2015].) A classic case of this is the Traveling Salesperson Problem, an *NP*-problem for which software like Google Maps solves special cases for us every day (even if their solutions are only "satisficing" ones [Simon, 1996a]).

14.1.4 Physical Computability

But since the only (or the best) way to decide whether a computable function really does what it claims to do is to execute it on a computer, computers become an integral part of CS. Question 4 brings in both empirical (hence scientific) and engineering considerations. Even a practically efficient algorithm for computing some function might run up against physical limitations. Here is one example: Even if, eventually, computational linguists devise practically efficient algorithms for natural-language "competence" (understanding and generation; [Shapiro, 1989], [Shapiro and Rapaport, 1991]), it remains the case that humans have a finite life span, so the infinite capabilities of natural-language competence are not really required (a Turing machine isn't needed; a push-down automaton might suffice). This is also the question that issues in the design and construction of real computers ("computer engineering") are concerned with. And it is where investigations into alternative physical implementations of computing (quantum, optical, DNA, etc.) come in.

14.1.5 Ethical Computability

Bruce Arden, elaborating Forsythe's question, said that "the basic question [is] . . . what can *and should* be automated" [Arden, 1980, p. 29, my italics]. Question 5 brings in ethical considerations [Tedre, 2015, pp. 167–168]. Actually, the question is slightly ambiguous. It could simply refer to questions of practical efficiency: Given a sorting problem, which sorting algorithm *should* be used; i.e., which one is the "best" or "most practical" or "most efficient" in the actual circumstances? But this sense of 'should' does not really differentiate this question from question 3.

It is the *ethical* interpretation that makes this question interesting: Suppose that there is a practical and efficient algorithm for making certain decisions (e.g., as in the case of autonomous vehicles). There is still the question of whether we *should* use those algorithms to actually make decisions for us. Or let us suppose that the goal of AI—a computational theory of cognition—is practically and efficiently computable by physically plausible computers. One can and should still raise the question whether such "artificial intelligences" *should* be created, and whether we (their creators) have any ethical or moral obligations towards them, and vice versa! [Delvaux, 2016] And there is the question of implicit biases that might be (intentionally or unintentionally) built into some machine-learning algorithms.

14.2 Wing's Five Questions

It may prove useful to compare my five questions with [Wing, 2008]'s "Five Deep Questions in Computing":

1. $P = NP$?

2. What is computable?

3. What is intelligence?

4. What is information?

5. (How) can we build complex systems simply?

All but the last, it seems to me, concern scientific (abstract, mathematical) issues: If we consider Wing's second question to be the same as our central one, then her first question can be rephrased as our "What is efficiently computable?", and her third can be rephrased as "How much of (human) cognition is computable?" (a special case of our central question). Her fourth question can then be seen as asking an ontological question about the nature of what it is that is computed (an aspect of our central question): numbers (0s and 1s)? symbols ('0's and '1's)? information in some sense (and, if so, in which sense)?

Wing's last question is ambiguous between two readings of 'build': On a software reading, it can be viewed in an abstract (scientific, mathematical) way as asking a question about the structural nature of software (the issues concerning the proper use of the "goto" statement [Dijkstra, 1968] and structural programming would fall under this category). As such, it concerns the grammar rules; it is then an aspect of our central question. But it can also be viewed on a hardware reading as asking an engineering question: How should we—literally—build computers?

Interpreted in this way, Wing's five questions can be boiled down to two:

- What is computation such that only some things can be computed?
  (And what can be computed (efficiently), and how?)
- (How) can we build physical devices to perform these computations?

The first is equivalent to our questions 1–3, the second to our question 4. And, in this case, we see once again the two parts of the discipline: the scientific (or mathematical, or abstract) and the engineering (or concrete).

It is interesting and important to note that none of Wing's questions correspond to the ethical question 5.

15 Conclusion

To sum up, computer science is the (scientific, or STEM) study of:

- what problems can be solved,
- what tasks can be accomplished, and
- what features of the world can be understood ...

... *computationally*, i.e., using a language with only:

- 2 nouns ('0', '1'),
- 3 verbs ('move', 'print', 'halt'),
- 3 grammar rules (sequence, selection, repetition; or just recursion), and
- nothing else,

and then to provide algorithms to show how this can be done:

- efficiently,
- practically,
- physically, and
- ethically.

I said that our survey *suggests* that there is no simple, one-sentence answer to the question: What is computer science? My definition above is hardly a simple sentence.

But our opening quotation—from an interview with a computational musician—comes closer, so I will end where I began:

> The Holy Grail of computer science is *to capture the messy complexity of the natural world* and *express it algorithmically*.
> — Teresa Marrin Nakra, quoted in [Davidson, 2006, p. 66, my italics].

References

Aaronson, S. (2013). Why philosophers should care about computational complexity. In Copeland, B.J., Posy, C.J., and Shagrir, O., editors, *Computability: Turing, Gödel, Church, and Beyond*, pages 261–327. MIT Press, Cambridge, MA.

Abelson, H., Sussman, G.J., and Sussman, J. (1996). *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA.

Anthes, G. (1 May 2006). Computer science looks for a remake. In *Computerworld*, http://www.computerworld.com/s/article/110959/Computer_Science_Looks_for_a_Remake.

Arden, B. W., editor (1980). *What Can Be Automated? The Computer Science and Engineering Research Study (COSERS)*. MIT Press, Cambridge, MA.

Barwise, J. (1989). For whom the bell rings and cursor blinks. *Notices of the American Mathematical Society*, 36(4):386–388.

Böhm, C. and Jacopini, G. (1966). Flow diagrams, Turing machines and languages with only two formation rules. *Communications of the A*CM, 9(5):366–371.

Boorstin, D. (1983). *The Discoverers*. Random House, New York. Ch. 49: ”The Microscope of Nature”.

Brooks, Jr., F.P. (1996). The computer scientist as toolsmith II. *Communications of the ACM*, 39(3):61–68.

Brooks, Jr., Frederick P. (1975). *The Mythical Man-Month*. Addison-Wesley, Reading, MA.

Carroll, L. (1871). *Through the Looking-Glass*. http://www.gutenberg.org/files/12/12-h/12-h.htm.

Castañeda, H.-N. (1966). ‘He’: A study in the logic of self-consciousness. *Ratio*, 8:130–157.

Cerf, V.G. (2016). Computer science in the curriculum. *Communications of the ACM*, 59(3):7.

Ceruzzi, P. (1988). Electronics technology and computer science, 1940–1975: A coevolution. *Annals of the History of Computing*, 10(4):257–275.

Chow, S.J. (2015). Many meanings of 'heuristic'. *British Journal for the Philosophy of Science*, 66:977–1016.

Darwin, C. (1872). *The Origin of Species*. Signet Classics, 1958, New York. [Davidson, 2006]

Davidson, J. (2006). Measure for measure: Exploring the mysteries of conducting.

*The New York*er, pages 60–69.

Davis, R., Samuelson, P., Kapor, M., and Reichman, J. (1996). A new view of intellectual property and software. *Communications of the ACM*, 39(3):21–30.

Delvaux, M. (2016). Draft report with recommendations to the Commission on Civil Law rules on robotics. European Parliament Committee on Legal Affairs http://www.europarl.europa.eu/sides/getDoc.do?pubRef=-//EP//NONSGML%2BCOMPARL%2BPE-582.443%2B01%2BDOC%2BPDF%2BV0//EN.

Denning, P.J. (1985). What is computer science? *American Scientis*t, 73:16–19. [Denning, 2007]

Denning, P.J. (2007). Computing is a natural science. *Communications of the ACM*, 50(7):13–18.

Denning, P.J. (2009). Beyond computational thinking. *Communications of the ACM*, 52(6):28–30.

Denning, P.J., Comer, D.E., Gries, D., Mulder, M.C., Tucker, A., Turner, A.J., and Young, P.R. (1989). Computing as a discipline. *Communications of the ACM*, 32(1):9–23.

Denning, P.J. and Freeman, P.A. (2009). Computing's paradigm. *Communications of the ACM*, 52(12):28–30.

Denning, P.J. and Rosenbloom, P.S. (2009). Computing: The fourth great domain of science. *Communications of the ACM*, 52(9):27–29.

Dijkstra, E.W. (1968). Go To statement considered harmful. *Communications of the ACM*, 11(3):147–148.

Dijkstra, E.W. (1974). Programming as a discipline of mathematical nature. *American Mathematical Monthly*, 81(6):608–612.

Dijkstra, E.W. (1975). EWD 512: Comments at a symposium. In *Selected Writings on Computing: A Personal Perspective*, pages 161–164. Springer-Verlag, New York.

Dijkstra, E.W. (1976). EWD 611: On the fact that the Atlantic Ocean has two sides. In *Selected Writings on Computing: A Personal Perspective*, pages 268–276. Springer- Verlag, New York.

Dretske, F. (1981). *Knowledge and the Flow of Information*. Blackwell, Oxford.

Feigenbaum, E.A. (2003). Some challenges and grand challenges for computational intelligence. *Journal of the ACM*, 50(1):32–40.

Fetzer, J.H. (1988). Program verification: The very idea. *Communications of the ACM*, 31(9):1048–1063.

Folger, T. (2016). The quantum hack. *Scientific American*, 314(2):48–55.

Forsythe, G.E. (1967). A university's educational program in computer science. *Communications of the AC*M, 10(1):3–8.

Forsythe, G.E. (1968). Computer science and education. *Information Processing 68: Proceedings of IFIP Congress 1968*, pages 1025–1039.

Fortnow, L. (2013). *The Golden Ticket: P, NP, and the Search for the Impossible*. Princeton University Press, Princeton, NJ.

Frazer, J.G. (1911–1915). *The Golden Bough: A Study in Magic and Religion, 3rd ed*. Macmillan, London.

Freeman, P.A. (1995). Effective computer science. *ACM Computing Surveys*, 27(1):27–29.

Gal-Ezer, J. and Harel, D. (1998). What (else) should CS educators know? *Communications of the ACM*, 41(9):77–84.

Goodman, N.D. (1987). Intensions, Church's thesis, and the formalization of mathematics. *Notre Dame Journal of Formal Logic*, 28(4):473–489.

Guzdial, M. (2011). A definition of computational thinking from Jeannette Wing. *Computing Education Blog*. https://computinged.wordpress.com/2011/03/22/a-definition-of-computational-thinking-from-jeanette-wing/.

Hartmanis, J. (1993). Some observations about the nature of computer science. In Shyamasundar, R., editor, F*oundations of Software Technology and Theoretical Computer Science*, volume 761 of *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin/Heidelberg.

Hartmanis, J. (1995). On computational complexity and the nature of computer science. *ACM Computing Surveys*, 27(1):7–16. Reprinted from *Communications of the ACM* 37(10) (October 1994): 37–43.

Hartmanis, J. and Lin, H. (1992). What is computer science and engineering? In Hartmanis, J. and Lin, H., editors, *Computing the Future: A Broader Agenda for Computer Science and Engineering*, pages 163–216. National Academy Press, Washington, DC. Ch. 6.

Hauser, S. (2017). Computing and connecting. *Rochester Review*, 79(3):16–17.

Hayes, B. (2007). Calculating the weather. *American Scientist*, 95(3).

Hendler, J., Shadbolt, N., Hall, W., Berners-Lee, T., and Weitzner, D. (2008). Web science: An interdisciplinary approach to understanding the Web. *Communications of the ACM*, 51(7):60–69.

Hofstadter, D.R. (1980). Review of [Sloman, 1978]. *Bulletin of the American Mathematical Society*, 2(2):328–339.

Jackendoff, R. (2012). *A User's Guide to Thought and Meaning*. Oxford University Press, Oxford.

Kant, I. (1781/1787). *Critique of Pure Reason*. St. Martin's Press, New York. Norman Kemp Smith translation published 1929.

Khalil, H. and Levy, L.S. (1978). The academic image of computer science. *ACM SIGCSE Bulletin*, 10(2):31–33.

Knuth, D.E. (1972). Ancient Babylonian algorithms. *Communications of the ACM*, 15(7):671–677.

Knuth, D.E. (1974a). Computer programming as an art. *Communications of the ACM*, 17(12):667–673.

Knuth, D.E. (1974b). Computer science and its relation to mathematics. *American Mathematical Monthly*, 81(4):323–343.

Knuth, D.E. (1985). Algorithmic thinking and mathematical thinking. *American Mathematical Monthly*, 92(3):170–181.

Koen, B.V. (1988). Toward a definition of the engineering method. *European Journal of Engineering Education*, 13(3):307–315. Reprinted from *Engineering Education* (December 1984): 150–155.

Lamport, L. (2012). How to write a 21st century proof. *Journal of Fixed Point Theory and Applications*,11(1):43–63.

http://research.microsoft.com/en-us/um/people/lamport/pubs/proof.pdf.

Lewis-Kraus, G. (2016). The great A.I. awakening. *New York Times Magazine*. http://www.nytimes.com/2016/12/14/magazine/the-great-ai-awakening.html.

Lindell, S. (24 January 2001). Computer science as a liberal art: The convergence of technology and reason. Talk presented at Haverford College. http://www.haverford.edu/cmsc/slindell/Presentations/Computer%20Science%20as%20a%20Liberal%20Art.pdf.

Loui, M.C. (1987). Computer science is an engineering discipline. *Engineering Education*, 78(3):175–178.

Loui, M.C. (1995). Computer science is a new engineering discipline. *ACM Computing Surveys*, 27(1):31–32.

Loui, M.C. (1996). Computational complexity theory. *ACM Computing Surveys*, 28(1):47–49.

Mahoney, M.S. (2011). *Histories of Computing*. Harvard University Press, Cambridge, MA. Edited by Thomas Haigh.

Maida, A.S. and Shapiro, S.C. (1982). Intensional concepts in propositional semantic networks. *Cognitive Science*, 6:291–330.

Marr, D. (1982). *Vision: A Computational Investigation into the Human Representation and Processing of Visual Information*. W.H. Freeman, New York.

McCarthy, J. (1963). A basis for a mathematical theory of computation. In Braffort, P. and Hirshberg, D., editors, *Computer Programming and Formal Systems*. North- Holland. Page references to PDF version at http://www-formal.stanford.edu/jmc/basis.html.

Miller, G.A., Galanter, E., and Pribram, K.H. (1960). *Plans and the Structure of Behavior*. Henry Holt, New York.

Minsky, M. (1979). Computer science and the representation of knowledge. In Dertouzos, L. and Moses, J., editors, *The Computer Age: A Twenty Year View*, pages 392–421. MIT Press, Cambridge, MA.

Moor, J.H. (1979). Are there decisions computers should never make? *Nature and System,* 1:217–229.

Newell, A. (1980). Physical symbol systems. *Cognitive Science*, 4:135–183.

Newell, A. (1985-1986). Response: The models are broken, the models are broken. *University of Pittsburgh Law Review*, 47:1023–1031.

Newell, A., Perlis, A.J., and Simon, H.A. (1967). Computer science. *Science*, 157(3795):1373–1374.

Newell, A. and Simon, H.A. (1976). Computer science as empirical inquiry symbols and search. *Communications of the ACM*, 19(3):113–126.

Oommen, B.J. and Rueda, L.G. (2005). A formal analysis of why heuristic functions work. *Artificial Intelligence*, 164(1-2):1–22.

Papert, S. (1980). *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, New York.

Perlis, A. (1962). The computer in the university. In Greenberger, M., editor, *Management and the Computer of the Future*, pages 181–217. MIT Press, Cambridge, MA.

Piccinini, G. (2015). *Physical Computation: A Mechanistic Account*. Oxford University Press, Oxford.

Rapaport, W.J. (1981). How to make the world fit our language: An essay in Meinongian semantics. *Grazer Philosophische Studien*, 14:1–21.

Rapaport, W.J. (1986). Philosophy, artificial intelligence, and the Chinese-room argument. *Abacus: The Magazine for the Computer Professional*, 3:6–17. Correspondence, *Abacus* 4 (Winter 1987): 6–7; 4 (Spring): 5–7; http://www.cse.buffalo.edu/~rapaport/Papers/abacus.pdf.

Rapaport, W.J. (1988). Syntactic semantics: Foundations of computational natural-language understanding. In Fetzer, J.H., editor, *Aspects of Artificial Intelligence*, pages 81–131. Kluwer Academic Publishers, Dordrecht, The Netherlands.

Rapaport, W.J. (1995). Understanding understanding: Syntactic semantics and computational cognition. In Tomberlin, J.E., editor, *Philosophical Perspectives, Vol. 9: AI, Connectionism, and Philosophical Psychology*, pages 49–88. Ridgeview, Atascadero, CA.

Rapaport, W.J. (1998). How minds can be computational systems. *Journal of Experimental and Theoretical Artificial Intelligence*, 10:403–419.

Rapaport, W.J. (1999). Implementation is semantic interpretation. *The Monist*, 82:109–130.

Rapaport, W.J. (2000). How to pass a Turing test: Syntactic semantics, natural-language understanding, and first-person cognition. *Journal of Logic, Language, and Information*, 9(4):467–490.

Rapaport, W.J. (2003). What did you mean by that? Misunderstanding, negotiation, and syntactic semantics. *Minds and Machines,* 13(3):397–427.

Rapaport, W.J. (2005a). Implemention is semantic interpretation: Further thoughts. *Journal of Experimental and Theoretical Artificial Intelligence*, 17(4):385–417.

Rapaport, W.J. (2005b). Philosophy of computer science: An introductory course. *Teaching Philosophy*, 28(4):319–341.

Rapaport, W.J. (2012). Semiotic systems, computers, and the mind: How cognition could be computing. *International Journal of Signs and Semiotic Systems,* 2(1):32–71. http://www.cse.buffalo.edu/~rapaport/Papers/Semiotic_Systems,_Computers,_and_the_Mind.pdf.

Rapaport, W.J. (2015). On the relation of computing to the world. Forthcoming in Powers, T.M., editor, *Philosophy and Computing: Essays in Epistemology, Philosophy of Mind, Logic, and Ethics*. Springer. Paper based on 2015 IACAP Covey Award talk; preprint at: http://www.cse.buffalo.edu/~rapaport/Papers/covey.pdf.

Rapaport, W.J. (2017). *Philosophy of computer science*. http://www.cse.buffalo.edu/~rapaport/Papers/phics.pdf.

Rapaport, W.J. and Kibby, M.W. (2007). Contextual vocabulary acquisition as computational philosophy and as philosophical computation. *Journal of Experimental and Theoretical Artificial Intelligence*, 19(1):1–17.

Rapaport, W.J. and Kibby, M.W. (2014). Contextual vocabulary acquisition: From algorithm to curriculum. In Palma, A., editor, *Castañ eda and His Guises: Essays on the Work of Hector-Neri Castañeda*, pages 107–150. Walter de Gruyter, Berlin.

Rapaport, W.J., Shapiro, S.C., and Wiebe, J.M. (1997). Quasi-indexicals and knowledge reports. *Cognitive Science*, 21:63–107.

Rescorla, M. (2015). The computational theory of mind. In Zalta, E.N., ed., *The Stanford Encyclopedia of Philosophy*, winter 2015 edition;
http://plato.stanford.edu/archives/win2015/entries/computational-mind/,

Robinson, J. (1994). Logic, computers, Turing, and von Neumann. In Furukawa, K., Michie, D., and Muggleton, S., editors, *Machine Intelligence 13: Machine Intelligence and Inductive Learning*, pages 1–35. Clarendon Press, Oxford.

Romanycia, M.H. and Pelletier, F.J. (1985). What is a heuristic? *Computational Intelligence*, 1(2):47–58.

Ruff, C. (2016). Computer science, meet humanities: In new majors, opposites attract. *Chronicle of Higher Education*, 62(21):A19.

Samuelson, P., Davis, R., Kapor, M.D., and Reichman, J. (1994). A manifesto concerning the legal protection of computer programs. *Columbia Law Review*, 94(8):2308–2431.

Sayre, K.M. (1986). Intentionality and information processing: An alternative model for cognitive science. *Behavioral and Brain Sciences*, 9(1):121–165.

Schagrin, M.L., Rapaport, W.J., and Dipert, R.R. (1985). *Logic: A Computer Approach*. McGraw-Hill, New York.

Scott, J. and Bundy, A. (2015). Creating a new generation of computational thinkers. *Communications of the ACM*, 58(12):37–40.

Searle, J.R. (1980). Minds, brains, and programs. *Behavioral and Brain Sciences*, 3:417–457.

Shannon, C.E. (1948). A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423, 623–656.

Shapiro, S.C. (1989). The Cassie projects: An approach to natural language competence. In Martins, J. and Morgado, E., editors, *EPIA 89: 4th Portugese Conference on Artificial Intelligence Proceedings,* pages 362–380. Springer-Verlag Lecture Notes in Artificial Intelligence 390, Berlin. http://www.cse.buffalo.edu/sneps/epia89.pdf.

Shapiro, S.C. (2001). Computer science: The study of procedures. *Technical report*, Department of Computer Science and Engineering, University at Buffalo, Buffalo, NY.
http://www.cse.buffalo.edu/~shapiro/Papers/whatiscs.pdf.

Shapiro, S.C. and Rapaport, W.J. (1987). SNePS considered as a fully intensional propositional semantic network. In Cercone, N. and McCalla, G., editors, *The Knowledge Frontier: Essays in the Representation of Knowledge*, pages 262–315. Springer- Verlag, New York.

Shapiro, S.C. and Rapaport, W.J. (1991). Models and minds: Knowledge representation for natural-language competence. In Cummins, R. and Pollock, J., editors, *Philosophy and AI: Essays at the Interface*, pages 215–259. MIT Press, Cambridge, MA

Sheraton, M. (1981). The elusive art of writing precise recipes. *New York Times*. http://www.nytimes.com/1981/05/02/style/de-gustibus-the-elusive-art-of-writing-precise-recipes.html.

Simon, H.A. (1977). What computers mean for man and society. *Science*, 195(4283):1186–1191.

Simon, H.A. (1996a). Computational theories of cognition. In O'Donohue, W. and Kitchener, R.F., editors, *The Philosophy of Psychology*, pages 160–172. SAGE Publications, London.

Simon, H.A. (1996b). *The Sciences of the Artificial, Third Edition*. MIT Press, Cambridge, MA.

Sloman, A. (1978). *The Computer Revolution in Philosophy: Philosophy, Science and Models of Mind*. Humanities Press, Atlantic Highlands, NJ.

Smith, B.C. (1985). Limits of correctness in computers. *ACM SIGCAS Computers and Society*, 14–15(1–4):18–26.

Soare, R.I. (2009). Turing oracle machines, online computing, and three displacements in computability theory. *Annals of Pure and Applied Logic*, 160:368–399.

Srihari, S.N. (2010). Beyond C.S.I.: The rise of computational forensics. *IEEE Spectrum*. http://spectrum.ieee.org/computing/software/beyond-csi-the-rise-of-computational-forensics.

Stevens, Jr., Phillip (1996). Magic. In Levinson, D. and Ember, M., editors, *Encyclopedia of Cultural Anthropology*, pages 721–726. Henry Holt, New York.

Tedre, M. (2015). *The Science of Computing: Shaping a Discipline*. CRC Press/Taylor & Francis, Boca Raton, FL.

Toussaint, G. (1993). A new look at Euclid's second proposition. *TheMathematical Intelligencer*, 15(3):12–23.

Turing, A.M. (1936). On computable numbers, with an application to the *Entscheidungsproblem*. *Proceedings of the London Mathematical Society, Ser. 2*, 42:230–265.

Turing, A.M. (1950). Computing machinery and intelligence. *Mind*, 59(236):433–460.

Wiedermann, J. (1999). Simulating the mind: A gauntlet thrown to computer science. *ACM Computing Surveys,* 31(3es):Paper No. 16.

Wiesner, J. (1958). Communication sciences in a university environment. *IBM Journal of Research and Development*, 2(4):268–275.

Wing, J.M. (2006). Computational thinking. *Communications of the ACM*, 49(3):33–35.

Wing, J.M. (2008). Five deep questions in computing. *Communications of the ACM*,51(1):58–60.

Wing, J.M. (2010). Computational thinking: What and why? *The Link (Carnegie- Mellon University)*. https://www.cs.cmu.edu/~CompThink/resources/TheLinkWing.pdf.

Wulf, W. (1995). Are we scientists or engineers? *ACM Computing Surveys*, 27(1):55–57.

Zemanek, H. (1971). Was ist informatik? (What is informatics?). *Elektronische Rechenanlagen (Electronic Computing Systems)*, 13(4):157–171.