Philosophy of 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 14260-2000**

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

June 14, 2015

Figure 1: ©2015, ??

Contents

---

[1]The material in this section is based on lectures by John Case. WHICH IN TURN ARE BASED ON CLARK & COWELL; SO COMPARE MY PRESENTATION WITH THEIRS!!

If you begin with Computer Science, you will end with Philosophy.[2]

---

[2]"Clicking on the first link in the main text of a Wikipedia article, and then repeating the process for subsequent articles, usually eventually gets you to the Philosophy article. As of May 26, 2011, 94.52% of all articles in Wikipedia lead eventually to the article Philosophy." —"Wikipedia:Getting to Philosophy" http://en.wikipedia.org/wiki/Wikipedia:Getting_to_Philosophy

If you begin with "Computer Science", you will end with "Philosophy" (in 12 links).

Preface

To readers who have found this document through Brian Leiter's *Leiter Reports* blog:[3]

Welcome!

This document is a continually-being-revised draft of a textbook on the philosophy of computer science. It is based on a course I created for the Department of Computer Science and Engineering and the Department of Philosophy at the State University of New York at Buffalo.

The syllabus, readings, assignments, and website for the last version of the course are online at:

http://www.cse.buffalo.edu/∼rapaport/584/

The course is described in:

Rapaport, William J. (2005), "Philosophy of Computer Science: An Introductory Course", *Teaching Philosophy* 28(4): 319–341.

A video of my Herbert Simon Keynote Address at NACAP-2006 describing the course is online at:

http://www.hss.rpi.edu/streaming/conferences/cap2006/nacp_8_11_2006_9_1010.asx

The current draft of the book is just that: a draft of a work in progress. Early chapters contain "Notes for the Next Draft" at the end of each chapter. These represent material that I have not yet fully incorporated into the body of the text. There are also one or two unfinished or unwritten chapters.

Please send any comments, suggestions for improvement, etc., to me via email at:

rapaport@buffalo.edu

And thanks for looking! (Changes are listed on next page.)

---

[3]http://leiterreports.typepad.com/blog/2013/10/a-philosophy-of-computer-science-textbook.html

Note: I am taking a brief hiatus from revisions while I am in the middle of moving from the Buffalo (NY) area to the Jamestown (NY) area.

Changes since previous version (dated 13 May 2015):

Major changes to: Ch. 2, and Ch. 16,
Minor changes to: Ch. 13, Ch. 20, and Bibliography

Acknowledgments

To Do

1. Possibly divide into parts, each with chapters, e.g.:
   I. What is computer science, including what is science, what is engineering
   II. Computers and computing, including history & philosophy of computers, algorithms & Turing
   III. CTCT, including procedures, hypercompn, maybe comprog
   IV.
   software v hardare, including maybe comprog, cVpat, maybe implementation, maybe theories
   V. maybe implementation, maybe theories, progver, limits
   VI. ethics and philai

2. Fix URLs, using BASHurl; see LaTeX Companion

3. Either include somewhere, or recommend to instructor to do, a slow reading of (at least part of) one paper.

4. clarify which chapter is a discussion of Church-Turing Thesis, and put that in the title

5. merge chap on progs as theories with chap on relation of progs to real world?

6. add exercises

   - put exercises at ends of approp chaps as per syllabus?

7. update section on reasoning to include sample valid, invalid args

8. add chap on syntactic semantics

9. write 2nd draft based on files

10. write a new closing chapter/article giving *my* phics.

11. add an "other topics" chapter, for things like the philosophy of the web, social interaction, etc.

12. check Further Readings for soundness & completeness w.r.t. online bibs at http://www.cse.buffalo.edu/~rapaport/584/S10/directory.html

13. Include all relevant emails from /web/faculty/rapaport/584/S07/EMAIL and /web/faculty/rapaport/584/S10/EMAIL as well as anything else relevant from my course online files.

Chapter 1

What Is Philosophy of Computer Science?

Many people "know about modern electronics in general and computers in particular. they know about code and the compilation process that turns source code into binary executable code. Computation theory is something different. It is an area of mathematics that overlaps with philosophy." [PolR, 2009]

Figure 1.1: ©1984 by Berkeley Breathed

1.1   Recommended Readings:

1. Strongly Recommended:

   - Scheutz, Matthias (2003), "Computation, Philosophical Issues about", *Encyclopedia of Cognitive Science* (London: Macmillan): 604–610.

     – http://onlinelibrary.wiley.com/doi/10.1002/0470018860.s00209/abstract
     (accessed 2 November 2012; requires login)

     – May also be accessible online at:

       * http://staffweb.worc.ac.uk/DrC/Courses%202006-7/Comp%204070/
         Reading%20Materials/scheutz%20Comp%20Philo%20Issues.pdf
         (accessed 2 November 2012)

       * http://home.dei.polimi.it/schiaffo/TFI/scheutz_2002.pdf
         (accessed 2 November 2012)

2. Recommended:

   - Turner, Raymond; & Eden, Amnon H. (2008), "The Philosophy of Computer Science", in Edward N. Zalta (ed.), *Stanford Encyclopedia of Computer Science*
     http://plato.stanford.edu/entries/computer-science/
     accessed 2 November 2012

1.2 What This Book Is About

In this book, we will look at some of the central issues in the philosophy of computer science. The book is not designed to answer all (or even any) of the philosophical questions that can be raised about the nature of computers and computer science. Rather, it is designed to "bring you up to speed" on a conversation about these issues— to give you some of the background knowledge about them—so that you can read the literature for yourself and perhaps become part of the conversation by contributing your own views.

This book is intended for readers who might know some philosophy but no computer science, as well as for readers who might know some computer science but no philosophy (and there might even be some readers who know little or nothing about both!). So, we will begin by asking what *philosophy* is. (Most of the rest of the book will be concerned with what *computer science* is.) Thus, our first question will be:

1. What is philosophy?

    (a) And, in particular: What is "the philosophy of *X*?" (where *X* = things like: science, psychology, history, etc., and, of course, computer science).

    Then we will ask:

2. What is computer science?

    To answer this, we will need to consider a series of questions, each of which leads to another:

    (a) Is computer science a science? Or is it a branch of engineering?
        i. What is science?
        ii. What is engineering?
    (b) If computer science is a science, what is it a science *of*?
        i. Is it a science of *computers*?
            A. In that case, what is a computer?
        ii. Or is computer science a science of *computation*?
            A. In that case, what is computation?
                - Computations are said to be algorithms (very roughly, procedures for doing something), so what is an algorithm?
                    – Algorithms are said to be procedures, or recipes, so what is a procedure? What is a recipe?
                - What are Church's and Turing's "theses" about computability? (Briefly, these theses are claims that algorithms can be understood as certain mathematical theories.)
                    – What is "hypercomputation"? (Hypercomputation, roughly, is the study of kinds of computation that cannot be understood by those mathematical theories that model algorithms.)

- Algorithms are said to be implemented in computer programs, so what is a computer program?
  - And what is an implementation?
  - What is the relation of a program to that which it models or simulates?
  - What is simulation?
  - Are programs (scientific) theories?
  - What is software?
  - Can computer programs be copyrighted, or patented?
  - Can computer programs be verified?

3. Next, we will turn to some of the issues in the philosophy of artificial intelligence (AI):

   (a) What is AI?

   (b) What is the relation of computation to cognition?

   (c) Can computers think?

   (d) What are the Turing Test and the Chinese Room Argument?
       (Very briefly: The Turing Test is a test proposed by one of the creators of the field of computation to determine whether a computer can think. The Chinese Room Argument is a thought experiment devised by a philosopher, which is designed to show that the Turing Test won't work.)

4. Finally, we will consider two questions that fall under the heading of computer ethics:

   (a) Should we trust decisions made by computers?

   (b) Should we build "intelligent" computers?

Along the way, we will look at how philosophers reason and evaluate logical arguments, and there will be some suggested writing assignments designed to help focus your thinking about these issues.

1.3   What This Book Is Not About

Have I left anything out? Most certainly! I do not wish to claim that the questions raised above and discussed in this book exhaust the philosophy of computer science. Rather, they are a series of questions that arise naturally from our first question: What is computer science?

But there are many other issues. Some are included in a topic sometimes called "philosophy of computing", which might include investigations into the ubiquity of computing (your smartphone is a computer, your car has a computer in it; perhaps someday your refrigerator or toaster or bedroom wall will contain (or even be) a computer): How will our notion of computing change because of this ubiquity? Will

this be a good or a bad thing?). Other topics include the role of the Internet and social media (for instance, Tim Berners-Lee, who created the World Wide Web, has argued that "Web science" should be its own discipline [Berners-Lee et al., 2006], [Lohr, 2006]). And so on.

Other issues more properly fall under the heading of the philosophy of artificial intelligence ("AI"). Confusingly, they are often called "philosophy of computer science". As noted, we will look at some of these in this book, but there are many others that we won't cover, even though the philosophy of AI is a proper subset of the philosophy of computer science.

And there are several books and essays on the philosophy of *information*. As we'll see, computer science is sometimes defined as the study of how to process information, so the philosophy of information is clearly a close cousin of the philosophy of computer science. But I don't think that either is included in the other; they merely have a non-empty intersection. The "Further Sources of Information" section below includes some of these books and essays if this is a topic you wish to explore.

Finally, there are a number of philosophers and computer scientists who have discussed topics related to what I am calling the philosophy of computer science whom we will not deal with at all (such as the philosopher Martin Heidegger and the computer scientist Terry Winograd).[1] An Internet search (for example: "Heidegger `"computer science"`") will help you track down information on these thinkers and others not mentioned in this book.

But I think that our questions above will keep us busy for a while, as well as prepare you for examining some of these other issues. Think of this book as an extended "infomercial" to bring you up to speed on the computer-science–related aspects of a philosophical conversation that has been going on for over 2500 years, to enable you to join in the conversation.[2]

So, let's begin ...

... but, before we do, on the next page you will find some ...

---

[1] One philosopher of computer science calls them the "Dark Side philosophers", because they tend not to be sympathetic to computational views of the world.

[2] In 2006, responding to a talk that I gave on the philosophy of computer science, Selmer Bringsjord (a philosopher and cognitive scientist who has written extensively on the philosophy of computer science) said, "Philosophy of Computer Science... is in its infancy" [Bringsjord, 2006]. This may be true as a discipline so called, but there have been philosophical investigations of computer science and computing since at least [Turing, 1936], and James H. Moor's work goes back to the 1970s (e.g., [Moor, 1978]).

1.4   . . . Further Sources of Information

Each chapter will end with a list of essays, books, or websites that discuss some or all of the topics covered—but not explicitly cited—in the chapter.[3]   They will also include items that extend the discussion to other topics that were not covered.

1. Books and Essays:

   - Allo, Patrick (2010), "Putting Information First: Luciano Floridi and the Philosophy of Information", *Metaphilosophy* 41(3) (April): 248–254.

   - Burkholder, Leslie (ed.) (1992), *Philosophy and the Computer* (Boulder, CO: Westview Press).

   - Bynum, Terrell Ward (2010), "Philosophy in the Information Age", *Metaphilosophy* 41(3) (april): 420–442.

   - Bynum, Terrell Ward; & Moor, James H. (eds.) (2000), *The Digital Phoenix: How Computers Are Changing Philosophy, Revised Edition* (Oxford: Blackwell).

   - Colburn, Timothy (2000), *Philosophy and Computer Science* (Armonk, NY: M.E. Sharpe).

   - Colburn, Timothy (2006), "What Is Philosophy of Computer Science?", extended abstract of paper presented at the European Conference on Computing and Philosophy (ECAP'06, 22–24 June, Trondheim, Norway)
     http://tinyurl.com/Colburn06 (accessed 2 November 2012)

   - Denning, Peter J.; & Bell, Tim (2012), "The Information Paradox", *American Scientist* 100 (November-December): 470–477.

   - Dodig-Crnkovic, Gordana (2006), "What Is Philosophy of Computer Science? Experience from the Swedish National Course", paper presented at the European Conference on Computing and Philosophy (ECAP'06, 22–24 June, Trondheim, Norway)
     http://www.cse.buffalo.edu/~rapaport/584/Dodig-Crnkovic.pdf
     (accessed 2 November 2012)

   - Dunn, J. Michael (2008), "Information in Computer Science", in Pieter Adriaans & Johan van Benthem (eds.), *Philosophy of Information*, Vol. 8 of Dov M. Gabbay, Paul Thagard, & John Woods (general eds.), *Handbook of the Philosophy of Science*: 587–614, short pre-print at
     http://www.illc.uva.nl/HPI/Draft_Information_in_Computer_Science.pdf
     (accessed 31 March 2015).

     – This is a very readable survey of the nature of information and its role in computer science, covering many of the same topics and issues as this textbook does.

   - Eden, Amnon H.; & Turner, Raymond (eds.), Special Issue on the Philosophy of Computer Science, *Minds and Machines* 21(2) (Summer).

     – Table of contents accessed 23 July 2014 from:
       http://link.springer.com/journal/11023/21/2/page/1

   - Floridi, Luciano (1999), *Philosophy and Computing: An Introduction* (London: Routledge).

---

[3]Though some may be cited in other chapters.

- Floridi, Luciano (2002), "What Is the Philosophy of Information?", *Metaphilosophy* 33(1–2) (January): 123–145.

- Floridi, Luciano (ed.) (2003), *The Philosophy of Information*, two special issues of *Minds and Machines* 13(4) & 14(1).

- Floridi, Luciano (2004), "Open Problems in the Philosophy of Information", *Metaphilosophy* 35(4) (July): 554–582.

- Floridi, Luciano (ed.) (2004), *The Blackwell Guide to the Philosophy of Computing and Information* (Malden, MA: Blackwell).

- Floridi, Luciano (2011), *The Philosophy of Information* (Oxford: Oxford University Press).

  – Essay Review:
    Dunn, J. Michael (2013), "A Guide to the Floridi Keys", *Metascience* 22: 93–98.

- Hilbert, Martin; & López, Priscila (2011), "The World's Technological Capacity to Store, Communicate, and Compute Information", *Science* 332 (1 April): 60–65.

- Longo, Giuseppe (ed.) (1999), *Philosophy of Computer Science*, special issue of *The Monist* 82(1).

- Machlup, Fritz; & Mansfield, Una (eds.) (1983), *The Study of Information: Interdisciplinary Messages* (New York: John Wiley & Sons).

- Magnani, Lorenzo (ed.) (2006), *Computing and Philosophy: Proceedings of the Second European Conference, Computing and Philosophy (ECAP2004-Italy)* (Pavia, Italy: Associated International Academic Publishers).

- Moor, James H.; & Bynum, Terrell Ward (eds.) (2002), *Cyberphilosophy: The Intersection of Computing and Philosophy* (Malden, MA: Blackwell).

- Pylyshyn, Zenon W. (1992), "Computers and the Symbolization of Knowledge", in Ralph Morelli, W. Miller Brown, Dina Anselmi, Karl Haberlandt, & Dan Lloyd (eds.), *Minds, Brains & Computers: Perspectives in Cognitive Science and Artificial Intelligence* (Norwood, NJ: Ablex): 82–94.

  – Preprint (1987) accessed 9 April 2014 from:
    http://ruccs.rutgers.edu/ftp/pub/papers/suffolk.pdf

- Rapaport, William J. (2005), "Philosophy of Computer Science: An Introductory Course", *Teaching Philosophy* 28(4): 319–341.

- Scarantino, Andrea; & Picciini, Gualtiero (2010), "Information without Truth", *Metaphilosophy* 41(3) (April): 313–330.

- Scheutz, Matthias (2003), "Computation, Philosophical Issues about", *Encyclopedia of Cognitive Science* (London: Macmillan): 604–610.

- Smith, Brian Cantwell (1996), *On the Origin of Objects* (Cambridge, MA: MIT Press).

  – Koepsell, David R. (1998), Review of Smith 1996, *Philosophical Psychology* 11(3): 389–391.
  – Loui, R.P. (1998), Review of Smith 1996, *Artificial Intelligence* 106: 353–358.

- Smith, Brian Cantwell (2002), "The Foundations of Computing", in Scheutz, Matthias (ed.), *Computationalism: New Directions* (Cambridge, MA: MIT Press): 23–58.

- DeJohn, Jerry; & Dietrich, Eric (2003), "Subvert the Dominant Paradigm! A review of *Computationalism: New Directions*, edited by Matthias Scheutz", *Journal of Experimental & Theoretical Artificial Intelligence* 15(4) (October-December): 375–382, esp. §3.1, pp. 378–379.

- Tedre, Matti (2007), "Know Your Discipline: Teaching the Philosophy of Computer Science", *Journal of Information Technology Education* 6: 105–122.

- Turner, Raymond; & Eden, Amnon H. (eds.) (2007), *Philosophy of Computer Science* special issue of *Minds and Machines* 17(2) (Summer): 129–247.

- Turner, Raymond; & Eden, Amnon H. (2008), "The Philosophy of Computer Science", special issue of *Journal of Applied Logic* 6(4) (December): 459–552.

2. Websites:[4]

- Aaronson, Scott (2011), "Philosophy and Theoretical Computer Science",
  http://stellar.mit.edu/S/course/6/fa11/6.893/
  (accessed 23 December 2013).

- Brey, Philip; & Søraker, Johnny Hartz (2008), "Philosophy of Computing and Information Technology",
  http://www.idt.mdh.se/kurser/comphil/2011/PHILO-INFORM-TECHNO-20081023.pdf
  (accessed 4 January 2013)

- Brown, Curtis (2000, Fall), Seminar: Philosophy and Computers
  http://www.trinity.edu/cbrown/philandcomp/ (accessed 2 November 2012)

- Commission for the History and Philosophy of Computing, http://www.hapoc.org/

- Dai, Yi; Lamberov, Lev; & Ostermann, Klaus (organizers) (2014), "Discussion Seminar: Philosophy of Programming Languages", Department of Mathematics and Computer Science, University of Marburg (Germany)
  https://github.com/plmday/DS-PhPL
  (accessed 13 June 2014)

- Hill, Robin K. (2014), "Teaching the Philosophy of Computer Science" (blog),
  http://teachingphilofcs.blogspot.com/

- Philosophy of Computer Science (ECAP'06)
  http://pcs.essex.ac.uk/ecap06/cfp.html (accessed 5 February 2013)

- Philosophy of Computer Science: Online and Offline Resources
  http://pcs.essex.ac.uk/ (accessed 2 November 2012)

- Philosophy of Computer Science Course (CD5650, 2004)
  http://www.idt.mdh.se/∼gdc/PI_04/PI-network-course.htm
  (accessed 2 November 2012)

- Philosophy of Computing and Informatics Network
  http://www.idt.mdh.se/∼gdc/pi-network.htm (accessed 2 November 2012)

- Price, Colin B. (2007), "Concepts and Philosophy of Computing", University of Worcester (UK)

  - Accessed 21 March 2014 from:
    http://staffweb.worc.ac.uk/DrC/Courses%202006-7/Comp

---

[4]If any of these websites should disappear between the time of my writing and the time of your reading, you may be able to find them either via a search engine such as Google or via the Wayback Machine Internet archive at: http://archive.org/web/web.php

- Tedre, Matti (2007, Winter-Spring), The Philosophy of Computer Science
  (175616), University of Joensuu, Finland
  http://cs.joensuu.fi/∼mmeri/teaching/2006/philcs/ (accessed 2 November 2012)

  – Contains links to detailed lecture notes.
  – See esp.: "Introduction to the Course"
    http://cs.joensuu.fi/∼mmeri/teaching/2006/philcs/files/lecture_notes1.pdf
    (accessed 2 November 2012)

Now let's really begin . . .

1.5   NOTES FOR NEXT DRAFT

- Include refs online at turing:01-WHAT IS PHILofCS

Chapter 2

What Is Philosophy?

"Philosophy is the microscope of thought."
(Victor Hugo, *Les Misérables* [Hugo, 1862, Vol. 5, Book Two, Ch. II, p. 1262]).

"Philosophy is the peculiarly stubborn attempt to think clearly."
(attributed to William James, 1842–1910).

"To the person with the right turn of mind,. . . all thought becomes philosophy" [Schwitzgebel, 2012].

"Philosophy can be any damn thing you want!"
(John Kearns, personal communication, 7 November 2013.)
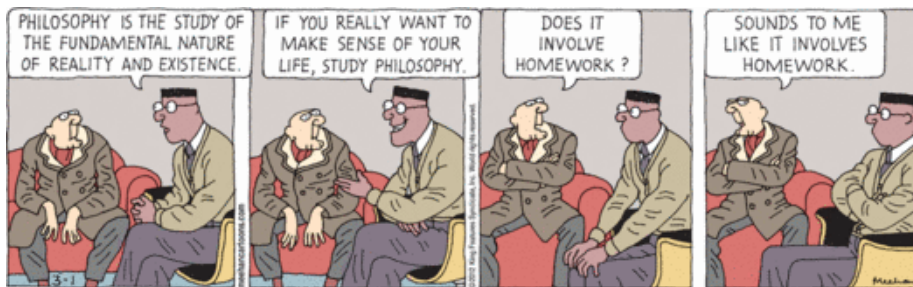
Figure 2.1: ©2012, ??

2.1   Recommended Readings:

1. Very Strongly Recommended:

   - Audi, Robert (1981), "Philosophy: A Brief Guide for Undergraduates" (Newark, DE: American Philosophical Assocation)

     – http://tinyurl.com/Audi81 (accessed 2 November 2012)

2. Strongly Recommended:

   - Plato, *The Apology*

     – Any edition; various versions are online: search for "Plato Apology".

     – This is Plato's explanation of what Socrates thought that philosophy was all about; it is a good introduction to the skeptical, questioning nature of philosophy.

3. Recommended:

   - Colburn, Timothy R. (2000), *Philosophy and Computer Science* (Armonk, NY: M.E. Sharpe):

     (a) Ch. 3: "AI and the History of Philosophy" (pp. 19–40)

     (b) Ch. 4: "AI and the Rise of Contermporary Science and Philosophy" (pp. 41–50)

     – Some of the material may be online at the Google Books (http://books.google.com/) website for this book: http://tinyurl.com/Colburn00

   (accessed 5 February 2013)

2.2 Introduction

"What is philosophy?" is a question that is not a proper part of the philosophy of computer science. But, because many readers may not be familiar with philosophy, I want to begin our exploration with a brief introduction to how I think of philosophy and how I would like non-philosophical readers who are primarily interested in computer science to think of it.

In this chapter, I will give you my definition of 'philosophy'. We will also briefly look at some of the history of philosophy that is relevant to the philosophy of computer science, and we will examine the principal methodology of philosophy: the evaluation of logical arguments. DO I DO THAT HERE, OR IN THE APPENDIX?

2.3 A Definition of 'Philosophy'

The word 'philosophy'[1] has a few different meanings. When it is used informally, in everyday conversation, it can mean an "outlook", as when someone asks you what your 'philosophy of life' is. It can also mean something like "calm", as when we say that someone takes bad news "very philosophically" (that is, very calmly).

But, in this chapter, I want to explicate the *technical* sense of modern, analytic, Western philosophy—that is, the kind of philosophy that has been done since at least Socrates's time (though 'modern philosophy' is itself a technical term that usually refers to the kind of philosophy that has been done since Descartes). It is "analytic" in the sense that it is primarily concerned with the logical analysis of concepts (rather than literary, poetic, or speculative approaches), and it is "Western" in the sense that it has been done by philosophers working primarily in Europe (especially in Great Britain) and North America—though, of course, there are very many philosophers who do analytic philosophy in other countries.

Western philosophy began in ancient Greece. Socrates (470–399 B.C.E.,[2] that is, around 2500 years ago) was opposed to the Sophists, a group of teachers who can be caricaturized as an ancient Greek version of ambulance-chasing lawyers: "purveyors of rhetorical tricks" [McGinn, 2012b]. The Sophists were willing to teach anything (whether it was true or not) to anyone, or to argue anyone's cause (whether their cause was just or not), for a fee. Like the Sophists, Socrates also wanted to teach and argue,

---

[1]Many philosophers have adopted a convention that single quotes are used to form the name of a word or expression. So, when I write this:

<div align="center">'philosophy'</div>

I am not talking about philosophy(!); rather, I am talking about the 10-letter word spelled p-h-i-l-o-s-o-p-h-y. This use of single quotes enables us to distinguish between a *thing* that we are talking about and the *name* or *description* that we use to talk about it. This is like the difference between a *number* (a thing that mathematicians talk about) and a *numeral* (a word or symbol that we use to talk about numbers). This is called the 'use-mention distinction'; see the *Wikipedia* article at http://en.wikipedia.org/wiki/Use-mention_distinction (accessed 5 February 2013).

I will use double quotes when I am directly quoting someone. I will also sometimes use double quotes as "scare quotes", to indicate that I am using an expression in a special or perhaps unusual way (as I just did). And I will use double quotes to indicate the meaning of a word or other expression.

[2]'B.C.E.' is the abbreviation for 'before the common era'; that is, B.C.E. years are the "negative" years before the year 1, which is known as the year 1 C.E. (for "common era").

but only to seek wisdom: truth in any field. In fact, the word 'philosophy' comes from Greek roots meaning "love of [philo] wisdom [sophia]". The reason that Socrates only *sought* wisdom rather than claiming that he *had* it (like the Sophists did) was that he believed that he *didn't* have it: He claimed that he *knew* that he didn't know anything (and that, therefore, he was actually wiser than those who claimed that they *did* know things but who really didn't). As Plato (430–347 B.C.E., Socrates's student) put it in his dialogue *Apology*, Socrates played the role of a "gadfly", constantly questioning (and annoying!) people about the justifications for, and consistency among, their beliefs, in an effort find out the truth for himself from those who considered themselves to be wise (but who really weren't).

Plato defined 'philosopher' (and, by extension, 'philosophy') in Book V of his *Republic* (line 475c):

> The one who feels no distaste in sampling *every study*, and who attacks the task of learning gladly and cannot get enough of it, we shall justly pronounce the lover of wisdom, the philosopher. [Plato, 1961b, p. 714, my emphasis].

Adapting this, I define 'philosophy' as:

the personal search for truth, in any field, by rational means.

This raises several questions:

1. Why only "personal"? (Why not "universal"?)

2. What is "truth"?

3. Why is philosophy only the *search* for truth? Can't we succeed in our search?

4. What does 'any field' mean? Is philosophy really the study of anything and everything?

5. What counts as being "rational"?

Let's look at each of these, beginning with the second.

2.4   What Is Truth?

The study of the nature of truth is one of the "Big Questions" of philosophy (along with things like: What is the meaning of life? What is good? and so on).[3] I cannot hope to do justice to it here, but there are two theories of truth that will prove useful to keep in mind on our journey through the philosophy of computer science: the *correspondence* theory of truth and the *coherence* theory of truth.

---

[3]See Gabriel Segal's response to the question, "What is it that is unique to philosophy that distinguishes it from other disciplines?", online at:
http://www.askphilosophers.org/question/5017 (accessed 13 February 2013).

2.4.1 The Correspondence Theory of Truth

According to the correspondence theory (cf. [David, 2009]), truth is a relation between two things: On the one hand, there are *beliefs* (or sentences, or propositions); on the other hand, there is "reality": A belief (or a sentence, or a proposition) is true if and only if ("iff") it corresponds to reality, that is, iff it "matches" or accurately characterizes or describes reality.

To take a classic example, the three-word English sentence 'Snow is white.' is true iff the stuff in the real world that precipitates in certain winter weather (that is, snow) has the same color as milk (that is, iff it is white). Put somewhat paradoxically (but correctly—see the first footnote in §2.3) 'Snow is white.' is true iff snow is white.

How do we determine whether a sentence (or belief, or proposition) is true? In principle, we would have to compare the parts of the sentence (its words plus its grammatical structure, and maybe even the context in which it is thought, uttered, or written) with parts of reality, to see if they correspond. But how do we access "reality"? How can we do the "pattern matching" between our beliefs and reality?

One answer is by sense perception (perhaps together with our beliefs about what we perceive). But sense perception is notoriously unreliable (think about optical illusions, for instance). And one of the issues in deciding whether our *beliefs* are true is deciding whether our *perceptions* are accurate (that is, whether *they* match reality).

So we seem to be back to square one, which gives rise to the coherence theory.

2.4.2 The Coherence Theory of Truth

According to the coherence theory of truth (cf. [Young, 2008]), a set of propositions (or beliefs, or sentences) is true iff:

1. they are mutually consistent, and

2. they are supported by, or consistent with, all available evidence;

that is, they "cohere" with each other and with all evidence.

Note that observation statements (that is, descriptions of what we observe in the world around us) are among the claims that must be mutually consistent, so this is *not* (necessarily) a "pie-in-the-sky" theory that doesn't have to relate to the way things really are. It just says that we don't have to have independent access to "reality" in order to determine truth.

Which theory is correct? Well, for one thing, there are more than two theories: There are several versions of each kind of theory, and there are other theories of truth that don't fall under either category. Fortunately, the answer to which kind of theory is correct (that is, which kind of theory is, if you will excuse the expression, *true*!) is beyond our present scope! But note that the propositions that a correspondence theory says are true must be mutually consistent (if "reality" is consistent!) and they must be supported by all available evidence; that is, a correspondence theory must "cohere". (Moreover, if you include both propositions and "reality" in one large, highly interconnect network, that network must also "cohere".)

Let's return to the question raised above: How can we decide whether a statement is true? One way that we can determine its truth is *syntactically* (that is, in terms of its

grammatical structure only, not in terms of what it means): by trying to prove it from axioms via rules of inference. (I'll say more about what axioms and rules of inference are in Chs. 14 and 22. For now, just think of proving theorems in geometry or logic.) It is important to keep in mind that, when you prove a statement this way, you are not proving that it is true! You are simply proving that it follows logically from certain other statements, that is, that it "coheres" in a certain way with those statements. Now, if the starting statements—the axioms—are true (note that I said "*if* they are true; I haven't told you how to determine their truth value yet), *and* if the rules of inference "preserve truth", then the statement that you prove by means of them—the "theorem"— will also be true. (Briefly, rules of inference—which tell you how to infer a statement from other statements—are truth-preserving if the inferred statement cannot be false as long as the statements from which it is inferred are true.)

Another way we can determine whether a statement is true is *semantically* (that is, in terms of what it means). This, by the way, is the only way to determine whether an axiom is true, since, by definition, an axiom cannot be inferred from any other statements. (If it could be so inferred, then it would be those other statements that would be the real axioms.)

But to determine the truth of a statement semantically is also to use syntax: We semantically determine the truth value of a complex proposition by syntactic manipulation (truth tables) of its atomic constituents. (We can use truth tables to determine that axioms are true.) NEED TO EXPLAIN WHAT SYN AND SEM ARE, AND HOW THEY ARE RELATED. How do we determine the truth value of an atomic proposition? By seeing if it corresponds to reality. But how do we do that? By comparing the proposition with reality, that is, by seeing if the proposition coheres with reality.

2.5   On Searching for the Truth vs. Finding It?

> Thinking is, or ought to be, a coolness and a calmness.... [Melville, 2002, Ch. 135, p. 419]

How does one go about searching for the truth, for answering questions? As we'll see below, there are basically two, complementary methods: thinking hard and empirical investigation. I have read in many places that philosophy is just thinking really hard about things.[4] Such hard thinking requires "rethinking explicitly what we already believe implicitly" [Baars, 1997, 187]. In other words, it's more than just expressing one's opinion unthinkingly.

Can we find the truth? Not necessarily; that is, we may not be *able* to find it. But I also believe that finding it is not necessary; that is, we may not *have* to find it:

---

[4]Several people have offered this characterization of philosophy. See the quotes at [Popova, 2012] from Marilyn Adams (originally in [Edmonds and Warburton, 2010, xiii]) and, especially, David Papineau: "Philosophy is thinking hard about the most difficult problems that there are. And you might think scientists do that too, but there's a certain kind of question whose difficulty can't be resolved by getting more empirical evidence. It requires an untangling of presuppositions: figuring out that our thinking is being driven by ideas we didn't even realize that we had. And that's what philosophy is" [Edmonds and Warburton, 2010, xx].

Philosophy is the *search* **for truth.**[5] Albert Einstein (quoting G.E. Lessing)[6] said that "the search for truth is more precious than its possession" [Einstein, 1940, 492]. In a similar vein, the mathematician Carl Friedrich Gauss said, "It is not knowledge, but the act of learning, not possession but the act of getting there, which grants the greatest enjoyment."[7]

One reason that this search will never end (which is different from saying that it will not succeed) is that you can always ask "Why?"; that is, you can always continue inquiring. This is

> the way philosophy—and philosophers—are[:] Questions beget questions, and those questions beget another whole generation of questions. It's questions all the way down. [Cathcart and Klein, 2007, 4]

(You can even ask why "Why?" is the most important question [Everett, 2012, 38]!) In fact, as the physicist John Wheeler has pointed out, the more questions you answer, the more questions you can ask: "We live on an island of knowledge surrounded by a sea of ignorance. As our island of knowledge grows, so does the shore of our ignorance."[8] And the US economist and social philosopher Thorstein Veblen said, "The outcome of any serious research can only be to make two questions grow where only one grew before"[Veblen, 1908, 396].

If the philosophical search for truth is a never-ending process, can we ever make any progress in philosophy? Mathematics and science, for example, are disciplines that not only search for the truth, but seem to find it; they seem to make progress in the sense that we know more mathematics and more science now than we did in the past. We have well-confirmed scientific theories, and we have well-established mathematical proofs of theorems. (The extent to which this may or may not be exactly the right way to look at things will be considered in Ch. 4.) But philosophy doesn't seem to be able to empirically confirm its theories or prove any theorems. So, is there any sense of "progress' in philosophy? Or are the problems that philosophers investigate unsolvable?

I think there can be, and is, progress. Solutions to problems are never as neat as they seem to be in mathematics; in fact, they're not even that neat in mathematics! Solutions to problems are always *conditional*; they are based on certain assumptions. In mathematics, those assumptions include axioms, but axioms can be challenged and modified (consider the history of non-Euclidean geometry, which began by challenging and modifying the Euclidean axiom known as the "parallel postulate").[9] So, solutions

---

[5]For more on the importance of search over success, see my website on William Perry's theory of intellectual development [http://www.cse.buffalo.edu/~rapaport/perry-positions.html] and Rapaport 1982.

[6]"The true value of a man [sic] is not determined by his possession, supposed or real, of Truth, but rather by his sincere exertion to get to the Truth. It is not possession of the Truth, but rather the pursuit of Truth by which he extends his powers..." [Lessing, 1778].

[7]From "Letter to Bolyai", 1808; see
http://blog.gaiam.com/quotes/authors/karl-friedrich-gauss/21863, accessed 9 September 2011.

[8]*Scientific American* 267 (1992); see
http://www.todayinsci.com/W/Wheeler_John/WheelerJohn-Quotations.htm, accessed 9 September 2011.

[9]One version of the Parallel Postulate is this: For any line L, and for any point P not on L, there is only one line L' that P is on and that is parallel to L. For some of the history of non-Euclidean geometries, see http://mathworld.wolfram.com/ParallelPostulate.html (accessed 13 February 2013) or http://en.wikipedia.org/wiki/Parallel_postulate (accessed 13 February 2013).

are really parts of larger theories, which include the assumptions that the solution
depends on, as well as other principles that follow from the solution. Progress can be
made in philosophy (as in other disciplines), not only by following out the implications
of your beliefs (a kind of "forward-looking" progress), but also by becoming aware of
the assumptions that underlie your beliefs (a kind of "backward-looking" progress).

Asking "Why?" is part—perhaps the principal part—of philosophy's "general
role of critically evaluating beliefs" [Colburn, 2000, 6] and "refusing to accept
any platitudes or accepted wisdom without examining it" (Donna Dickenson, in
[Popova, 2012]. Critical thinking in general, and philosophy in particular, "look...
for crack[s] in the wall of doctrinaire [beliefs]—some area of surprise, uncertainty, that
might then lead to thought" [Acocella, 2009, 71]. Whenever you have a question, either
because you do not understand something or because you are surprised by it or unsure
of it, you should begin to think carefully about it. And one of the best ways to do this
is to ask "Why?": Why did the author say that? Why does the author believe it? Why
should I believe it? And a related question is this: What are the implications of that?
(What else must be true if that were true? And should I believe those implications?)

Philosophy is a "watchdog" [Colburn, 2000, 6]. This zoological metaphor is related
to Socrates's view of the philosopher as gadfly, investigating the foundations or reasons
for beliefs and for the way things are, always asking "What *is X*?" and "Why?".
Of course, this got him in trouble: His claims to be ignorant were thought (probably
correctly) to be somewhat disingenuous. As a result, he was tried, condemned to death,
and executed. (For the details, read Plato's *Apology*.)

One moral is that philosophy can be dangerous. As Eric Dietrich puts it:

> Thinking about the Big Questions is serious, difficult business. I tell my
> philosophy students: "If you like sweets and easy living and fun times and
> happiness, drop this course now. Philosophers are the hazmat handlers
> of the intellectual world. It is we who stare into the abyss, frequently
> going down into it to great depths. This isn't a job for people who scare
> easily or even have a tendency to get nervous." (Personal communication,
> 5 October 2006.)

But there are advantages to always asking questions and being skeptical: "A
skeptical approach to life leads to advances in all areas of the human condition; while
a willingness to accept that which does not fit into the laws of our world represents a
departure from the search for knowledge" [Dunning, 2007]. Being skeptical doesn't
necessarily mean refraining from having any opinions or beliefs. But it does mean
being willing to question anything and everything that you read or hear (or think!).
(Another way f putting this is this: In philosophy, the jury is always out!—cf.
[Polger, 2011, 11]. But, as we saw above, this does not mean that there can be no
progress in philosophy.) Why would you want to question anything and everything?
So that you can find reasons for (or against) believing what you read or hear (or think).
And why is it important to have these reasons? For one thing, it can make you feel
more confident about your beliefs and the beliefs of others. For another, it can help you
try to convince others about your beliefs—not necessarily to convince them that they
should believe what you believe, but to help them understand why you believe what
you do.

2.6 What Is "Rational"?

> "Philosophy: the ungainly attempt to tackle questions that come naturally
> to children, using methods that come naturally to lawyers" [Hills, 2007].

"The main concern of philosophy is to question and understand very common ideas that all of us use every day without thinking about them" [Nagel, 1987, 5]. This is why, perhaps, the questions that children often ask (especially, "Why?") are often deeply philosophical questions.

But *mere* statements (that is, opinions) *by themselves* are **not** rational. Rather, **arguments**—reasoned or *supported* statements—are capable of being rational. That is, being rational requires *logic*. But there are lots of different (kinds of) logics, so there are lots of different kinds of rationality. And there is another kind of rationality, which depends on logics of various kinds, but goes beyond them in at least one way: empirical, or scientific, rationality.

2.6.1 Logical Rationality

There are (at least) two *basic* kinds of logical rationality: deductive-logical (or absolutely certain) rationality and scientific (or probabilistic) rationality. There is also, I think, a third kind, which I'll call "psychological" or maybe "economic", and which is at the heart of knowledge representation and reasoning in AI.

2.6.1.1 Deductive Logic

"Deductive" logic is the main kind of *logical* rationality. Premises $P_1, \ldots, P_n$ deductively support (or "yield", or "entail", or "imply") a conclusion C iff C *must* be true *if* all of the P_i are true; that is, C is true iff the P_i are "truth preserving".

I will use the symbol '$\vdash_D$' to represent this truth-preserving relation between premises and a conclusion that is deductively supported by them. For example,

$$P, P \rightarrow C \vdash_D C$$

means that the two premises P and $P \rightarrow C$ (read this as: "if P, then C") deductively support (that is, they deductively imply) C.

For example, let P = "Today is Wednesday." and let C = "We are studying philosophy." So the inference becomes: "Today is Wednesday. If today is Wednesday, then we are studying philosophy. Therefore (deductively), we are studying philosophy."

Note that C can be false! It only has to be true *relative to* the premises (that is, true relative to its context).

Also, any or all of the P_i can be false! A deductive argument is said to be "valid" iff it is impossible for all of the premises to be true but the conclusion false. This can be said in a slightly different way: A deductive argument is valid iff, whenever all of its premises are true, its conclusion cannot be false (or: its conclusion must also be true).

A deductive argument is said to be "sound" iff it is valid *and* all of its premises are, in fact, true. So, any or all of the premises of a deductively valid argument can be false, as long as—*if* they *were* true, then the conclusion *would also have to be* true.)

Also, the P_i can be *irrelevant* to C! But that's not a good idea, because it wouldn't be a *convincing* argument. ("Relevance" logics are one way of dealing with this problem; [Anderson and Belnap, 1975, Anderson et al., 1975].)

2.6.1.2   Inductive Logic

"Inductive" logic is one of the three main kinds of *scientific* rationality: The first is *deductive* (as we just saw): Logical rationality is certainly part of science. The third is "abductive" (discussed in the next section).

In inductive logic, $P_1, \ldots, P_n \vdash_I C$ iff C is *probably* true *if* all of the P_i *are* true. For example, suppose that you have an urn containing over a million ping-pong balls, and suppose that you take out one of them at random and observe that it is red. What do you think the chances are that the next ball will also be red? They are probably not very high. But suppose that the second ball that you examine is also red. And the third. ... And the 999,999th. Now how likely do you think it is that the next ball will also be red? The chances are probably very high, so:

$$\text{Red(ball}_1), \ldots, \text{Red(ball}_{999,999}) \vdash_I \text{Red(ball}_{1,000,000}).$$

Unlike deductive inferences, however, inductive ones do not *guarantee* the truth of their conclusion. Although it is not likely, it is quite possible that the millionth ping-pong ball will be, say, the only blue one in the urn.

2.6.1.3   Abductive Logic

"Abductive" logic, sometimes also known as "inference to the best explanation" [Harman, 1965, Lipton, 2004, Campos, 2011],[10] is also scientific: From observation O made at time t_1 and from a theory T that deductively or inductively entails O, one can abductively infer that T must have been the case at earlier time t_0. In other words, T is an explanation of why you have observed O. O:f course, it is not necessarily a good, much less the best, explanation, but the more observations that T explains, the better a theory it is.

In another form of abduction, from observation O_1 made at time t_1, and from observation O_2 made at a later time t_2, one can abductively infer that O_1 might have caused or logically entailed O_2.

Like inductive inferences, abductive ones do not guarantee the truth of their conclusion. Moreover, abductive inferences are deductively invalid! But they are at the heart of the scientific method for developing and confirming theories.

---

[10]For the origin of the term in the writings of the American philosopher Charles Sanders Peirce (who pronounced his name like the word 'purse'), see:
http://www.helsinki.fi/science/commens/terms/abduction.html (accessed 6 February 2013).

2.6.1.4   Non-Monotonic Logic

"Non-monotonic" reasoning is more "psychologically real" than any of the others. It also underlies what the economist and AI researcher Herbert Simon called "satisficing" (or being satisfied with having a reasonable answer to your question rather than an optimal one), for which he won the Nobel Prize in Economics. (We'll return to this topic in Ch. 5, §6.)

In *monotonic* logics (such as deductive logics), once you have proven that a conclusion *C* follows from a premise *P*, then it will always so follow. But in *non-*monotonic logic, you might infer conclusion *C* from premise *P* at time t_0, but, at later time t_1, you might learn that it is not the case that *C*. In that case, you must revise your beliefs.

For example, you might believe that birds fly and that Tweety is a bird, from which you might conclude that Tweety flies. But if you then learn that Tweety is a penguin, you will need to revise your beliefs.

2.6.2   Scientific Rationality

If philosophy is a search for truth by rational means, what is the difference between philosophy and science? After all, science is also a search for truth by rational means!

2.6.3   Is Science Philosophy?

Is the experimental or empirical methodology of science "rational"? It is *not* deductive. But it yields highly likely conclusions, and is often the best we can get.

I would say that science *is* philosophy, as long as experiments and empirical methods are considered to be "rational" and yield truth. Physics and psychology, in fact, used to be branches of philosophy: Isaac Newton's *Principia*—the book that founded modern physics—was subtitled "Mathematical Principles of *Natural Philosophy*" (italics added), not "Mathematical Principles of *Physics*", and psychology split off from philosophy only at the turn of the 20th century. The philosophers Aristotle and Kant wrote physics books. The physicists Einstein and Mach wrote philosophy. And the "philosophy naturalized" movement in contemporary philosophy (championed by the philosopher Willard Van Orman Quine) sees philosophy as being on a continuum with science.[11]

But, if experiments don't count as being rational and only logic counts, then science is *not* philosophy. And science is also not philosophy, if philosophy is considered to be the search for *universal* or *necessary* truths, that is, things that would be true no matter what results science came up with or what fundamental assumptions we made.

---

[11][Weatherson, 2012] considers how academic disciplines such as physics and psychology were once part of philosophy but aren't anymore, and then wonders, if other parts of philosophy became separate academic disciplines, what would be left. His conclusion, possibly though not necessarily his view on what constitutes the core of philosophy, is: ethics, epistemology, and metaphysics. (Or, as [Schwitzgebel, 2012] put it: "The issues at the foundation are always the same: What there really is [metaphysics], how we know about it [epistemology], and what separates the god from the bad [ethics]". Cf. [Flanagan, 2012, B4] on the nature of "comprehensive" philosohphies.

There might be conflicting world views (e.g., creationism vs. evolution, perhaps). Therefore, the best theory is one that is **consistent**, that is **as complete as possible** (that is, that explains as much as possible), and that is **best-supported** by **good** evidence.

You can't *refute* a theory. You can only point out *problems* with it and then offer a *better* theory. Suppose that you infer a prediction *P* from a theory *T* and a hypothesis *H*, and then suppose that the prediction doesn't come true (your experiment fails; that is, the experimental evidence is that *P* is not the case). Then, logically, *either H* is not the case *or T* is not the case (*or* both!). And, since *T* is probably a complex conjunction of claims $A_1, \ldots, A_n$, then, if *T* is not the case, then at least one of the A_i is not the case. In other words, you need not *give up* a theory; you only need to *revise* it. (This is not to say, however, that sometimes you *should* give up a theory. This is what happens in the case of "scientific revolutions", such as (most famously) when Copernicus's theory that the Earth revolves around the Sun (and not vice versa) replaced the Ptolemaic theory, small revisions to which were making it overly complex without significantly improving it. I'll have a bit more to say about this in Ch. 4.)

2.6.4   Could (Should?) Philosophy Be Scientific?

Could philosophy be more scientific (that is, experimental) than it is? Should it be? Colin McGinn takes philosophy to be a science ("a systematically organized body of knowledge"), in particular, what he dubs 'ontical science': "the subject consists of the search for the essences of things by means of a priori methods" [McGinn, 2012b, McGinn, 2012a].

There is a relatively recent movement (with some older antecedents) to have philosophers do scientific (mostly psychological) experiments in order to find out, among other things, what "ordinary" people (for example, people who are not professional philosophers) believe about certain philosophical topics. My feeling is that this is not really philosophy, but rather an (interesting) branch of cognitive science.[12]

But there is another way that philosophy can be part of a scientific worldview. This can be done by philosophy being continuous with science, that is, by being aware of, and making philosophical use of, scientific results. Rather than being a passive, "archair" discipline that merely analyzes what others say and do, philosophy can—and probably should—be a more active discipline, even helping to contribute to science (and other disciplines that it thinks about). (For a useful discussion of this, which is sometimes called "naturalistic philosophy", see [Thagard, 2012]).

(Philosophers can also be more "practical" in the public sphere: "The philosophers have only *interpreted* the world in various ways; the point is to *change* it" [Marx, 1845]. But an opposing point of view considers that "philosophers… are ordained as priests to keep alive the sacred fires in the altar of impartial truth" [Cohen), 1919, 19]! For more on this, see Ch. 5, §6.))

---

[12]For more information on this movement, see [Nahmias et al., 2006, Appiah, 2007, Appiah, 2008, Knobe, 2009, Beebe, 2011, Nichols, 2011]; for an argument against experimental philosophy, see [Deutsch, 2009].

2.6.5   Is It Always Rational to Be Rational?

Is there anything to be said in favor of *not* being rational?

Suppose that you are having trouble deciding between two apparently equal choices. This is similar to a problem from mediaeval philosophy known as "Buridan's Ass" (see [Zupko, 2011]): According to one version, an ass (that is, a donkey) was placed equidistant between two equally tempting bales of hay but died of starvation because it couldn't decide between the two of them. My favorite way out of such a quandary is to imagine throwing a coin and seeing how you feel if it lands heads up: If you would rather that it had landed *tails* up, then you know what you would have preferred, even if you had "rationally" decided that both choices were perfectly equally balanced.

For another consideration, look up Andrew N. Carpenter's response to the question "To what extent do philosophers/does philosophy allow for instinct, or gut feelings?" on the *AskPhilosophers* website.[13]

Finally, an interesting discussion of the role—and limits—of rationality in AI research is [Russell, 1995].

2.7   What Is the Import of "Personal Search"?

My major professor, Hector-Neri Castañeda (see [Rapaport, 2005a]) used to say that philosophy should be done in the first person, for the first person. So, philosophy is whatever *I* am interested in, as long as I study it in a rational manner and aim at truth (or, at least, aim at the best theory). As the computer scientist Richard W. Hamming [Hamming, 1998, 650] warned, "In science and mathematics we do not appeal to authority, but rather *you are responsible for what you believe*."

There is another way in which philosophy must be a personal search for truth. As one introductory book puts it, "the object here is not to give answers...but to introduce you to the problems in a very preliminary way *so that you can worry about them yourself*" [Nagel, 1987, 6–7].

The point is not to hope that someone else will tell you the answers to your questions. That would be nice, of course, but why should you believe them? The point, rather, is for you to figure out answers for yourself.[14]

2.8   What Is the Import of "In Any Field"?

> [He is a *philosopher*, so he's interested in everything.... [Said by the philosopher David Chalmers about the philosopher Andy Clark, as cited in [Cane, 2014].]

Philosophy also studies things that are *not* studied by any *single* discipline; these are sometimes called "the Big Questions": What is truth? What is beauty? What is good (or just, or moral, or right)? What is the meaning of life? What is the nature of

---

[13]http://www.askphilosophers.org/question/2992.

[14]For a somewhat different take on the value of first-person thinking, see [Kornblith, 2013].

mind?[15] Or, as Jim Holt [Holt, 2009] put it: "Broadly speaking, philosophy has three concerns: how the world hangs together, how our beliefs can be justified, and how to live." The first of these is metaphysics, the second is epistemology, and the third is ethics.

The main branches of philosophy are:

1. **Metaphysics (or ontology)**,[16] which tries to answer the question "What is there?"[17] (and also the question "Why is there anything at all?"). Some of the things that "there might be" include: physical objects, properties, relations, individuals, time, God, actions, events, minds, bodies, etc.

   There are major philosophical issues surrounding each of these. Here are just a few examples:

   - Which physical objects "really" exist? Do rocks and people exist? Or are they "merely" collections of molecules? But molecules are constituted by atoms; and atoms by electrons, protons, and neutrons. And, according to the "standard model", the only really elementary particles are quarks, leptons (which include electrons), and guage bosons; so maybe those are the only "really existing" physical objects.[18] Here is a computationally relevant version of this kind of question: Do computer programs that deal with, say, student records model students? Or are they just dealing with 0s and 1s? And, on perhaps a more fanciful level, could a computer program model students so well that the "virtual" students in the program believe that they are real?[19]

   - Do "socially constructed" things like money, universities, governments, etc., really exist (in the same way that people or rocks do)?

   - Do properties really exist? Or are they just collections of similar (physical) objects; in other words, is there a property—"Redness"—over and above the class of individual red things? Sometimes, this is expressed as the problem of whether properties are "intensional" (like Redness) or "extensional" (like the set of individual red things).[20]

   - Are there any important differences between "accidental" properties (such as my property of being a professor of computer science rather than my being a professor of philosophy) and "essential" properties (such as my property of being a human rather than being a laurel tree)?

---

[15]A "Rhymes with Orange" cartoon (31 August 2007), titled "The Big Questions", shows two men on the beach; one says, "Does turning the air conditioning 'down' mean you are turning it 'up'? Does turning it 'up' mean you are turning it 'down'?", and the other replies, "And is there a true nature of 'up' and 'down'?". The caption is: "Philosophy professors, the summer months".

[16]For more information on ontology, see
http://www.cse.buffalo.edu/~rapaport/563S05/ontology.html, accessed 9 September 2011.

[17]For a computational approach to this ontological question, see
http://www.cse.buffalo.edu/~rapaport/663/F06/course-summary.html, accessed 19 September 2011.

[18]For an interesting take on this, see [Unger, 1979b, Unger, 1979a].

[19]If this sounds like the film *The Matrix*, see Ch. 20.

[20]For more information on the intensional-extensional distinction, see
http://www.cse.buffalo.edu/~rapaport/intensional.html, accessed 19 September 2011.

- What is the mind, and how does it relate to the brain? (This is known as the "mind-body" problem.) Are minds and bodies two different kinds of substances? Or are they two different aspects of some one, underlying substance? Or are there no minds at all, but only brains? Or are there no independently existing physical objects, but only ideas in our minds?

- Do "non-existents" (e.g., Santa Claus) exist? We can and do think and talk about them. Therefore, whether or not they "exist" in any sense, they do need to be dealt with.[21]

And so on.

2. **Epistemology** is the study of knowledge and belief: How do we know what there is? How do we know that there is anything? What is knowledge? Is it justified, true belief, as Plato thought, or are there counterexamples to that analysis (such as those of [Gettier, 1963])? Are there other kinds of knowledge, such as knowing how to do something, or know someone by acquaintance or by description? What is belief, and how does it relate to knowledge? Can a computer (or a robot) be said to have beliefs or knowledge? In fact, the branch of AI called "knowledge representation" applies philosophical results in epistemology to issues in AI and computer science in general, and it has contributed many results to philosophy as well (see, e.g., [Buchanan, 2006] AND OTHERS).

3. **Logic** is the study of good reasoning: What is truth? What is rationality? Which arguments are good ones? Can logic be computationally automated?

4. **Ethics** tries to answer "What is good?", "What ought we to do?". We'll look at some ethical issues arising from computer science in Chs. 18 and 20.

5. Ethics is closely related to **social and political philosophy**, which tries to answer "What are societies?", "What are the relations between societies and the individuals who constitute them?", "What is the nature of law?".

6. **Aesthetics** tries to answer "What is beauty?", "What is art?". (Can computer programs, like mathematical theorems or proofs, be "beautiful"?)

7. Philosophy is one of the few disciplines (history is another) in which the history of itself is one of its branches: The **history of philosophy** looks at what famous philosophers of the past believed and tries to reinterpret their views in the light of contemporary thinking.

8. And of central interest for the philosophy of computer science, there are numerous "philosophies of":

---

[21]See [Quine, 1948]. See also [Hirst, 1991], for a survey of the AI approach to this. And see [Maida and Shapiro, 1982], [Rapaport, 1986], [Wiebe and Rapaport, 1986], [Shapiro and Rapaport, 1987], [Shapiro and Rapaport, 1991], [Rapaport et al., 1997] for some papers on a fully intensional AI approach to these issues.

Philosophy of language tries to answer "What is language?", "What is meaning?". It has large overlaps with linguistics and with cognitive science (including AI and computational linguistics).

Philosophy of mathematics tries to answer "What is mathematics?", "What are numbers?", "Why is mathematics so applicable to the real world?". (For a useful survey, see [Pincock, 2011]

Philosophy of mind tries to answer "What is 'the' mind?", "How is the mind related to the brain?".

Philosophy of science tries to answer "What is science?", "What is a *scientific* theory?", "What is a *scientific* explanation?". The philosophy of computer science is part of the philosophy of science. The philosopher Daniel C. Dennett has written that there was a "reform that turned philosophy of science from an armchair fantasy field into a serious partnership with actual science. There came a time when philosophers of science decided that they really had to know a lot of current science from the inside" [Dennett, 2012, 12]. Although you do not need to know a lot about computer science (or philosophy, for that matter) to learn something from the present book, clearly the more you know about each topic, the more you will be able both to understand what others are saying and to contribute to the conversation. (We will look more closely at the philosophy of science in Ch. 4.)

And, for any *X*, there is a **philosophy of** *X*, which is the philosophical investigation of the fundamental assumptions, methods, and goals of *X* (including metaphysical, epistemological, and ethical issues), where *X* could be: mathematics (what is a number? is math about numbers, numerals, sets, structures?), science, physics, biology, psychology, etc., including, of course, AI and computer science. "Philosophy is 99 per cent about critical reflection on anything you care to be interested in" (Richard Bradley, in [Popova, 2012].

X, by the way, could also be. . . philosophy! The philosophy of philosophy, also known as "metaphilosophy", is exemplfied by this very chapter, which is an investigation into what philosophy is and how it can be done. Some people might think that the philosophy of philosophy is the height of "gazing at your navel", but it's really what's involved when you think about thinking, and, after all, isn't AI just *computational* thinking about thinking?

Philosophy, besides being interested in any specific topic, also has an overarching or topic-spanning function: It asks questions that don't fall under the aegis of specific topics and that span multiple topics: "The aim of philosophy, abstractly formulated, is to understand how things in the broadest possible sense of the term hang together in the broadest possible sense of the term" [Sellars, 1962, 35].[22]

---

[22]Sellars goes on to say:

- "the reflective knowing one's way around in the scheme of things. . . is the aim of philosophy"

- "What is characteristic of philosophy os not a special subject-matter, but the aim of knowing one's way around with respect to the subject-matters of all the special disciplines"

- "It is therefore the 'eye on the whole' which distinguishes the philosophical enterprise."

So, for instance, while it is primarily (but not only) mathematicians who are interested in mathematics *per se* and primarily (but not only) scientists who are interested in science *per se*, it is primarily (but not only) philosophers who are interested in how and why mathematics is so useful for science (cf. [Smith, 2010]).

Are there any topics that philosophy doesn't touch on? I'm sure that there are some topics that philosophy *hasn't* touched on. But I'm equally sure that there are no topics that philosophy *couldn't* touch on.[23]

So, let's now begin our philosophical investigation into computer science.

---

[23]See http://www.askphilosophers.org/question/3001, accessed 19 September 2011.

2.9   NOTES FOR NEXT DRAFT

1. In addition to asking the big questions (what is existence? what is knowledge? what is good? what is beauty?), philosophy asks synoptic questions (how do things hang together)

2. "my vision of the philosopher's role has been gradually reshaped, ever more in the direction of providing the conceptual clarifications and underpinnings for theories that are testable, empirical, scientific." Dennett, Daniel (2003?), "Autobiography (Part 2)", *Philosophy Now*, No. 69.

   • This has some of the synoptic character of Sellars's def but also emphasizes the naturalistic side of philosophy.

3. Phil is a meta-discipline, esp. philosophy of X: In X, you think about X; in philosophy of X, you think about thinking about X. The only subjects that might be purely philosophical are metaphysics, epist, and ethics, yet each of those has strong liks to things like physics, philosophy of mind, and political sci, inter alia.

4. Make sure that list of philosophy topics includes philosophy of mind and philosophy of language, philosophy of art/aesthetics.

5. "Why?" is what children ask, which is why philosophy tackles qns that children ask naturally.

6. discuss "why" (philosophy) vs "how" (sci)

7. Perhaps this should merely go under "Further reading", with annotation:
   Floridi, Luciano (2013), "What Is a Philosophical Question?", *Metaphilosophy* 44(3) (April): 195–221.

   Argues that:

   • "Philosophical questions are questions not answerable empirically or mathematically, with observations or calculations. They are *open questions*, that is, questions that remain in principle open to informed, rational, and honest disagreement, even after all the relevant observations and calculations have become available and the answers have been formulated" (pp. 200–201, my emphasis).

     – Such questions don't have right/wrong answers; they're Contextually Relativistic. An example is "whether one should accept Alice's invitation to her party" (p. 209). A Multiplist would say that there's no correct answer, so any answer is OK. A CR says that whether you should accept or not depends on all sorts of supplementary assumptions, and you have to try to make the best argument you can for an answer.

   • And philosophical questions are also "closed under further questioning" (p. 205).

- – 'Closed' in the sense that "the natural numbers are closed under addition" (p. 204), that is, adding two natural numbers always and only yields another natural number, never anything else. So, "if you question . . . [a philosophical question], you obtain one more philosophical question" (p. 204)

- And philosophical questions are also "ultimate questions" (p. 207).

  - – "Given the network of questions that mutually pose or presuppose each other, ultimate questions are those questions whose answers are most influential in terms of a cascade of further questions and answers to other related questions within that network." (p. 207)

- And they "require noetic resources to be answered" (p. 211), where, by 'noetic resources', Floridi means resources that "include. . . [people's] beliefs, what . . . [they]read on the web, their cultural background, their language, religion, and art, their social practices, their memories. . . , their expectations. . . , their emotional intelligence, their past experiences, and so forth. It is the world of mental contents, conceptual frameworks, intellectual creations, intelligent insights, dialectical reasonings." (pp. 210–211).

  - – Since CR questions depend on their background assumptions, this fits.

- And they are "not absolute questions" (p. 215), where 'absolute questions' are "questions that are formulated with no regard for the kind of LoA [Level of Abstraction] at which their possible answers become sensible" (p. 212–213)., where, in turn, an LoA is, roughly, the "units of measurement" or "currency" (p. 213) in which an answer must be given. So, for example, to say that a car costs 5,000 is an "absolute" answer that makes no sense unless its LoA is made clear: $5,000 or 5,000 Euros. According to Floridi, [Turing, 1950] doesn't ask whether machines can think (in an "absolute" sense) but whether machines can think "at the Level of Abstraction represented by the imitation game" (p. 214).

8. I said that philosophy is the search for truth *in any field*. Here's an interesting variation on that theme:

   > Wittgenstein claims that *there are no realms of phenomena whose study is the special business of a philosopher. . .* [Horwich, 2013, my italics][24]

There are two ways to interpret the italicized passage: One way is to read it as consistent with my claim that all fields are open to philosophical investigation. But the way that Horwich (and Wittgenstein) intended more negatively: There is nothing unique that is the object of study for philosophy (and that is not the object of study of any other discipline). Whether or not this is the case (and, admittedly, it is hard to find counterexamples—metaphysics, possible worlds, epistemology—that are not studied by other disciplines: physics, logic,

---

[24]For related discussions, see [Gutting, 2012], [Lynch, 2013].

psychology), surely the methods that philosophers use and their approaches to various problems differ from those of other disciplines.

9. The American philosopher John Dewey said:

> Active, persistent, and careful consideration of any belief or supposed form of knowledge in the light of the grounds that support it and the further conclusion to which it tends constitutes reflective thought. [Dewey, 1993, 9][25]

In other words, it's not enough to merely think something; you must also consider reasons for believing it, and you must also consider the consequences of believing it.

10. On progress in philosophy, or possibly on the relation of philosophy to sci, or (better) on Marx's Fichte quote:

> There is no way of telling upstream how great an impact any specific bit of research will have. … Who would have guessed that the arcane research done by the small set of mathematicians and philosophers working on formal logic a century ago would lead to the development of computing, and ultimately to completely new industries, and to the reconfiguring of work and life across the globe? [O'Neill, 2013, 8]

11. On philosophy as a conversation across the ages:

> My mind does not simply receive impressions. It talks back to the authors, even the wisest of them, a response I'm sure they would warmly welcome. It is not possible, after all, to accept passively everything even the greatest minds have proposed. One naturally has profound respect for Socrates, Plato, Pascal, Augustine, Descartes, Newton, Locke, Voltaire, Paine, and other heroes of the pantheon of Western culture; but each made statements flatly contradicted by views of the others. So I see the literary and philosophical tradition of our culture not so much as a storehouse of facts and ideas but rather as a hopefully endless Great Debate at which one may be not only a privileged listener but even a modest participant. [Allen, 1989, 2], as cited in [Madigan, 2014, 46].

12.
> One of the things about philosophy is that you don't have to give up on any other field. Whatever field there is, there's a corresponding field of philosophy. Philosophy of language, philosophy of politics, philosophy of math. All the things I wanted to know about I coudl still study within a philosophical framework. —Rebecca Newberger Goldstein, cited in [Reese, 2014b]

---

[25] As cited in [Neuberger, 2012, 293].

13. On abduction:

    > Adding a new hypothesis or axiom to a theory for the purpose of explaining already known facts is a process known as "abduction". [Sloman, 2010, slide 56]

14. In addition to the two kinds of "rationality" discussed in §2.6 (logical rationality and scientific rationality, [Heng, 2014] argues that,

    > a third, modern way of testing and establishing scientific truth—in addition to theory and experiment—is via simulations, the use of (often large) computers to mimic nature. It is a synthetic universe in a computer. ... If all of the relevant physical laws are faithfully captured [in the computer program] then one ends up with an emulation—a perfect, *The Matrix*-like replication of the physical world in virtual reality. [Heng, 2014, 174]

    One consideration that this raises is whether this is really a third way, or just a version of logical rationality, perhaps extended to include computation as a kind of "logic". (This might be a good place to discuss, or to forward-reference, Bostrum's theories on virtual realities, as well as Ch. 15.)

15. Does philosophy lead to skepticism? See http://www.askphilosophers.org/questions/5572

16. Is philosophy worth doing? Or can science answer all of our questions? See:

    (a) Linker, Damon (2014), "Why Neil deGrasse Tyson Is a Philistine", *This Week* (6 May), accessed 12 May 2014 from:
    http://theweek.com/article/index/261042/why-neil-degrasse-tyson-is-a-philistine

    (b) Powell, Lewis (2014), "An Open Letter to Neil deGrasse Tyson", *The Horseless Telegraph* (8 May), accessed 12 May 2014 from:
    http://horselesstelegraph.wordpress.com/2014/05/08/an-open-letter-to-neil-degrasse-tyson/

    (c) Pigliucci, Massimo (2014), "Neil deGrasse Tyson and the Value of Philosophy", *Scientia Salon* (12 May), accessed 13 May 2014 from:
    http://scientiasalon.wordpress.com/2014/05/12/neil-degrasse-tyson-and-the-value-of-philosophy/

17. Here's another nice definition of philosophy:

    > [Philosophy] seek[s] to question, clarify and cohere our assumptions and concepts behind things. [Boey, 2014]

    This is consistent with my view of how there can be progress in philosophy (it can be "backwards progress" forged, not by inferring new consequences from assumptions, but by questioning those assumptions themselves [Rapaport, 1982]. Boey also notes that

this is why there exists the potential for there to be the phi of pretty much *anything*—whenever we critically question fundamental assumptions about our beliefs, our activities, and the world around us, we are in the business of doing philosophy.

18. NPR reporter Scott Simon once told future-NPR-reporter Tamara Keith (when she was in high school), "Consider majoring in philosophy. I did. . . . it taught me how to break apart arguments, how to ask the right questions" [Keith, 2014]

19. A very nice, early-and-later-imitated definition and characterization of philosophy (here called 'metaphysics'):

Metaphysics means only **an unusually obstinate attempt to think clearly and consistently.** . . . A geologist's purposes fall short of understanding Time itself. A mechanist need not know how action and reaction are possible at all. A psychologist has enough to do without asking how both he and the mind which he studies are able to take cognizance of the same outer world. But it is obvious that problems irrelevant from one standpoint may be essential fro another. And as soon as one's purpose is the attainment of the maximum of possible insight into the world as a whole, the metaphysical puzzles become the most urgent ones of all. [James, 1892, "Epilogue: Psychology and Philosophy", p. 427; my boldface]

20. Need to distinguish philosophical ontology from AI ontology.

21. Nice description of an issue in metaphysics:

What is reality? Is it a game, a simulation that minds beyhond our own are skillfully creating for study or for fun? If so, we may be just the characters in their game, foolishly thinking that we are autonomous, sentient beings. Is reality something "out there"— something we simply perceive through our senses? Or is it something "in here"—derivative of the inner workings of the brain, a fabrication, so to speak, of how this small mass of some 85 billion neurons integrates stimuli and creates a sense of the real? [Gleiser, 2014, B13]

The first option, that reality is a game, has been studied by [Bostrom, 2003] and suggested in a story by Stanisław Lem that we'll discuss in Ch. 20 (see also [Lightman, 2012]'s novel *Mr g*). The other two options may be a false dichotomy: Another option is that there is a reality "out there", but that what we experience as that reality is actually a construction of our brain.

22.   . . . what is the use of studying philosophy if all that it does for you is to enable you to talk with some plausibility about some abstruse questions of logic, etc., & if it does not improve your thinking about the important questions of everyday life. . . .
         —Ludwig Wittgenstein (1944), quoted in [Malcolm, 2001, 35];

http://wings.buffalo.edu/epc/authors/perloff/witt_intro.html
(accessed 12 November 2014).

23.   'Two years!' said Dantés. 'Do you think I could learn all this in two years?'

'In their application, no; but the principles, yes. Learning does not make one learned: there are those who have knowledge and those who have understanding. The first requires memory, the second philosophy.'

'But can't one learn philosophy?'

'Philosophy cannot be taught. Philosophy is the union of all acquired knowledge and the genius that applies it...' [Dumas, 1844, Ch. 17, pp. 168–169]

24. On epistemology:

Epistemology is concerned with the question of how, since we live, so to speak, inside our heads, we acquire knowledge of what there is outside our heads. [Simon, 1996a, 162]

25. On the mind-body problem:

With a large number of programs in existence capable of many kinds of performances that, in humans, we call thining, and with detailed evidence that the processes some of these programs use parallel closely the observed human processes, we have in hand a clear-cut answer to the mind-body problem: How can matter think and how are brains related to thoughts? [Simon, 1996a, 164]

Simon's answer is that certain "patternings in matter, in combination with processes that can create and operate upon such patterns" [Simon, 1996a, 164] can do the trick. (Comapare the discussion later in §9.4 on computers as "magic paper". Cite also the book *Patterns in Stone*.)

26. [McGinn, 2015] argues that philosophy is a science, just like physics or mathematics. More precisely, he says that it is the logical science of concepts (pp. 87–88). This paper might be considered as required (or whatever) reading for this chapter.

27. Under truth, mention Tarski!

28. Under non-mono logic: Should satisficing be included here, or is it an orthogonal issue?

29. A self-inconsistent theory can be refuted, or can it? After all, one can always change one's logic! If your theory is inconsistent with an observation, you can augment the theory with an account of how the observation is misleading. But perhaps make this clearer in the text.

30. On "first-person" philosophy (§2.7):

> I do not pretend that I can refute these two views; but I can challenge
> them.... [Popper, 1978, §4, p. 148]

This is the heart of philosophy: not (necessarily) coming up with answers, but
challenging assumptions and forcing you to think about alternatives.

My father's favorite admonition was: Never make assumptions. That is, never
assume that something is the case or that someone is going to do something;
rather, try to find out if it is the case, or ask the person. In other words, challenge
all assumptions.

This is one way that progress can be made in philosophy: It may be "backwards-
looking" progress (because instead of looking "forward" to implications of
your assumptions, you look "backwards" to see where those assumptions might
have come from). (Besides these two directions of progress, there can be a
third, which is orthogonal to these two: Considering other issues that might
not underlie ("backwards" or follow from ("forwards") the one that you are
considering, but that are "inspired" or "suggested" by it ("sideways" progress?).

The desire for answers to all questions has been called the "Dualistic" stance
towards knowledge ([Perry, 1970, Perry, 1981],
http://www.cse.buffalo.edu/~rapaport/perry.positions.html).
But the Dualist soon realizes that not all questions have answers that everyone
agrees with, and some questions don't seem to have answers at all (at least,
not yet). Rather than stagnating (my term, not Perry's!) in a middle stance of
"Multiplism" (because not all questions have answers, all opinions—proposed
answers—are equally good), a further stance is that of "Contextual Relativism":
All proposed answers/opinions can (should!) be considered—and evaluated!—
relative to and *in the context of* assumptions/reasons/evidence that can support
them.

Eventually, you "Commit" to one of these answers, and you become responsible
for defending your commitment to "Challenges". But that is (just) more thinking
and analysis—more philosophizing.

Moreover, the commitment that you make is a personal one (one that you are
responsible for). It is in this way that philosophy is done "in the first person for
the first person", as Castañeda said.

And, finally:

> And among the philosophers, there are too many Platos to enumerate.
> All that I can do is try to give you mine. [Goldstein, 2014, 396]

And even more finally?:

> ... my   purpose   is
> to put my own intellectual home in order.... [Putnam, 2001, as cited
> at http://putnamphil.blogspot.com/2015/02/rational-reconstuction-in-
> 1976-when.html]

And:

> "The philosophy of every thinker is the more or less unconscious
> autobiography of its auther," Nietzsche observed.... [Martin, 2015]

31. Add to the list of philosophies of *X*: philosophy of art, philosophy of law,
    philosophy of religion, philosophy of education, philosophy of history.

32. On progress in philosophy:

    > Progress in philosophy consists, at least in part, in constantly bringing
    > to light the covert presumptions that burrow their way deep down
    > into our thinking, too deep down for us to even be aware of them.
    > ... But whatever the source of these presumptions of which we are
    > oblivious, they must be brought to light and subjected to questioning.
    > Such bringing to light is what philosophical progress often consists
    > of.. ... [Goldstein, 2014, 38]

    Goldstein goes on to say that Plato said as much in his myth of the cave, in that
    the cave-dwellers presume/assume that the shadows are reality, but that, to get
    closer to reality (if not actually to reach it), they must challenge that assumption.

33. On argumentation in philosophy:

    > This is why philosophy must be argumentative. It proceeds by way of
    > arguments, and the arguments are argued over. Everything is aired in
    > the bracing dialectic wind stirred by many clashing viewpoints. Only
    > in this way can intuitions that have their source in societal or personal
    > idiosyncrasies be exposed and questioned. [Goldstein, 2014, 39]

    The arguments are argued over, typically, by challenging their assumptions. (It
    is rare that a philosophical argument will be found to be invalid, and, when that
    happens, it is usually a source of disappointment—unless the invalidity is a very
    subtle one or one that reveals a flaw in the very nature of logic itself, which is an
    even rarer, but not unknown, occurrence.)

34. A nice motto for the beginning of this chapter:

    > And what is it, according to Plato, that philosophy is supposed to do?
    > Nothing less than to render violence to our sense of ourselves and our
    > world, our sense of ourselves in the world. [Goldstein, 2014, 40]

    Because it is violent to have one's assumptions challenged.

35. And another one:

    > MCCOY: . . . Or maybe you just don't have the courage of your
    > convictions.
    > PLATO: I would prefer the courage of my questions.
    > [Goldstein, 2014, 338]

36. On the nature of philosophy, consistent with the view of philosophical progress
    as consisting in the challenging of assumptions:

    > [Plato] is. . . a man keenly aware of the way that assumptions and
    > biases slip into our viewpoints and go unnoticed, and he devised a
    > field devoted to trying to expose these assumptions and biases and to
    > do away with any that conflict with commitments we must make in
    > order to render the world and our lives maximally coherent. . . . Above
    > all, my Plato is the philosopher who teaches us that we should never
    > rest assured that our view, no matter how well argued and reasoned,
    > amounts to the final word on any matter. [Goldstein, 2014, 396]

    and

    > Often philosophy, from Plato on, causes me to *see* that a concept is
    > problematic that I had always felt I could just take for granted. –
    > Hilary Putnam (2015),
    > http://putnamphil.blogspot.com/2015/02/rational-reconstuction-in-
    > 1976-when.html
    > (accessed 5 March 2015)

37.   Thinking does not guarantee that you will not make mistakes. But not
        thinking guarantees that you will. [Lamport, 2015, 41]

    This is relevant to the idea of philosophy as thinking hard.

38. On philosophy as dangerous (Socrates):

    > . . . philosophy is difficult because the questions are hard, and the
    > answers are not obvious. We can only arrive at satisfactory answers
    > by thinking as rigourously as we can with the strongest logical and
    > analytical tools at our disposal.
    > . . . I want them [my students] to care more about thinks like truth,
    > clear and rigorous thinking, and distinguishing the truly valuable from
    > the specious.
    > The way to accomplish these goals is not by indoctrination.
    > Indoctrination teaches you what to think; education teaches you how
    > to think. Further, the only way to teach people how to think is to
    > challenge them with new and often unsettling ideas and arguments.
    > . . . Some people fear that raising such quetions and prompting
    > students to think about them is a dangerous thing. They are right.

> As Socrates noted, once you start asking questions and arguing out the answers, you must follow the argument wherever it leads, and it might lead to answers that disturb people or contradict their ideology. [Parsons, 2015]

A propos of this, the whole point of Western philosophy since Socrates has been to get people to think about their beliefs, to question and challenge them. It is not (necessarily) to come up with answers to difficult questions.

2.10   Further Sources of Information

1. Standard Reference Works (in print and online):

   (a) Craig, Edward (ed.) (1998), *Routledge Encyclopedia of Philosophy* (London: Routledge).

   (b) Edwards, Paul (ed.) (1967), *Encyclopedia of Philosophy* (New York: Macmillan).

   (c) Fieser, James; & Dowden, Bradley (eds.), *Internet Encyclopedia of Philosophy*, http://www.iep.utm.edu/ (accessed 6 February 2013)

   (d) Zalta, Edward N. (ed.), *Stanford Encyclopedia of Philosophy* http://plato.stanford.edu/ (accessed 6 February 2013)

2. What Is Philosophy?

   (a) Books and Essays:

       • Cline, Austin (2012), "Introduction to Philosophy: Defining, Studying, Doing Philosophy Is Important" (about.com) http://atheism.about.com/od/aboutphilosoph1/a/philosophyintro.htm accessed 2 November 2012

       • McGinn, Colin (2003), "Finding Philosophy", *Prospect*, Issue 92 (November) http://www.prospectmagazine.co.uk/magazine/findingphilosophy/ accessed 2 November 2012 (requires login)
           – A brief autobiography of how a well-known contemporary philosopher got into the field.

       • Nagel, Thomas (1987), *What Does It All Mean? A Very Short Introduction to Philosophy* (Oxford: Oxford University Press).
           – My favorite introduction to philosophy.

       • Russell, Bertrand (1912), *The Problems of Philosophy*, various editions and publishers.
           – One of my favorite introductions to philosophy.

       • Russell, Bertrand (1946), "Philosophy for Laymen", in his *Unpopular Essays* (London: George Allen & Unwin); http://www.users.drew.edu/~jlenz/br-lay-philosophy.html (accessed 13 January 2014).
           – Not so much about what philosophy is, but about why studying it is important for everyone, not just professional philosophers. (Warning: The online version has many typographical errors!)

       • Sinnott-Armstrong, Walter (2004), "What Is Philosophy?" http://ismailispirit.wordpress.com/apporved-what-is-philosophy/ accessed 2 November 2012

       • Tallis, Raymond (2013), "Philosophy Isn't Dead Yet", *The Guardian* (26 May); online at: http://www.guardian.co.uk/commentisfree/2013/may/27/physics-philosophy-quantum-relativity-einstein (accessed 16 July 2013)

       • Green, Daryn (2014), "A Philosophical Round", *Philosophy Now* 105 (November/December): 22
           – A short poem illustrating the questioning nature of philosophy.

   (b) Websites:

- Dennett, Daniel; & Steglich-Petersen, Asbjørn (2008), "The Philosophical Lexicon"
  http://www.philosophicallexicon.com/ (accessed 2 November 2012)

  – A satirical dictionary of philosophical jargon.

- Florian, Meghan (2013), "Notes from an Employed Philosopher", *Femmonite* (31 January); online at:
  http://www.femmonite.com/2013/01/notes-from-employed-philosopher.html
  (accessed 13 February 2013)

  – Interesting comments about how philosophy can make you "feel alive" and prepare you for the "real world".

- Philosophers on Their Conceptions of Philosophy (Leiter Reports: A Philosophy Blog, June 2010)
  http://leiterreports.typepad.com/blog/2010/06/philosophers-on-their-conception-of-philosophy.html
  (accessed 2 November 2012)

- Philosophical Fun (Department of Philosophy, University of Rochester)
  http://www.rochester.edu/College/PHL/fun.html
  (accessed 2 November 2012)

- Rosenberg, Alex (2011), "What Exactly Is Philosophy of Science—and Why Does It Matter?",
  http://www.routledge.com/philosophy/articles/
  what_exactly_is_philosophy_of_science_and_why_does_it_matter/
  (accessed 23 July 2013)

- Shammas, Michael (2012), "For a Better Society, Teach Philosophy in High Schools", *Huffington Post* (26 December); online at:
  http://www.huffingtonpost.com/mike-shammas/for-a-better-society-teac_b_2356718.html (accessed 13 February 2013)

- Suber, Peter (2003), Guide to Philosophy on the Internet
  http://www.earlham.edu/~peters/philinks.htm
  (accessed 2 November 2012)

- What Is Philosophy? (Department of Philosophy, University of Melbourne)
  http://philosophy.unimelb.edu.au/about/what-is-philosophy.html
  (accessed 2 November 2012)

- What Is Philosophy? (Department of Philosophy, University of Nevada, Reno)
  http://www.unr.edu/philosophy/2-what_is_philosophy.html
  (accessed 2 November 2012)

- Why Philosophy? (Department of Philosophy, SUNY Buffalo)
  http://philosophy.buffalo.edu/undergraduate/
  (accessed 2 November 2012)

- Why Study Philosophy? (Indiana University Department of Philosophy)
  http://www.indiana.edu/~phil/undergraduate/why.shtml
  (accessed 2 November 2012)

- AskPhilosophers.org questions:

  i. What do people mean when they speak of "doing" philosophy?
     http://www.askphilosophers.org/question/2915
     (accessed 2 November 2012)

ii. Why are philosophers so dodgy when asked a question?
http://www.askphilosophers.org/question/2941
(accessed 2 November 2012)

iii. Are there false or illegitimate philosophies, and if so, who's to say which
ones are valid and which are invalid?
http://www.askphilosophers.org/question/2994
(accessed 2 November 2012)

iv. What does it take to be a philosopher?
http://www.askphilosophers.org/question/4609 (accessed 13 February
2013)

3. Can There Be Progress in Philosophy? Can Philosophy Ever Solve Any of Its Problems?

(a) Books and Essays:

- Moody, Todd C. (1986), "Progress in Philosophy", *American Philosophical
Quarterly* 23(1) (January): 35–46.

- Rapaport, William J. (1982), "Unsolvable Problems and Philosophical
Progress", *American Philosophical Quarterly* 19: 289-298.

- Rapaport, William J. (1984), "Can Philosophy Solve Its Own Problems?", *The
[SUNY] News* 13(May/June): F2–F3
http://www.cse.buffalo.edu/~rapaport/Papers/rapaport84b.canphilsolve.pdf
(accessed 6 February 2013)

- Rescher, Nicholas (1985), *The Strife of Systems: An Essay on the Grounds and
Implications of Philosophical Diversity* (Pittsburgh: University of Pittsburgh
Press).

(b) Websites:

- Have philosophers ever produced anything in the way that scientists have?
(AskPhilosophers.org, 28 July 2008)
http://www.askphilosophers.org/question/2249 (accessed 2 November 2012)

- How is "philosophical progress" made, assuming it is made at all?
(AskPhilosophers.org, 2 February 2012)
http://www.askphilosophers.org/question/4523 (accessed 2 November 2012)

4. What Is Metaphysics?

- Quine, Willard van Orman (1948), "On What There Is", *Review of Metaphysics*
2(5): 21–38; reprinted in: Quine, *From a Logical Point of View: 9
Logico-Philosophical Essays, Second Edition, revised* (Cambridge, MA: Harvard
University Press, 1980): 1–19.

- Hirst, Graeme (1991), "Existence Assumptions in Knowledge Representation",
Artificial Intelligence 49: 199–242.

  – A computational perspective on non-existents and other "intensional" items.

5. On Epistemology: What Is Knowledge?

- Gettier, Edmund L. (1963), "Is Justified True Belief Knowledge?", *Analysis* 23:
121–123; reprinted in many anthologies, and online at:
http://www.ditext.com/gettier/gettier.html (accessed 2 November 2012)

6. How to Study and Write Philosophy:

(a) Books:

- Martinich, A.P. (2005), *Philosophical Writing: An Introduction* (Malden, MA: Blackwell).
- Woodhouse, Mark B. (2007), *A Preface to Philosophy, 8th edition* (Wadsworth Publishing).

(b) Websites:

- Fiala, Andrew (2004), Student Survival Guide
  http://www.wadsworth.com/philosophy_d/special_features/popups/survival_guide.html
  (accessed 2 November 2012)
- Miller, David Marshall (2011), "Writing Philosophy Papers"
  http://people.duke.edu/~dmmiller/Classes/Writing%20Philosophy%20Papers.docx
  (accessed 2 November 2012)
- Suber, Peter (2003), Courses, http://www.earlham.edu/ peters/courses.htm
  (accessed 2 November 2012)
    – See especially the links at the bottom of the page under the heading
    "Files and links that pertain to more than one course".
- Do a computer search for: **"**writing philosophy papers**"** (include the quotes in the search box)

The Philosopher within You

There's the legend of the fish who swam around asking every sea creature he'd meet, "Where is this great ocean I keep hearing about?"...

We are very much like that fish.

For consider, it's hard to look at a newborn baby without thinking: what an incredible miracle. But when was the last time you looked at an adult and had the same thought? But why not? Every adult *was* a little baby; if the latter is a miracle then so is the former. But it never occurs to us to think this way for one simple reason: we're so used to seeing people that we stop reflecting on them.

Or you drop something, a spoon, and it falls to the floor. But why? Couldn't it, in theory, have remained floating in air or moved upwards? And *how* exactly does it fall to the floor, by "gravity"? There are no strings connecting the earth to the spoon. How can the earth pull on something froma distance, that it's not even attached to? Why don't we pause every time something drops and say: what an incredible miracle!

The most ordinary things contain a whole lifetime of questions, if only we are reminded to start asking them.

Children already know to ask these questions. Every answer you provide to one of their "Why?" questions just generates the next question. Btu we were all children once. What we need to do now is to let the child still within us—the philosopher within us—re-emerge. ... We need to take a cold wet plungh into the great deep ocean of thought.

It's time to start thinking.

—Andrew Pessin, *The 60-Second Philosopher* (London: Oneworld): 6–7.

Chapter 3

What Is Computer Science?

Thanks to those of you who [gave their own] faculty introductions [to the new graduate students]. For those who [weren't able to attend], I described your work and courses myself, and then explained via the Reductionist Thesis how *it all comes down to strings and Turing machines operating on them.*
— Kenneth Regan, email to University at Buffalo Computer Science & Engineering faculty (27 August 2004); italics added.

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, 66].
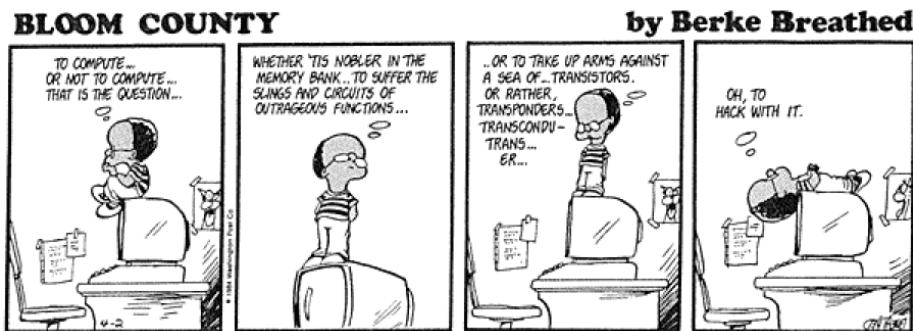
Figure 3.1: ©1984, Washington Post Co.

3.1   Readings:

1. Required:

    (a) Newell, Allen; Perlis, Alan J.; & Simon, Herbert A. (1967), "Computer Science", *Science* 157(3795) (22 September): 1373–1374.

    (b) Knuth, Donald (1974), "Computer Science and Its Relation to Mathematics", *American Mathematical Monthly* 81(4) (April): 323–343.

        - required: §§1–3
        - very strongly recommended: §4
        - strongly recommended: readers who are more mathematically inclined may wish to read the whole essay.

    (c) Newell, Allen; & Simon, Herbert A. (1976), "Computer Science as Empirical Inquiry: Symbols and Search", *Communications of the ACM* 19(3) (March): 113–126.

        i. For the purposes of this chapter, concentrate especially on what Newell & Simon have to say about what computer science is:
            - read the "Introduction" (pp. 113–114)
            - read from "§I. Symbols and Physical Symbol Systems" to the end of the subsection "Physical Symbol Systems" (pp. 114–117)
            - read the "Conclusion" (pp. 125–126)

        ii. For a detailed follow-up, see:
            Newell, Allen (1980), "Physical Symbol Systems", *Cognitive Science* 4: 135–183.
            - Accessed 15 March 2014 from[1] http://tinyurl.com/Newell1980

    (d) Hartmanis, Juris, & Lin, Herbert (1992), "What Is Computer Science and Engineering?", in Juris Hartmanis & Herbert Lin (eds.), *Computing the Future: A Broader Agenda for Computer Science and Engineering* (Washington, DC: National Academy Press), Ch. 6, pp. 163–216.[2]

        - required: "Computer Science & Engineering", pp. 163–168.
        - required: "Abstractions in Computer Systems", pp. 168–174.
        - very strongly recommended: skim the rest.

    (e) Brooks, Frederick P., Jr. (1996), "The Computer Scientist as Toolsmith II", *Communications of the ACM* 39(3) (March): 61–68.

        - required: pp. 61–64.
        - very strongly recommended: skim the rest.

    (f) Shapiro, Stuart C. (2001), "Computer Science: The Study of Procedures",
        http://www.cse.buffalo.edu/∼shapiro/Papers/whatiscs.pdf
        accessed 3 November 2012

---

[1] http://onlinelibrary.wiley.com/store/10.1207/s15516709cog0402_2/asset/s15516709cog0402_2.pdf?v=1&t=hst07yxo&s=e842daedf48a17

[2] This book was the subject of a petition to withdraw it from publication "because we consider it misleading and even harmful as an agenda for future research"
(http://www-formal.stanford.edu/jmc/petition/whysign/whysign.html).
The petition was sponsored by Bob Boyer, John McCarthy, Jack Minker, John Mitchell, and Nils Nilsson. Commentaries on it appeared in [Kling et al., 1993].

2. Very Strongly Recommended (arranged in chronological order):

   (a) Arden, Bruce W. (1980), "COSERS Overview", in Bruce W. Arden (ed.), *What Can Be Automated? The Computer Science and Engineering Research Study (COSERS)* (Cambridge, MA: MIT Press), Ch. 1, pp. 1–31.

   (b) Krantz, Steven G. (1984), Letter to the Editor about the relation of computer science to mathematics, *American Mathematical Monthly* 91(9) (November): 598–600.

   (c) Denning, Peter J. (1985), "What Is Computer Science?", *American Scientist* 73 (January-February): 16–19.

   (d) Abrahams, Paul (1987), "What Is Computer Science?", *Communications of the ACM* 30(6) (June): 472–473.

   (e) Loui, Michael C. (1987), "Computer Science Is an Engineering Discipline", *Engineering Education* 78(3) (December): 175–178.

   (f) Bajcsy, Ruzena K.; Borodin, Allan B.; Liskov, Barbara H.; & Ullman, Jeffrey D. (1992), "Computer Science Statewide Review" (unpublished report)
   http://tinyurl.com/BajcsyEtAl1992[3]
   (personal copy accessed 15 February 2013; login required)

   (g) Abelson, Harold, & Sussman, Gerald Jay, with Sussman, Julie (1996), *Structure and Interpretation of Computer Programs*, "Preface to the First Edition"
   http://tinyurl.com/AbelsonSussman1996Pref[4]
   (accessed 15 February 2013)

   (h) Gal-Ezer, Judith, & Harel, David (1998), "What (Else) Should CS Educators Know?", *Communications of the ACM* 41(9) (September): 77–84.

      - contains a section titled "What Is CS?"
      - contains a "Bibliography for 'What Is CS?' "

   (i) Jacob, Christian (1999), "What Is Computer Science?"
   http://tinyurl.com/Jacob1999[5]
   (accessed 15 February 2013)

   (j) Shagrir, Oron (1999), "What Is Computer Science About?", *The Monist* 82(1): 131–149.

      - Despite its title, this paper is more about what computers are and what computation is; Shagrir assumes that computer science is the science of computers.

   (k) Johnson, George (2001), "All Science Is Computer Science", *The New York Times* (25 March): WK1, WK5 http://tinyurl.com/GJohnson2001[6] (accessed 15 February 2013)

   (l) Foley, Jim (2002), "Computing > Computer Science", *Computing Research News* 14(4) (September): 6; http://archive.cra.org/reports/computing/index.html
   (accessed 3 November 2012)

---

[3]http://www.cse.buffalo.edu/~rapaport/Papers/Papers.by.Others/bajcsyetal92.pdf
[4]http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-7.html#%_chap_Temp_4
[5]http://pages.cpsc.ucalgary.ca/~jacob/Courses/Fall99/CPSC231/03-ComputerScience.pdf
[6]http://www.nytimes.com/2001/03/25/weekinreview/the-world-in-silica-fertilization-all-science-is-computer-science.html

- See also:
  Glass, Robert L. (2003), Response to "Computing > Comptuer Science",
  online at: http://archive.cra.org/reports/computing/glass.html
  (accessed 18 June 2013)

(m) Boston University Department of Computer Science (2003), "What Is Computer
Science?", http://cs-www.bu.edu/AboutCS/WhatIsCS.pdf
(accessed 3 November 2012)

(n) Parlante, Nick (2005), "What Is Computer Science?", *Inroads—The SIGCSE
Bulletin* 37(2) (June): 24–25.

(o) Naur, Peter (2007), "Computing Versus Human Thinking", *Communications of the
ACM* 50(1) (January): 85–94.

3.2 Introduction

In a sense, the fundamental question of this book is: What is computer science? Almost all of the other questions we will be considering flow from this one, so a final answer (if there is one!) will have to await the end of the book.

In this chapter, we will look at several definitions of 'computer science' that have been given by both philosophers and computer scientists. Each definition raises issues that we will examine in more detail later.

3.3 Why Ask What Computer Science Is?

> When our discipline was newborn, there was the usual perplexity as to its proper name. [Brooks, 1996, 61]

Before we even try to answer the question, it's worth asking a preliminary question: Why should we even bother seeking a definition? There are at least two motivations for doing so, academic ones and intellectual (or philosophical) ones.

3.3.1 Academic Motivations

3.3.1.1 Academic Politics

First, there are academic political reasons for asking what computer science is: Where should a "computer science" department (or, as it is in my case, a "computer science and engineering" department) be adminstratively housed? Intellectually, this might not matter (after all, a small school might not even have academic departments, merely teachers of various subjects). But deciding where to place a computer science department can have political repercussions:

> In a purely intellectual sene such jurisdictional questions are sterile and a waste of time. On the other hand, they have great importance within the framework of institutionalized science—e.g., the organization of universities and of the granting arms of foundations and the Federal Government. [Forsythe, 1967b, 455]

Some possible locations include:

- a college or school[7] of arts and sciences

- a college or school of engineering

- a school of informatics.

---

[7]Colleges and universities are usually administratively divided into smaller units, variously known as "schools", "colleges", or—generically—"division", each typically headed by a "dean" and divided into still smaller units, called "departments".

Another possibility is that computer science (or engineering) should not be (merely) a department, but an entire school or college itself, with its own dean (and perhaps with its own departments: a department of theory of computation, a department of artificial intelligence (AI), a department of software engineering, a department of computer architecture, etc.).

There are examples of each, even within a single university system: My own university (State University of New York at Buffalo) has a Department of Computer Science and Engineering within a School of Engineering and Applied Sciences, but, when I joined the university, there were both a Department of Computer Science in a Faculty of Natural Sciences and Mathematics and a separate Department of Electrical and Computer Engineering in a Faculty of Engineering and Applied Sciences. At its sibling institution, State University of New York at Albany, the Department of Computer Science is in the College of Computing and Information. And at my former college, State University of New York College at Fredonia, the Department of Computer and Information Sciences is in the College of Arts and Sciences.

3.3.1.2 Naming the Discipline

In fact, given all of these possibilities, we can even ask the question, "What should this discipline be called?":

- 'Computer *science*'?

- 'Computer *engineering*'?

- '*Computing* science (or engineering)'?

- 'Informatics'? (This name is more common in Europe.)

In this book, but only for convenience, I will call it 'computer science', but, by doing so, I do not mean to exclude the other parts of the discipline, such as engineering or its relationship to information.

(For those who are interested, [Ceruzzi, 1989, esp. pp. 265–270] contains a history of the phrase 'computer science'. And, in a response to a letter that appeared in one of the earliest issues of *Communications of the ACM*, the editor (possibly Alan J. Perlis) listed several, admittedly "facetious", names, including 'turingineering', 'turology', 'applied meta-mathematics', and 'applied epistemology' [of DATA-LINK, 1958, 6]. In 1966, Turing-award-winner Peter Naur suggested 'datalogy' [Naur, 2007, 86].)

3.3.1.3 Academic Pedagogy

Another academic purpose for asking what computer science is concerns pedagogy: What should be taught in an introductory computer science course?

- Should it be a programming course (that is, is computer science the study of programming)?

- Should it be a computer literacy course (that is, is computer science all about how to use computers)?

- Should it be a course in the mathematical theory of computation (that is, is computer science the study of computation)?

- Should it be a course that introduces students to several different branches of computer science, including, perhaps, some of its history?

And so on.

3.3.1.4 Academic Publicity

A related part of the academic purpose concerns publicity for prospective students:

- How should a computer science department advertise itself so as to attract good students?

- How should the discipline of computer science 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? (For more motivations along these lines, see [Denning, 2013, 35].)

- How how should the discipline advertise itself so as to attract more women to the field?

- How should it advertise itself to the public at large, so that ordinary citizens might have a better understanding of what it is?

Many of the definitions of computer science that you can find on various academic websites are designed with one or more of these purposes in mind. Here is an exercise for the reader: Try to find some of these, and see if you can figure out whether they were designed with any of these purposes in mind.

3.3.2 Intellectual or Philosophical Purposes

Perhaps the academic (and especially political) purposes for asking what computer science is are ultimately of little more than practical interest. But there are deep intellectual or philosophical issues that underlie those questions, and this will be the focus of our investigation:

- What is computer science "really"?

- Is it like any other academic discipline? (For instance, is it more like mathematics or more like engineering?)

- Or is it "*sui generis*" (a Latin phrase meaning "own kind")?[8]

To illustrate this difference, consider these comments by two major computer scientists (as cited in [Gal-Ezer and Harel, 1998, 79]): Marvin Minsky, a co-founder of artificial intelligence, once said:

---

[8]Here is a simple analogy: A poodle and a pit bull are both kinds of dogs. But a wolf is not a dog; it is its own kind of animal. Actually, some biologists believe that dogs are actually a kind of wolf, but others believe that dogs are *sui generis*.

> 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]

On the other hand, Juris Hartmanis, a founder of the field of computational complexity
theory, 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, 1] (cf.
> [Hartmanis, 1995a, 10])[9]

3.4   What Does It Mean to Ask What Something Is?

> We will not try to give a few-line definition of computer science since
> no such definition can capture the richness of this new and dynamic
> intellectual process, *nor can this be done very well for any other science.*
> [Hartmanis, 1993, 5]

> It does not make much difference how you divide the Sciences, for they
> are one continuous body, like the ocean. [Leibniz, 1685, 220]

Before we explore our main question further, there is a fundamental principle that
should be kept in mind whenever you ask what something is, or what kind of thing
something is: There are no sharp boundaries in nature; there are only continua. A
"continuum" (plural = 'continua') is like a line with no gaps in it, hence no natural
places to divide it up. The color spectrum is an example: Although we can identify
the colors red, orange, yellow, green, and so on, there are no sharp boundaries where
red ends and orange begins. An apparent counterexample might be biological species:
Dogs are clearly different from cats, and there are no intermediary animals that are not
clearly either dogs or else cats. But both dogs and cats evolved from earlier carnivores
(it is said that both evolved from a common ancestor some 42 million years ago).[10] If
we traveled back in time, we would not be able to say whether one of those ancestors
was a cat or a dog; in fact, the question wouldn't even make sense.

Moreover, although logicians and mathematicians like to define categories in terms
of "necessary and sufficient conditions" for membership, this only works for abstract,
formal categories. For example, we can define a circle of radius r and center c as the
set of all and only those points that are r units distant from c.[11]   As philosophers,

---

[9][Hartmanis, 1995a] covers much of the same ground, and in many of the same words, as
[Hartmanis, 1993], but is more easily accessible, having been published in a major journal that is widely
available online, rather than in a harder-to-find conference proceedings.

[10]See the GreenAnswers.com webpage at:
http://greenanswers.com/q/95599/animals-wildlife/pets/what-common-ancestor-cats-and-dogs
(which cites the *Wikipedia* articles "Feliformia" (http://en.wikipedia.org/wiki/Feliformia) and "Carnivora"
(http://en.wikipedia.org/wiki/Carnivora)) and the gather.com webpage "Miacids" at:
http://www.gather.com/viewArticle.action?articleId=281474977041872   (accessed   15   February   2013).
(According to a *Ziggy* cartoon (http://www.gocomics.com/ziggy/2013/02/19), however, a platypus could be
considered as an intermediary animal between birds and mammals!)

[11]"All" such points is the "sufficient condition"; "only" such points is the "necessary condition": C is a
circle of radius r at center c if and only if $C = \{p : p$ is a point that is r units distant from $c\}$. That is, p is r

psychologists, and cognitive scientists have pointed out, non-abstract, non-formal ("real") categories usually don't have such precise, defining characteristics. The most famous example is the philosopher Ludwig Wittgenstein's unmet challenge to give necessary and sufficient defining characteristics for something's being a game [Wittgenstein, 1958, §66ff]. And the psychologist Eleanor Rosch has pointed out that even precisely definable, mathematical categories can have "blurry" edges: Most people consider 3 to be a "better" example of a prime number than, say, 251 GET CITATION. (For further discussion, see [Rosch and Mervis, 1975], [Rosch, 1978], [Mervis and Rosch, 1981], [Lakoff, 1987], [Hofstadter and Sander, 2013, esp. Ch. 7].)

So, we do not "carve nature at *its* joints" [Plato, 1961a, 265e–266a, my emphasis]; rather, we "carve nature" at "joints" that are usually of our own devising: We impose our own categories on nature. But I would not be a good philosopher if I did not immediately point out that this claim is controversial! After all, isn't the point of science to describe and explain a reality that exists independently of us and of our concepts and categories—that is, independently of the "joints" that we "carve" into nature? (We'll return to this topic in Ch. 4.) And aren't there "natural kinds"? Dogs and cats, after all, do seem to be kinds of things that are there in nature, independently of us.

Is computer science similar to such a "natural kind"? Here, I think the answer is that it pretty clearly is not. There would be no academic discipline of computer science without humans, and there probably wouldn't even be any computers without us, either (though we'll see some reasons to think otherwise, in Ch. 9). So, to the extent that it is we who impose our categories on nature, there may be no good answer to the question "What is computer science?" beyond something like: Computer science is what computer scientists do. (In a similar vein, [Abrahams, 1987, 472] says, "computer science is that which is taught by computer science departments". Perhaps intended more seriously, [Denning, 1999, 1] defines "The discipline of computer science... [as] the body of knowledge and practices used by computing professionals in their work.") But then we can ask: What is it that computer scientists do? Of course, one can beg that last question by saying that computer scientists do computer science! [Hamming, 1968, 4] suggests something like this, but goes on to point out that "there is often no clear, sharp definition of... [a] field".[12]

Here is a continuation of the exercise that I suggested earlier, in §3.3.1.4: Link to the websites for various computer science departments (including your own school's website!) and make a list of the different definitions or characterizations of computer science that you find. Do you agree with them? Do any of them agree with each other? Are any of them so different from others that you wonder if they are really trying to describe the same discipline?

Such advertising blurbs should not, perhaps, be taken too seriously. But there are several published essays that try to define 'computer science', whose authors—all well-

---

units from c only if p is a point on C (that is, if p is r units from c, then p is a point on C); so, being a point that is r units from c is a sufficient condition for being on C. And if p is a point on C, then p is r units from c; so, being a point that is r units from c is a necessary condition for being on C.

[12][Bringsjord, 2006, my emphasis] argued that "any answer...that *includes* some such notion as 'Whatever computer scientists actually do.' is unacceptable." I would replace 'includes' by 'is limited to'; surely, 'includes' has to be included.

respected computer scientists—presumably put a lot of thought into them. They *are* worth taking seriously, which we will do in the next few sections.

3.5   Computer Science Studies Computers

The first such answer that we will look at comes from three computer scientists: Allen Newell, Alan Perlis, and Herbert Simon:[13]

> 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, 1373, my emphasis]

They then reply to several objections to this definition, such as this one: Sciences study *natural* phenomena, but computers aren't natural; they are artifacts. Their reply to this is to deny the premise: Sciences are not limited to studying natural phenomena. There are sciences of artifacts; for example, botanists study hybrid corn.[14] In fact, Simon wrote a book (originally in 1969) called *The Sciences of the Artificial* [Simon, 1996b], and [Knuth, 2001, 167] called it "an *unnatural* science [because] [c]omputer science deals with artificial things, not bound by the constraints of nature." (For a commentary on this, see the *Abstruse Goose* cartoon in Figure 3.3 at the end of the chapter.)

The objector might respond that the fact that Simon had to write an entire book to argue that there could be sciences of artifacts shows that the premise—that science only studies natural phenomena—is not *obviously* false. Moreover, botanists study mostly *natural* plants; hybrid corn is not only not studied by all botanists, it is certainly not the only thing that botanists study (that is, botany is not defined as the science of hybrid corn). Are there any natural phenomena that computer scientists study? (We will see a positive answer to this question in §3.10.)

But let's not be unfair. There certainly are sciences that study artifacts in addition to natural phenomena: Ornithologists study both birds (which are natural) and their nests (which are artifacts); apiologists study both bees (natural) and their hives (artifacts). On the other hand, one might argue (a) that beehives and birds' nests are not *human*-made phenomena, and that (b) 'artifact' should be used to refer, not to any *manufactured* thing (as opposed to things that are grown), but only to things that are manufactured by *humans*, that is, to things that are not "found in nature", so to speak. The obvious objection to this claim is that it unreasonably singles out humans as being apart from nature.[15]

Moreover, there once was a science that only studied a particular artifact— microscopes!

---

[13]Newell was also a cognitive scientist, and Simon (in addition to being a cognitive scientist) was also an economist—in fact, a winner of the Nobel prize in economics.

[14]Curiously, they say that it is *zoologists* who study hybrid corn!

[15]For more on the nature of artifacts in general, see [Dipert, 1993], [Hilpinen, 2011]; on artifacts in computer science, see [Mizoguchi and Kitamura, 2009].

> ...Marcello Malpighi (1628–1694), was a great scientist whose work had
> no dogmatic unity.[16] He was one of the first of a new breed of explorers
> who defined their mission neither by the doctrine of their master nor
> by the subject that they studied. They were no longer 'Aristotelians'
> or 'Galenists.' Their eponym, their mechanical godparent, was some
> device that extended their senses and widened their vistas. What gave
> his researches coherence was a new instrument. Malpighi was to be a
> 'microscopist,' and his science was 'microscopy'.... His scientific career
> was held together not by what he was trying to confirm or to prove,
> but by the vehicle which carried him on his voyages of observation"
> [Boorstin, 1983, 376].

In a similar fashion, surely computers are "device[s] that [have] extended [our] senses
and widened [our] vistas", and the science of computer scientists is, well, *computer*
science. (One of the two principal professional associations is the Association for
Computer Machinery.) What "holds" computer scientists "together... [is] the vehicle
which carrie[s] on [their] voyages of observation".

But this is not necessarily a positive analogy. Arguably, microscopy no longer
exists as an independent science. Now, if you search for 'Department of Microscopy'
on the World Wide Web, you will, indeed, find that there are some universities and
museums that have one. But, if you look closer, you will see that they are really
departments of microbiology. Non-biologists who use microscopes (such as some
geologists or physicists) are not found in departments of microscopy today. What
has happened, apparently, is that the use of this artifact by scientists studying widely
different phenomena was not sufficient to keep them in the same academic discipline.
The academic discipline of microscopy splintered into those who use microscopes to
study biology, those who use it to study geology, and so on, as well as those who
build new kinds of microscopes (who might be found in an engineering or an optics
department). A former dean who oversaw the Department of Computer Science at
my university once predicted that the same thing would happen to our department:
The computer-theory researchers would move (back) into the math department; the
AI researchers would find homes in psychology, linguistics, or philosophy; those who
built new kinds of computers would move (back) into electrical engineering; and so on.
This hasn't happened yet, and I don't forsee it happening in the near future, if at all.
But it is something to consider. ([McBride, 2007] suggests that it is already happening;
[Mander, 2007] disagrees.)

It is, in fact, close to another objection: Computer science is really just a branch
of (choose at least one): electrical engineering, mathematics, psychology, etc. Newell,
Perlis, and Simon reply that, although computer science does intersect each of these,
there is no other, *single* discipline that subsumes all computer-related phenomena.
This, however, begs the question of whether computer science *is* a single discipline.
I used to be in a philosophy department; although certain branches of philosophy
were not my specialty (ethics and history of philosophy, for instance), I was expected
to, and was able to, participate in philosophical discussions on these topics. But

---

[16]I assume that this means that Malpighi did not study any *single*, natural phenomenon, but that he studied
all such phenomena that are visible with a microscope.

my colleagues in computer science often do not, nor are expected to, understand the details of those branches of computer science that are far removed from their own. As a computer scientist specializing in AI, I have far more in common with colleagues in the philosophy, psychology, and linguistics departments than I do with my computer-science colleagues down the hall who specialize in, say, computer networks or computer security. (And this is not just an autobiographical confession on my part; my colleagues in computer networks and computer security would be the first to agree that they have more in common with some of their former colleagues in electrical engineering than they do with me.)

NOTE FOR NEXT DRAFT:

> The applications of computers to a discipline should be considered properly a part of the natural evolution of the discipline. . . . The mass spectrometer has permitted significant advances in chemistry, but there is no 'mass spectrometry science' devoted to the study of this instrument.
> [Loui, 1987, 177]

Similarly, the microscope has permitted significant advances in biology (and many other disciplines) but microscopy is (no longer) a science devoted to the study of that instrument. Presumably, contributions made by AI researchers to philosophy or psychology should not be considered to be the results of AI research?

END NOTE

If computer science does study computers, but "sciences" properly so-called (the "natural sciences") do not study such artifacts, perhaps computer science isn't a ("natural") science? Perhaps it is really a branch of engineering. Newell, Perlis, and Simon reply that it is both. But this is a major topic that we will look at in greater detail later (especially in Ch. 5).

The most interesting—and telling—objection to their view is that computer science is really the study, not (just) of computers, but of algorithms: very roughly, the programs and rules that tell computers what to do. (We'll devote a great deal of time later looking at what algorithms and programs are, so, at this point, I will just assume that you already have an idea of what they are and won't try to define them further.) For example, [Bajcsy et al., 1992, 1] explicitly mention "the (incorrect) assumption that. . . [computer science] is based solely on the study of a device. . .".

What is interesting about this objection is how Newell, Perlis, and Simon reply: They agree! They now say:

> In the definition [that is, in their definition quoted at the beginning of this section], 'computers' means. . . the hardware, their programs or algorithms, and all that goes along with them. **Computer science is the study of the phenomena surrounding computers**" [Newell et al., 1967, 1374, my emphasis].

Readers would be forgiven if they objected that the authors have changed their definition! But, instead of making that objection, let's turn to an interestingly different, yet similar, definition due to another celebrated computer scientist.

3.6 Computer Science Studies Algorithms

Donald Knuth, author of a major, multi-volume work on algorithms (called *The Art of Computer Programming* [Knuth, 1973]) as well as developer of the TEX computer typesetting system, gave this answer to our question:

> [C]omputer science is... **the study of *algorithms***" [Knuth, 1974b, 323].

He cited, approvingly, a statement by George E. Forsythe [Forsythe, 1968] that the central question of computer science is: What can be automated? (Presumably, a process can be automated—that is, done automatically, by a machine, without human intervention—if it can be expressed as an algorithm.)

He even noted [Knuth, 1974b, 324] that the name 'comput*ing* science' might be better than 'comput*er* science', because the former sounds like the discipline is the science of computing (what you do with algorithms) as opposed to computers (the tools that do the computing) (cf. [Foley, 2002], [Denning, 2013, 35][17]. As he pointed out, "a person does not really understand something until he teaches it to someone else. Actually a person does not *really* understand something until he can teach it to a *computer*, i.e., express it as an algorithm" [Knuth, 1974b, 327].[18] (For a commentary on this, see the *XKCD* cartoon in Figure 3.4 at the end of the chapter.)

But, as Knuth then went on to point out, 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, 323]. But are computers really necessary? Even if they are, does that mean that computer science is (as Newell, Perlis, and Simon claim) the study of computers?

Let's consider some similar questions for other disciplines: Do you need a compass and straight edge to study geometry (or can you study it just by proving theorems about points, lines, and angles)? Do you need a calculator to study physics or mathematics (or do they just help you perform calculations more quickly and easily)? Do you need a microscope to study biology? Even if you do, does that make geometry the study of compasses and straight edges, or physics and math the study of calculators, or biology the study of microscopes? With the possible exception of geometry, I think most people would say that these disciplines are not studies of those tools.[19] But the tools do make it easier to study the subject matter.

So, whereas Newell, Perlis, and Simon say that computer science 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*)![20] These are what are called "extensionally equivalent" descriptions: They may approach the discipline from

---

[17]Denning says: "I have encountered less skepticism to the claim that 'computing is science' than to 'computer science is science'."

[18]Cf. [Schagrin et al., 1985, PAGE], [Rapaport and Kibby, 2010, §2.4.2]. As the celebrated cellist Janos Starker once said, "When you have to explain what you are doing, you discover what you are really doing" [Fox, 2013].

[19]Geometry might be different, because Euclid may have considered geometry to be the study of which geometrical figures were constructible using only compass and straightedge [Toussaint, 1993] (see also [Goodman, 1987, §4]). We'll come back to this in Chapter 7.

[20]Similarly to Knuth, [Gal-Ezer and Harel, 1998, 80] say that the "heart and basis" of the field is "algorithmics" even though this does not "cover the full scope of CS".

different viewpoints (one from the viewpoint of a physical tool, one from the viewpoint of an abstract procedure), but the "bottom line" is the same—only the emphasis is different.[21]

   So, we now have two (only slightly different) definitions:

1. Computer science is the study of computers (and related phenomena such as the algorithms that they execute).

2. Computer science is the study of algorithms (and related phenomena such as the computers that execute them).

Both characterize computer science as a "study", but I think that it's pretty clear from their discussions that they think of it as a science. Is it? To fully answer that, we'll need to ask a preliminary question: What is science? We'll do that in Ch. 4. Here, let's look at a slight variant of the first of our two definitions.

3.7   Is Computer Science a Science?

> Debates about whether [computer science is] science or engineering can…be counterproductive, since we clearly ar both, neither, and more (mathematics is so much a part of what [it is]…. [Freeman, 1994, 27]

In a classic paper from 1976 [Newell and Simon, 1976], Newell and Simon updated their earlier characterization. Instead of saying that computer science is the science of computers and algorithms, they now said that it is the "empirical" "study of the phenomena surrounding computers", "not just the hardware, but the programmed, living machine". (Cf. [Abrahams, 1987, 472]: "Computer science focuses, not on the properties of particular computers, but on the properties of computers in general. It goes beyond the study of computers as objects to, at least, the study of computers as active agents"). This may not sound too different from what Newell and Simon said when they were writing with Perlis nine years earlier, but there are some interesting differences:

   First, here they emphasize the *empirical* nature of the subject: Programs running on computers are *experiments*, though not necessarily like experiments in other experimental sciences. In fact, one difference between computer "science" and other experimental sciences is that, in computer science, the chief objects of study (the computers and the progams) are not "black boxes", that is, things whose internal workings we cannot see directly but must infer from experiments we perform on them. Rather, because *we* (not nature) designed and built the computers and the programs, we can understand them in a way that we cannot understand more "natural" things.

---

[21]Here's a more mundane example of a similar phenomenon: Large supermarkets these days not only carry groceries, but also pharmaceuticals, greeting cards, hardware, etc. Large drugstores these days not only carry pharmaceuticals, but also groceries, greeting cards, hardware, etc. (And large "general stores" (like Walmart) also carry pretty much the same mix of products.) Although each kind of store "approaches" merchandising "from different viewpoints"—we tend to think of Walgreens as a drugstore and Wegmans as a supermarket—their "bottom line" is the same; only their emphasis is different. On "extensional equivalence", see [Rapaport, 2009].

Second, "science" may be the wrong term (note that the title of their paper mentions only 'empirical *inquiry*'). Computer science may not be a science (in the classic sense) because it doesn't always strictly follow the scientific (or "experimental") method. (We'll talk more about what that method is, in §4.6.) For example, often one experiment will suffice to answer a question in computer science, whereas in other sciences, numerous experiments have to be run.

Third, by "programmed, living machines", they mean computers that are actually running programs, not the machines just sitting there waiting for someone to use them, not the programs just sitting there on a piece of paper waiting for someone to load them into the computer, and not the algorithms just sitting there in someone's mind waiting for someone to express them in a programming language. Sometimes, a distinction is made between a program and a "process": 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. (We'll look at some of these distinctions in more detail later, in Chapter 12; on the program-process distinction, see [Eden and Turner, 2007, §2.2].)

If computer science is an empirical discipline (whether it's a science, a "study", or an "inquiry"), it should have some empirical results, or what Newell and Simon call "laws of qualitative structure". They give examples of these for well-established empirical disciplines: the cell doctrine (biology), plate techtonics (geology), the germ theory of disease (medicine), and the atomic theory (physics) [Newell and Simon, 1976, 115]. What about computer science? They cite two "laws of qualitative structure", but it should be emphasized that the two that they cite are due primarily to them and are highly controversial: the Physical Symbol System Hypothesis (which explains the nature of symbols) and Heuristic Search (which is a problem-solving method).

Here, we'll focus on the first of these,[22] which they state as follows [Newell and Simon, 1976, 116]:

> A physical symbol system has the necessary and sufficient means for general intelligent action.

Another way of stating this is:

> x is a physical symbol system if and only if x has the means for general intelligent action,

where[23]

> x is a physical symbol system $=_{def}$
>
> 1. x is a set of *physical symbols* with a grammatical syntax, together with a set of *processes* that manipulate the symbols (where a process can be thought of as a function or mapping or algorithm that takes symbols as inputs and produces (other) symbols as output), and

---

[22]We'll look at the second—heuristics—in Ch. 7.

[23]In this book, I use the symbol '$=_{def}$' to mean: "means by definition".

2. *x* is a physical *machine* that "produces through time an evolving collection of symbol structures" [Newell and Simon, 1976, 116].

In short, a physical symbol system is a physical Turing machine or universal computer. We will see in great detail what that means in Chapter 8, but for now you can think of it as your laptop computer, or even your smartphone.[24] SEE [Rapaport, 2012b] FOR MORE ON THIS.

(It is interesting to note that the use of symbols to accomplish tasks in the real, physical world bears some similarities to the way that magic is supposed to work [Stevens, 2001]. SAY MORE ABOUT WHAT MAGIC IS. See NOTES FOR NEXT DRAFT, 16, below.)

Note that Newell and Simon agree with Newell, Perlis, and Simon about *what* computer science studies, and they agree with Knuth about its fundamental nature. But let's examine these definitions a bit more closely before looking at others.

3.8   Physical Computers vs. Abstract Algorithms

As we saw at the end of §3.6, we have two apparently different views. Is computer science the science (or study) of the physical machine, or the abstract algorithm? Of course, as Newell, Perlis, Simon, and Knuth pointed out, each of these characterizations includes the other, so perhaps there really is no difference other than one of emphasis or point of view. But consider that, in order to study a computer, it seems to be necessary to know what it does; that is, it seems to be necessary to know what algorithm it 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; that is, the algorithm can be studied as a mathematical object, using mathematical techniques. 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 (and we'll see an objection in a moment), then it would seem that algorithms are really the essential object of study of computer science: Both views require algorithms, but only one requires computers.

But is it really the case that you cannot study computers without studying algorithms? Compare the study of computers with neuroscience: the study of brains. Although neuroscience studies both the anatomy of the brain (its static, physical structure) and its physiology (its dynamic activity), it generally treats the brain as a "black box": Its parts are typically named or described, not in terms of what they *do*, but in terms of *where they are located*. It is as if a person from the 19th century found what we know to be a laptop computer lying in the desert and tried to figure out what it was, how it worked, and what it did (see [Weizenbaum, 1976, Ch. 5] for the source of this kind of thought experiment). They might identify certain physical features: a keyboard, a screen, internal wiring (and if they were from the 18th century, they might describe these as buttons, glass, and strings), and so on. More likely, they would describe the device as we do the brain, in terms of the *locations* of the parts: an array of button-like objects on the lower half, a glass rectangle on the upper half,

---

[24]A more detailed exposition is given in [Newell, 1980].

and so on. But without knowledge of what the entire system and each of its parts was supposed to *do*—what their *functions* were—-they would be stymied. Yet this seems to be what neuroscientists study. Of course, modern neuroscience, especially modern *cognitive* neuroscience, well understands that it cannot fully understand the brain without understanding its processing (its algorithms, if indeed it executes algorithms). But this is a topic for another branch of philosophy: the philosophy of cognitive science (see [Fodor, 1968], [Gazzaniga, 2010], [Piccinini, 2010a]).

So, it seems that we cannot study computers without studying algorithms, but we can study algorithms without studying computers. So, even though it is quite obvious that computer science does, in fact, study both, it seems that a definition of computer science that focuses on its "essence", on what it "really is", would be one that takes algorithms to be central. So, let us provisionally summarize our investigation so far:

> Computer science is the science of algorithms.

Knuth is not the only person to make this claim: [Denning et al., 1989, 16, my emphasis] define computer science as the "*systematic study* of *algorithmic processes*—their theory, analysis, design, efficiency, implementation, and application—that describe and transform information". Note that they avoid the word 'science', perhaps for political reasons (so as not to offend engineers). It is with "implementation" that real, physical computers (as well as programming languages) enter their picture. But it is also interesting to note that Denning et al. immediately go on to cite a version of Forsythe's definition: "The fundamental question underlying all of computing is, What can be (efficiently) automated."

NOTE FOR NEXT DRAFT
The background to [Denning et al., 1989]'s definition-by-committee (that is, a "klunky" one designed to be acceptable to a variety of competing interests)[25] is that, first, "no fundamental difference exists between" computer science and computer engineering "in the core material"; second, there are "three major paradigms[:]...*theory*, [which] is rooted in mathematics...[,] *abstraction* (modeling), [which] is rooted in the experimental scientific method...[; and] *design*, [which] is rooted in engineering.... ...[I]n computing[,] the three processes are so intricately intertwined that it is irrational to say that any one is fundamental" [Denning et al., 1989, 10].
END NOTE

But there are still some questions to consider: First, there are the questions that we have been putting off: Is it a science? What is a science? Second, there are questions about whether there are any alternatives to it being a science (or other kind of study) of algorithms (or computers). Let's continue to put off the questions about science, and focus on what else computer science might be a science of.

---

[25]The standard joke about such definitions is that a camel is a horse designed by a committee. See http://en.wikipedia.org/wiki/Design_by_committee (accessed 22 April 2013).

3.9   Computer Science Studies Information

As we just saw, Denning et al. constrain the algorithmic processes to be only those that concern "information". Although this may seem to be overly narrow (after all, the algorithmic processes that undoubtedly underlie your use of Facebook on your laptop, tablet, or smartphone may not seem to be related to "information" in any technical sense), but others also focus on this: [Hartmanis and Lin, 1992, 164] define computer science this way:

> What is the object of study [of computer science 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. (Question: Are humans included among the "machines and systems"? After all, *we* represent and process information, too!).

Similarly, [Bajcsy et al., 1992, 1] say that computer science is "a broad-based quantitative and qualitative study of how information is represented, organized, algorithmically transformed, and used." More precisely, "Computer science is the discipline that deals with representation, implementation, manipulation, and communication of information" [Bajcsy et al., 1992, 2]. I think their second characterization is too broad; I can imagine other disciplines (including journalism) that also deal with these four aspects of information. But their first definition contains a crucial adverb—'algorithmically'—and that's what makes computer science unique. Indeed, [Hartmanis and Lin, 1992, 164] say that "The key intellectual themes in CS&E [computer science and engineering] are algorithmic thinking, the representation of information, and computer programs." But the second of these, although an important branch of computer science (in data structures, knowledge representation in AI, and database theory), is also studied by logicians. And the third— computer programming—although clearly another important branch of computer science (in software engineering and program verifications) is, arguably, "merely" the implementation of algorithms. So, once again, algorithms come to the fore.

Are there any other kinds of algorithmic processes besides those that manipulate information? We will return to this question [WHERE?].

3.10   Computer Science Is a Science of Procedures

Here is yet another definition, though it is one that slightly modifies the algorithmic point of view:

> Computer Science is a natural science that studies procedures. [Shapiro, 2001]

The "procedures" are, or could be, carried out in the real world by physical (biological, mechanical, electronic, etc.) agents. 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. Procedures include, but are not limited to, algorithms. Whereas algorithms are typically considered to halt and to produce correct solutions, the more general notion of *procedure* allows for an algorithm-like entity that may be vague (e.g., a recipe), that need not halt (e.g., that goes into an infinite loop either by accident or on purpose, as in an operating system or a program that computes the infinite decimal expansion of π), or that does not necessarily produce a correct solution (e.g., a chess procedure does not always play optimally). (We will return to these issues in Ch. 7.) Does *computer* science really study things like recipes? According to Shapiro (personal communication), the answer is 'yes': An education in computer science should help you write a better cookbook (because it will help you understand the nature of procedures better)! (For a nice discussion of the difficulties of writing recipes, see [Sheraton, 1981].)

And computer science 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 suggested in their discussion of empirical results, there are "fundamental principles" of computer science as a science. Shapiro cites two: The Church-Turing Thesis to the effect that any algorithm can be expressed as a Turing-machine program (which we will discuss in Chs. 7, 8, and 10) and the Boehm-Jacopini Theorem that codifies "structured programming" (which we will discuss in Ch. 7).

Where do computers come in? A computer is simply "a general-purpose procedure-following machine". But does a computer "follow" a procedure, or merely "execute" it? (We'll come back to this in Ch. 10.)

How about the role of information? According to [Shapiro, 2001], computer science is *not just* concerned with abstract information, but also with procedures that are linked to sensors and effectors that allow computers to operate in the real world. (We'll examine the relationship between computers and the world in Ch. 17.)

3.11 Computer Science Is *Not* a Science

We have a number of open questions: Insofar as computer science studies both algorithms and computers, we need to look further into what, exactly, algorithms are (and how they are related to the more general notion of "procedure"), what kinds of things they manipulate (information? symbols? real-world entities?), what computers are, and how computers and algorithms are related to each other.

Another open question that we should look at is whether computer science is a science. Don't forget, we are calling the subject 'computer *science*' only for convenience; it is not a tautology to say that computer science is a science nor is it a self-contradiction to say that it computer science is not a science. Think of the subject

as being called by the 16-letter word 'computerscience' that has as much to do with the words 'computer' and 'science' as 'cattle' has to do with 'cat'.

[Brooks, 1996] says that it isn't an analytic science, presumably because, according to him, it is not concerned with the "discovery of facts and laws". Whether or not Brooks's notion of what science is is accurate will be our focus in Chapter 4. For now, it will have to suffice to point out that computer scientists who study the mathematical theory of computation certainly seem to be studying scientific laws and that computer scientists like Newell, Simon, and Shapiro have pointed out that the physical symbol system hypothesis or the Boehm-Jacopini theorem certainly seem to be scientific theories, facts, or laws: "Computer *programming* is an exact science in that all the properties of a program and all the consequences of executing it in any given environment can, in prinicple, be found out from thetext of the program itself by means of purely deductive reasoning" [Hoare, 1969, 576, my italics].[26] So, it certainly seems that at least part of computer science is a science. But perhaps there are parts that aren't.

In particular, Brooks argues that computer science is "an engineering discipline"; he views the computer scientist as a toolmaker. Computer scientists are "concerned with *making things*" like physical computers and abstract tools like algorithms, programs, and software systems for others to use. He uses J.R.R. Tolkien's phrase the "gift of subcreation" to describe this concern. 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). And the purpose of the tools is to enable us to manage complexity. So, "the discipline we call 'computer science' " is really the synthetic—i.e., engineering—discipline that is concerned with computers.

Of course, before we can seriously consider whether computer "science" is really an engineering discipline as opposed to a science, we'll need to look into what engineering is, just as we will have to look into what science is. But here is an argument for your consideration: Perhaps computer science is both a science and an engineering discipline. First, it makes no sense to have a computer without a program: The computer wouldn't be able to do anything. (It doesn't matter whether the program is hardwired (in the way that a Turing machine is; see §8.12) or is a piece of software (like a program inscribed on a universal Turing machine's tape; see §8.13). But, second, it also makes no (or at least very 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 (see Ch. 16), study its computational complexity [Loui, 1996, Aaronson, 2013], etc.). But what good (what fun) would it be to have, say, a program for passing the Turing test that never had an opportunity to pass it? So, computers require programs in order for them to do anything, and programs require computers in order for the program to be able to do anything. The computer scientist R.W. Hamming said:

> Without the [computing] machine almost all of what we [computer scientists] do would become idle speculation, hardly different from that of the notorious Scholastics of the Middle Ages.

[Hamming, 1968, 5] This is reminiscent of the 18th-century philosopher Immanuel

---

[26]We'll look into this claim in more detail in Ch. 16.

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, 93 (A51/B75)][27]

Similarly, we can say: Computers without programs are empty; programs without computers are blind.[28] Only through their union can computational processing arise. So, computer science must be both a science (that studies algorithms) and an engineering discipline (that builds computers).

But now, I think, we're ready to look into all of these issues in more detail, so that we'll be able to have a better grasp of exactly what computer science is. Before doing that, let's sum up to see where we are.

3.12 Summary

What is computer science? As we have seen, there are several different, but related, answers to this question.

The answers differ on several dimensions:

Category (or "genus"):[29]

Computer science is. . .

1. a science

2. a (systematic) "study"

3. a branch of engineering

Qualification ("kind" of "science" or "study"; i.e., "differentia"):

Computer science is. . .

1. (a) mathematical

    (b) empirical

2. (a) natural

    (b) artificial

---

[27]For an informal presentation of some of Kant's ideas, see [Cannon, 2013]. For a more philosophically sophisticated introduction, see [Rohlf, 2010] and other articles on Kant in the online *Stanford Encyclopedia of Philosophy* at:
http://plato.stanford.edu/search/searcher.py?query=kant (accessed 15 November 2013).

[28]Or is it the other way round?: Programs without computers are empty; computers without programs are blind?

[29]Here, I am using the word 'genus' in the sense of Aristotle's notion of a "definition by genus and differentia", where a term is defined by reference to its "genus" (what kind of thing it is) and its "differentia" (how it differs from other things of that kind); see the *Wikipedia* article "Genus-differentia definition", http://en.wikipedia.org/wiki/Genus-differentia_definition, accessed 30 September 2011.

(These "cross-cut" each other: That is, there are 4 possibilities.)

Subject matter: 1. computers (and related phenomena)

∴ empirical, artificial?

2. (a) algorithms (and related phenomena)
    (b) procedures (and related phenomena)

∴ natural?, mathematical?, empirical/experimental?

3. information (and related phenomena)

∴ natural?, mathematical?, empirical/experimental?

Finally, these may all turn out to be *extensionally equivalent*—that is, it may be the case that computer science is an engineering discipline of computers if and only if it is also science of algorithms, etc. The only differences may be the "focus" or "point of view" that depend on what the "related phenomena" are in each case.

A more complete answer is going to depend on answers to many other questions, including (but not limited to):

- What is science?

- What is engineering?

- What is a computer?

- What is an algorithm?

We will discuss some, but not all, of these topics later. For example, we won't discuss mathematics vs. empirical sciences or natural vs. artificial sciences; if you are interested in these questions, you should read books on the philosophy of science or the philosophy of mathematics.

But we'll discuss science vs. engineering (Ch. 4 and Ch. 5), computers (Chs. 6 and 9), and algorithms and procedures (Ch. 7). And we'll begin by asking, "If computer science is a science, what is science?".

3.13 NOTES FOR NEXT DRAFT

1. Need to split the question of what CS is into two parts:

    (a) What is its object? (What does it study/investigate?)

      - Might want to put part of this off to Ch. 9 on what a computer is. [Barwise, 1989] identifies three ways to understand what a computer and computation is, in terms of what they "traffic" in: numbers, symbol strings, information.

    (b) What is its methodology? (Is it a sci, a branch of engg, math?...)

    (c) see spreadsheet in Dropbox:WhatIsCS.xlsx

    Actually, there's a prior division: CS is nothing but something else (the John Ho/CS is math/CS is EE arguments; bring in microscopy here) vs. CS is its own discipline; in the latter case, it has a methodology and an object (see above). Still, it might be the case that its methodology and/or its object are (also) those of some other discipline. Finally, it may have a new, unique methodology and/or object (sui generis).

2. Then: Need to make a complete list of all published alternatives:

    (a) object: computers, computing/algorithms/procedures, info, ...
       method: sci, not a sci, engg, math, "study"...

    Possibly do all of this chronologically, rather than "logically":
    1967: newell/simon/perlis: sci of comprs (etc.)
    1974: knuth: study of algs (etc.)
    1976: newell/simon: emp'l study/artif sci of comprs etc.
    1987: loui: "new kind of" engg of algs for procg info
    1989: denning: systematic study of alg'ic procs over info
    1992: hartmanis/lin: sci & engg of info
    1996: brooks: engg of comprs
    2001: shapiro: nat sci of procedures

3. Relevant to this might be Parlante's suggestion that computer science is "a sort of spectrum. . . with 'science' on the one end and 'engineering' on the other." [Parlante, 2005, 24]

4. [Shannon, 1950], writing way before AI was created, wrote a "paper. . . concerned with the problem of constructing a computing routine or 'program' for a modern general purpose computer which will enable it to play chess." That is, his pricipal question was whether chess can be *mathematically* analyzed and, in particular, whether the kind of mathematical analysis to be given was a *computational* one, presumably, a *procedural* one. (Another way of phrasing the concern of his paper is: Can chess be played rationally?) This is clearly a computer science problem from our modern perspective. So, the methodology he advocates is that of mathematical analysis, and, in particular,

computational analysis. And the object of the analysis is a procedure—the procedure of playing chess.

5. My def: Computer science is centrally concerned with the question: Which functions are computable? Various branches of CS are concerned with identifying which problems can be expressed by computable functions. For instance, the central question of AI is whether the functions that describe cognitive processes are computable. And 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. (This is closest to Knuth's definition, I think.)

   E.g., devising computational methods of fingerprint identification *is* CS, because it's computer scientists who are trained in procedural methods for solving problems. Though it is *applied* CS, perhaps.

   E.g., Shannon's 1950 paper on chess: The principal question is: Can we mathematically analyze chess? In partic, can we *computationally* analyze it (suggesting that computational analysis is a branch/kind of mathematical analysis)—i.e., can we analyze it procedurally? I.e., can we play chess rationally?

   E.g., is the weather computable? See [Hayes, 2007a].

   E.g., 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 grounds that no one in the university administration had ever been able to produce such a schedule). So, is final-exam-scheduling computable?

6. Discussion of Arden 1980:

   (a) (Untitled) preface on the (unnumbered) pages immediately preceding p. 1:
       - A riff on Forsythe's 1968 quote.
       - There's an interestingly subtle difference between my "What can be computed?" and Forsythe's "What can be automated?": The latter allows, a bit more clearly, the possibility of including computers, and not just computing, in the subject matter, because 'automation' has a machine-oriented connotation that 'computation' lacks.

   (b) p 4: Distinguishes between "computer science research" and "computer engineering research", suggesting that he (or they) think it's easier/simpler to keep these separate, rather than trying to fit both under a single label.

   (c) Sections "About Names and Labels" (pp. 5–7):

- Has a useful discussion about the meanings of such terms as 'computation', 'computer', 'information', etc., and about 'science' vs. 'engineering'.
- p. 6: Argues that neither math nor computer science are sciences, because their objects are not natural phenomena. In the case of math, the object is "human-produced systems...concerned with the development of deductive structures". But this gets us too deeply into the controversial philosophy of math (Platonism vs. logicism vs. "syntacticism" vs. intuitionism. In the case of computer science, the object is "man-made". But what is the object? Computers? Yes, they're clearly man-made, and this leads us back to the Perlis et al. arguments. Algorithms? They're only man-made in whatever sense mathematical structures are.
- pp. 6–7: Argues that engineering is concerned with "implementation, rather than understanding" (6), which "is the best distinction" between engineering and science (7). Because of multiple implementations of a single theory, questions of efficiency come to the fore in engineering (7), and "much of computer science is concerned with...efficiency" (7). But surely complexity theory is on the mathematical/scientific side of the science/engineering border.
- p 7: But then suggests that implementations need not be physically realized (cf. [Rapaport, 1999, Rapaport, 2005b]), so that the implementation/understanding distinction doesn't really capture the engineering/science distinction. Suggests that the engineering/science spectrum matches the physical/theoretical spectrum, with no clear dividing line, and chalks this up to the lack of a natural phenomenon that is the object of study. But [Shapiro, 2001] argues that procedures are the natural phenomena under study.

(d) Section "Some Definitions", pp. 5-10:

- Accepts [Newell et al., 1967]'s "the phenomena related to computers" (Arden, p.7) as the subject of study.
- pp. 7–8: Points out that the non-natural nature of this subject matter is mitigated by its complexity.
- p. 8: Suggests a dilemma: Natural systems are ipso facto interesting to study (presumably because they're "out there", so we should be curious about them). But why study a "synthetic" system if it has no "rationale"? "On the other hand, ...if there is external motivation", why not study that external motivation directly?
    i. But the first horn of the dilemma is countered by "pure" mathematics.
    ii. Ant the second horn can be countered by taking [Shapiro, 2001]'s position.
- Contra both [Newell et al., 1967] and, in particular, [Knuth, 1974b], "the *study of algorithms* and the *phenomena related to computers* are

not coextensive, since there are important organizational, policy, and nondeterministic aspects of computing that do not fit the algorithmic mold" (p. 9).

    i. I don't think that either [Newell et al., 1967] or [Knuth, 1974b] had those things in mind.

    ii. And if "phenomena related to computers" is taken that widely, then it encompasses pretty much everything, thus making any definition based on such a wide notion virtually useless. The classical sciences (physics, chemistry, biology, etc.) also have "important organizational, policy, and nondeterministic aspects", but those aren't used in trying to define what those sciences are about.

- Suggests, but does not endorse, a "committee-produced, all-purpose" definition: "computer science is the study of the design, analysis, and execution of algorithms, in order to better understand and extend the applicability of computer systems" (p. 9). Note that this avoids the science/engineering quandary, by its use of 'study', and tries to cover all the ground. Suggests, however, that the entire book of which this is the overview should be taken as the "elaboration" of this definition, which is a bit like saying that computer science is what computer scientist do.

(e) Contains the following "sample questions" that are "more specific" than "What can be automated?" (p. 10):

    i. Are there useful, radically different computer organizations yet to be developed?

    ii. What aspects of programming languages make them well suited for the clear representation of algorithms in specific subject areas?

    iii. What principles govern the efficient organization of large amounts of data?

    iv. To what extent can computers exhibit intelligent behavior?

    v. What are the fundamental limits on the complexity of computational processes?

    vi. What are the limits on the simulation of physical systems?

    vii. How can processes be distributed in a communicating network to the advantage of microtechnology?

    viii. To what extent can numerical computation be replaced by the computer manipulation of symbolic expressions?

    ix. Is it possible to replace a major part of system observation and testing with automated proofs of the correctness of the constituent algorithms?

(f) Also contains the following list of "subject areas" of CS:

    i. Numerical computation

    ii. Theory of computation

      iii. Hardware systems

      iv. Artificial intelligence

      v. Programming languages

      vi. Operating systems

      vii. Database systems

      viii. Software methodology

From this, he concludes that "What can be automated?" means: "What are the limits, imposed by complexity, on the application of computers?" NOTE: I can't find this quote in the Overview. In any case, asking what can be automated logically entails the possibility that some things cannot be automated, hence that there are limits. But these limits may have nothing to do with complexity: They could be entirely theoretical (as in the Halting Problem), or practical (time and space limitations, some of which might be theoretically interesting, as in computational complexity theory, but some of which might be uninteresting, such as if a research group runs out of funds before being able to automate something), or ethical.)

(g) Contains a good, short history of modern computing (§"A Brief History", pp. 10–13.

(h) "the basic question[:] ... what can and what should be automated" (p. 29). Note that "should" can have both an efficiency answer (*X* shouldn't be automated, because it's more efficient not to do so) and an ethical answer (*X* shouldn't be automated because it's unethical/immoral to do so).

7. From Abelson & Sussman 1996:

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

"How to" is certainly important, and interestingly distinct from "what is". But consider Euclid's *Elements*; it was originally written in "how to" form: To construct an equilateral triangle, do such-and-such (http://tinyurl.com/kta4aqh[30] [Toussaint, 1993]). But today it is expressed in "what is" form: The triangle that is constructed by doing such-and-such is equilateral. So, is there really a difference between "how to" and "what is"? (See also [Goodman, 1987, §4].)

---

[30]http://www.perseus.tufts.edu/hopper/text?doc=Perseus%3Atext%3A1999.01.0086%3Abook%3D1%3Atype%3DProp%3Anumber%3D1

8. On Malpighi:

   "The debate over the appropriate place of computing in grade schools and high schools echoes the debate in universities decades ago, when computers and software were initially seen as mere plumbing. And certainly not something worthy of study in its own right. A department of computer science? Why not a department of slide rules?" From: Lohr, Steve (2008) "Does Computing Add Up in the Classroom?", New York Times (1 April).

9. include various taxonomies of subfields of CS; see
   http://www.cse.buffalo.edu/ rapaport/584/whatiscs.html 9b

10. discuss sui generis; include this quote

    "Trying to categorize computing as engineering, science, or math is fruitless; we have our own paradigm." Denning & Freeman 2009

11. Fant 1993 argues that the notion of "process" is more central to CS than the notion of "algorithm". His main argument seems to be that the notion of "algorithm" is too narrow, requiring halting, disallowing randomization or probabilisitic methods, etc. He is also more concerned with the "expression" of an algorithm than with the algorithm itself, the latter of which he considers to be an object of mathematical study.

    See also Guzdial 2008, esp citation to Alan Perlis; Also on the nature of both procedural thinking and computational thinking more generally, see: Carey 2010.

12. Two quotes from Bernard Chazelle, Prof. of CS, Princeton:

    CS is the new "new math," and people are beginning to realize that CS, like math, is unique in the sense that many other disciplineswill have to adopt that way of thinking. It offers a sort of conceptual framework for other disciplines, and that's fairly new."

    "We're at the tail end of Moore's Law....But this will be the best thing that can happen to CS. Moore's Law has been tremendously beneficial to society. At the same time, it's been so damn powerful that it has set back the development of algorithms. But that's about to change. Any student interested in science and technology needs to learn to think algorithmically. That's the next big thing."

    From Anthes 2006

13. On difference between CS and science:

    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. 2008:63

14. On CS as math:

> That computer science is somehow a mathematical activity was a view held by many of the pioneers of the subject, especially those who were concerned with its foundations. At face value it might mean that the actual activity of programming is a mathematical one. Indeed, at least in some form, this has been held. But here we explore a different gloss on it. We explore the claim that programming languages are (semantically) mathematical theories. This will force us to discuss the normative nature of semantics, the nature of mathematical theories, the role of theoretical computer science and the relationship between semantic theory and language design. Turner 2010

15. On the nature of defs, discuss the def of 'planet'. See, e.g., [Lemonick, 2015].

> Exact definitions are undoubtedly necessary but are rarely perfect reflections of reality. Any classification or categorization of reality imposes arbitrary separations on spectra of experience or objects. Matt Craver, letter to the editor of SciAm, May 2007, p. 16. also see reply by Soter on same page.

Cf. the "def" of 'alg', culminating in the Ch-T thesis.

Typically, when formalizing an informal notion, you exclude some "old" things—i.e., things that were informally considered to fall under the notion— and you include some "new" things—i.e., things that you hadn't previously considered to fall under the notion. The attitude towards the former is sour grapes: Our intuitions were wrong; those things really weren't Xs after all. The attitude towards the latter is: Wow! That's right! Those things really *are* Xs! Cf. not only the Ch-T thesis, but also the def of "thinking" as a kind of computation.

16. On magic:

Figure 3.2: ©2014, John L. Hart FLP

> The entities that populate the referent node [of the "meaning triangle"
> consisting of words, concepts (abstract, or in the head), and referents
> (concrete, and "part of the external world")] are usually construed
> as "objects out there"... because they are independent of what any
> person thinks or says. The qualifier "out there" is to capture the
> intuition that words can be uttered, and thought can be generated,
> but no amount of thinking or talking is going to bring a table or cat
> or chair into being. (p. 507) Regoczei, Stephen; & Hirst, Graeme
> (1990), "The Meaning Triangle as a Tool for the Acquisition of
> Abstract, Conceptual Knowledge", *International Journal of Man-
> Machine Studies* 33: 505–520.

That's exactly what magic does: "spells" bring physical objects and events into
being. (syntax produces semantics!)

But

> There are socially constructed abstract entities, especially in the
> fields of law, culture, interpersonal interactions, organizations, and
> mathematical sciences, which *are* brought about through talking and
> thinking by individuals within a group. (p. 507)

I.e., they are magical!

> 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, 7–8, my emphasis].

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. As Arthur C. Clarke said, "Any sufficiently advanced technology is indistinguishable from magic" (http://en.wikipedia.org/wiki/Clarke's_three_laws).

> 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, 721, col. 1]

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

> In anthropology, magic generally means beliefs in the use of symbols to control forces in nature.... [Stevens, 1996, 721, col. 1]
> A definition of magic can be constructed to say something like the following: Magic involves the human effort to manipulate the forces of nature directly, through symbolic communication and without spiritual assistance. [Stevens, 1996, 723, col. 2].

The first quote is even better, because it has no "mystical" overtones. Clearly, programming involves exactly that kind of use of symbols (which we'll look at in more detail when we investigate the relation of computers to the world, in Ch. 17). The second quote either clearly includes the use of computers, or else is too broad a definition of 'magic'!

> [James G.] Frazer [[Frazer, 1915]] had suggested that primitive people imagine magical impulses traveling over distance through "a kind of invisible ether." [Stevens, 1996, 722, col. 1]

That sounds like a description of electricity: Think of electrical currents running from a keyboard to a CPU, or even of information traveling across the Internet.

> 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, 722, col. 2, citing Bronislaw Malinowski]

Ignoring "the dramatic expression of emotion", use of a compuer involves gestures with a mouse, a trackpad, or a touchscreen; the computer itself can be thought of as "imbued with power" when we issue a command, either spoken or typed; and the words used by the programmer or user are surely important—so the "rite" criterion is satisfied. It's not clear what Stevens (or Malinowski) have in mind by 'formula' or by 'condition of the performer', but 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".

> [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, 724, col. 1, my italics]

First, we see this happening in computers when we treat icons (which are a kind of symbol) on a desktop or a WYSIWYG word processor as if they were the very things they represent. But, perhaps more significantly, we see this in the case of those computer simulations in which the simulation of something *is* that (kind of) thing, as in online banking, where the simulation of transferring funds between accounts *is* the transferring of funds. (See [Rapaport, 2012b, §8] for further discussion.) Computers are not just metaphorically magic (as Arthur C. Clarke might have said); they *are* magic!

> 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, 99]

> I am tempted to lay claim to the term "magic". A lot of what computer scientists do is now seen by the lay public as magical. Programmers (especially systems engineers) are often referred to as gurus, sorcerers, and wizards. Given the lofty goals of white magic, understanding the power of names and the value of pure thought, the power of labels is indeed attractive. However, many historically compelling reasons argue against this connotation, including the sad history of Isaac Newton's alchemical pursuit of the philosopher's stone and eternal life, and the religiously driven 17th century witch trials in Salem, MA, and other seemingly rational explanations for irrational behaviors. [Crowcroft, 2005, 19n2]

This would make a nice intro and closing quote for a discussion of computer science as magic.

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

> 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*. 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." Process P behaves as it does because of a symbolic "spell" cast at a distance from P itself.

[Samuelson et al., 1994, 2324n44,n46; 2325n47] cite an NRC report that 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, 2334]

If that isn't magic, I don't know what is!

17.   I consider computer science, in general, to be the art and science
       of representing and processing information and, in particular,
       processing information with the logical engines called automatic
       digital computers. [Forsythe, 1967a, 3]

    (a) both art & sci; in what sense art?—he never says. (but see below)

    (b) And why info? Possibly because of this earlier (1963) statement:

        Machine-held strings of binary digits can simulate a great many
        kinds of things, of which numbers are just one kind.   For
        example, they can simulate automobiles on a freeway, chess
        pieces, electrons in a box, musical notes, Russian words, patterns
        on a paper, human cells, colors, electrical circuits, and so on.
        (Quoted in [Knuth, 972b, 722].)

        - This is one of the ”Great Insights”. What's common to all of these,
          encoded/abstracted as bit strings, is the information contained in them.
        - And perhaps because some of these are not “sciences”, if CS is going
          to study them, it must be an “art”.

    (c) Forsythe goes on to say:

        [A] central theme of computer science is analogous to a central
        theme of engineering science—namely, the *design* of complex
        systems to optimize the value of resources. [Forsythe, 1967a, 3].

        Note that he doesn't say that CS *is* engineering.
        And note that he speaks of “engineering *science*”, suggesting that he thinks
        of engineering as a kind of science; certainly, engineering and science per
        se are on the same side as opposed to art.

    (d) That Forsythe sees CS as more of a science than a branch of engineering
        is emphasized when he says, “Probably a department of computer science
        belongs in the school of letters and sciences, because of it close ties with
        departments of mathematics, philosophy, and psychology. But its relations
        with engineering departments. . . should be close.” [Forsythe, 1967a, 6].

18. In order to teach “nontechnical students” about computers and computational
    thinking, and “specialists in other technical fields” about how to use computers
    as a tool (alongside “mathematics, English, statistics”), and “computer science
    specialists” about how to “lead the future development of the subject”,

        The first major step. . . is to create a department of computer
        science. . . . . . . Without a department, a university may well acquire
        a number of computer scientists, but they will be scattered and
        relatively ineffective in dealing with computer science as a whole.
        [Forsythe, 1967a, 5]

    Compare the history of microscopy.

19. Outline of revised version of Newell, Simon, Perlis:

(a) Although they *say* initially that computer science is the science of computers, they almost immediately modify that in their Obejction 3, where they say that by 'computers' they mean " 'living computers'— the hardware, their programs or algorithms, and all that goes with them. *Computer science is the study of the phenomena surrounding computers*" [Newell et al., 1967, 1374, my emphasis]. And, as they point out in their reply to Objection 5, "Phenomena define the focus of a science, not its boundaries" [Newell et al., 1967, 1374].

(b) They also say that astronomy is the science of stars [Newell et al., 1967, 1373]. Edsger W. Dijkstra once noted that astronomy was *not* the science of telescopes CITE; telescopes are used to study the stars.[31] Similarly, one might say that computers are used to study algorithms (and that algorithms, in turn, are used to study various phenomena in mathematics, other sciences, etc.). So, Dijkstra would say that computer science should not be considered a science of the instrument that is used to study algorithms, any more than astronomy is considered the science of the instrument that is used to study stars. Newell et al. address this in their Objection 4: "The computer is such a novel and complex instrument that its behavior is subsumed under no other science" [Newell et al., 1967, 1374]. But the same was once said about microscopes. And we know what happened to the "science" of microsopy. DISCUSS MICROSCOPY.

(c) Moreover, that there is no other *single* science that studies computer behavior may be irrelevant. Newell et al. deal with this in their Objection 5 "FORMER DEAN" STORY FROM ABOVE.

(d) At the end, they even allow that the study of computers may also be an engineering discipline [Newell et al., 1967, 1374]. So, they ultimately water down their definition to something like this: Computer science is the science and engineering of computers, algorithms, and other related phenomena.

20. When the discipline was first getting started, it emerged from various other disciplines: "electrical engineering, physics, mathematics, or even business" [Hamming, 1968, 4]. (In fact, the first academic computer programming course I took (in Fortran)—the only one offered at my university in the late 1960s—was given by the School of Business.)

21. CS questions:

(a) What can be computed? (computability vs. non-computability)

> The question "What can be automated?" is one of the most inspiring philosophical and practical questions of contemporary civilization. (George Forsythe, 1968; quoted in [Knuth, 972b, 722–723].)

---

[31]Dikjstra 1987, cited in Tedre & Sutinen 2008, said that calling the discipline 'computer science' "is like referring to surgery as 'knife science'." This may be true, but the problem, of course, is that we have no term corresponding to 'surgery'.

(b) What can be computed efficiently? (complexity theory)

(c) What can be computed practically? (SW engineering considerations) (Cf. [Hamming, 1968]

(d) What can be computed physically? How can computations be performed practically? (engineering considerations)

(e) What should be computed? (ethical considerations)

22.   [I]n the long run the solution of problems in field X on a computer should belong to field X, and computer science should concentrate on finding and explaining the principles ["the methodology"] of problem solving [with computers]. [Forsythe, 1967b, 454]

(a) So, the kind of work that we did for QIKR was really philosophy, not CS.

(b) As an example of such "methodological research", he cites " 'interactive systems,' in which a man [sic] and a computer are appropriately coupled...for the solution of scientific problems" [Forsythe, 1967b, 454]. This is a kind of oracle computation.

(c) Presumably, the idea is for CS proper to abstract away from solving specific problems in X and to concentrate on general methods of problem solving using computers. But what does that come down to besides how to write algorithms? (It does ignore how to make computers.)

23. Computer science as the mathematics of computing:
E.W. Dijkstra [Dijkstra, 1974] argues that "programming [i]s a mathematical activity". He doesn't explicitly say that (all) of computer science is a branch of mathematics, but I think that he would agree, and that it is quite clear, that large portions of computer science can be considered to be branches of math—not only programming, but also all of the theory of computability and the theory of complexity (and no doubt other aspect of computer science, too).

His argument for programming to be considered as math is, roughly, this [Dijkstra, 1974, 608]:

(a) A discipline D's assertions are:
    i. "unusually precise",
    ii. "general in the sense that they are applicable to a large (often infinite) class of instances", and
    iii. capable of being reasoned about "with an unusually high confidence level"
iff D is a mathematical discipline.

(b) Programming satisfies "characteristics" 23(a)i–23(a)iii.

(c) ∴ Programming is a mathematical discipline.

He does not argue for premise 23a, taking the "only if" half (mathematical disciplines satisfy 23(a)i–23(a)iii) as something that "most of us can agree upon", and implicitly justifying the "if" half (disciplines that satisfy 23(a)i–23(a)iii are mathematical) on the grounds that the objects of mathematical investigation need not be such usual suspects as numbers, shapes, etc., because what matters is *how* objects are studied, not *what* they are.

And he argues for premise 23b on the grounds that programming clearly requires extraordinary precision, that programs can accept a wide range of inputs (and thus are general), and that contemporary program-verification techniques are based on logical reasoning. We will look into those techniques in Ch. 16; for now, let's assume that this is true.

Are programs really general in the sense that mathematical assertions are? A typically general mathematical assertion might be something like this: For any triangle, the sum of its angles is 180 degrees. In other words, the generality of mathematical assertions comes from their being universally quantified ("for any x..."). Is that the kind of generality that programs exhibit? A program (as we will see more clearly in Ch. 7) computes a (mathematical) function. So, insofar as mathematical functions are "general", so are programs. Consider a simple mathematical function: $f(x) = 2x$. If we universally quantify this, we get: $\forall x[f(x) = 2x]$. This seems to be general in the same way that our assertion about triangles was, so I think we can agree with Dijkstra about programming being mathematical.

24. More on Knuth:

   (a)   Why should we give so much emphasis to teaching how to use computers, if they are merely valuable tools like (say) electron microscopes? [Knuth, 1974b, 324]

   - Nice lead-in to the microscopy section.
   - His response is that "computers are more than this", that "[t]he phenomena surrounding computers are immensely varied and complex, requiring description and explanation" [Knuth, 1974b, 324]. But that can be taken two ways: Algorithms are the phenomena surrounding computers, and surely they are an appropriate object of study. Alternatively, because the phenomena that *surround* computers are an appropriate object of study, that makes their focus, namely, computers, also appropriate (unlike microscopes, the construction of which doesn't illuminate biology directly).

   (b)   [L]ike electricity, these phenomena [surrounding computers] belong both to engineering and to science. [Knuth, 1974b, 324]

   - So maybe when Knuth says that computer science is the "study" of algorithms, by 'study' he means both science *and* engineering. In what sense does the study of electricity belong both to engineering

and to science? Certainly, the science of physics studies electricity as a physical phenomenon. And, certainly, electrical engineering studies electricity from an engineering perspective. But physics and electrical engineering are typically considered to be separate (albeit related) disciplines. So, perhaps the same should be said for computer science (which would study algorithms) and computer engineering (which would study computers and, perhaps, software engineering).

- Cf. [Abrahams, 1987, 472]: "Computer science is both a scientific discipline and an engineering discipline. ...The boundary [between "the division of computer science into theory" (that is, science) "and practice" (that is, engineering)] is a fuzzy one."

(c) Knuth is quite clear that he does not view computer science as a branch of math, or vice versa [Knuth, 1974b, §2], primarily because math allows for infinite searches and infinite sets, whereas computer science presumably does not. But there is no reason in principle why one couldn't write an algorithm to perform such an infinite search. The algorith could not be executed, perhaps, but that is a physical limitation, not a theoretical one. I think a better contrast is that mathematics makes declarative assertions, whereas computer science is concerned with procedural statements ([Loui, 1987, 177] makes a similar point, quoting Abelson & Sussman, in arguing that computer science is not mathematics). And Knuth suggests this in §3. (But this distinction is hard to make precise. First, consider programming languages such as Prolog, where each statement can be interpreted either declaratively ("p if q") or imperatively ("to make p true, first make q true"). Second, any imperative statement can be converted to a declarative one (consider Euclidean geometry, which was originally expressed in imperative language: To construct an angle bisector, do this...; but most modern treatments are declarative: The angles created by doing this are equal. [Toussaint, 1993]))

25. Is computer science an art or a science?

(a) Only Forsythe suggests that it is, at least in part, an art.

(b) But [Knuth, 1974a] defends his use of 'art' in the title of his multi-volume classic *The Art of Computer Programming* [Knuth, 1973]. He does not say that all of computer science is an art, but he does argue that 'art' can be applied to, at least, computer programming, not in opposition to 'science', but alongside it.

(c) He gives a useful survey of the history of the term, and concludes that "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, 668]. This suggests that being an art is a possibly temporary stage in the development of our understanding of something, and that our understanding of computer

programming is (or at least was at the time of his writing) not yet fully scientific. If some subject is never fully scientifically understood, then what is left over remains an art:

> Artificial intelligence has been making significant progress, yet there is ahuge gap between what computers can do in the foreseeable future and what ordinary people can do. . . . [N]early everthing we do is still an art. [Knuth, 1974a, 668–669].

(d) But he then goes on to give a slightly different meaning to 'art', one more in line with the sense it has in 'the fine arts', namely, as "an art *form*, in an aesthetic sense" [Knuth, 1974a, 670]. In this sense, programming can be "beautiful" and "it can be like composing poetry or music" [Knuth, 1974a, 670]. This not inconsistent with programming being considered a science or a branch of mathematics (which we'll discuss in Ch. 16); indeed, many scientists and mathematicians speak of theories, theorems, demonstrations, or proofs as being "beautiful". One can, presumably, scientifically (or mathematically) construct a logically verifible program that is ugly (e.g., difficult to read or understand) as well as one that is beautiful (e.g., a pleasure to read or easy to understand).

(e) So, whether computer science is an art is probably not to be set alongside the question of whether it is a science or a branch of engineering.

26. Our (first order) definition is that computer science is the study of the theory and practice of programming computers. [Khalil and Levy, 1978, 31]

- A "study", not necessarily a science; possibly an art.

- Focus on programming; presumably, the "theory" includes the study of algorithms, and the "practice" includes executing the programs on computers.

27. Computer science as a branch of math:

- [Krantz, 1984] offers a sketch of an argument to the effect that computer science is (just) a branch of mathematics. He doesn't really say that; rather, his concern is that computer science as an academic discipline is young and unlikely to last as long as math: "Computer Science did not exist twenty-five years ago [i.e., in 1959]; will it exist twenty-five years from now? [I.e., in 2009]" [Krantz, 1984, 600]. Now, as an academic department in a university, computer science probably did not exist in 1959, although, arguably, it did two years later [Knuth, 972b, 722]; it certainly continued to exist, quite strongly, in 2009. So, if anything, computer science is becoming much stronger as an academic discipline, not weaker. (But consider John Ho's observations, made around the same time as Krantz's, as well as the history of microscopy.)

- The closest Krantz comes to making an argument is in this passage: "Computer scientists, physicists, and engineers frequently do not realize that the technical problems with which they struggle on a daiy basis are *mathematics*, pure and simple" [Krantz, 1984, 599]. As a premise for an argument to the conclusion that computer science is nothing but mathematics, this is obviously weak; after all, one could also conclude from it that physics and engineering are nothing but mathematics, a conclusion that I doubt Krantz would accept and that I am certain that no physcist or engineer would accept.

  But let's see if we can strengthen it a bit: Suppose, for the sake of argument, that all of the problems that a discipline *D*1 is concerned with come from discipline *D*2. Does it follow that *D*1 is nothing but *D*2? Here's a (perhaps weak) analogy, which I will strengthen in a moment: Suppose that you want to express some literary idea; you could write a short story, a novel, or a poem perhaps. Does it follow that prose fiction and poetry are the same thing? Probably not; rather, poetry and prose are two different ways of solving the same problem (in our example, the problem of expressing a certain literary idea). Similarly, even if both computer science and mathematics study the same problems, they do so in different ways: Mathematicians prove (declarative) theorems; computer scientists express their solutions algorithmically.

  So, the argument is probably invalid. Worse, its premise is most likely false: Surely, there are many problems dealt with in computer science that are not purely mathematical in nature. Even if we eliminate hardware issues (the purely computer engineering side of the discipline) from consideration, surely topics such as operating systems and artificial intelligence are not purely mathematical in nature.

28. CS as the study of information:

   - In addition to others noted above, [Denning, 1985, 16] defines it as "the body of knowledge dealing with the design, analysis, implementation, efficiency, and application of processes that transform information." Because of the long list of techniques ("design [etc.]"), I will assume that, by 'body of knowledge', he means the neutral "study" (rather than "science"). Moreover, he says that "Computer science includes in one discipline its own theory, experimental method, and engineering. This contrasts with most physical sciences.... ...I do not think the science and the engineering can be separated within computer science because of the fundamental emphasis on efficiency. But I do believe [in contrast with [Krantz, 1984], John Ho, et al.] that the discipline will endure, because the fundamental question underlying all of computer science—what can be automated?—will not soon be answered" [Denning, 1985, 19]. So, he seems to be on the side of sui generis. He takes the object to be "processes

that transform information". So, one could either take him as seeing algorithms (processes, procedures) as being the focus, or else information. Interestingly, he twice cites with great approval Forsythe's "What can be automated?" [Denning, 1985, 16, 19] without clearly stating its relation to "processes that tranform information".

- [Barwise, 1989, 386–387] says 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". Although he doesn't argue that computer science is the study of information or information processing, I think he might side with that characterization. His principal reason seems to be thet "the...view of computers as informational engines...makes sense of the battle for computational resources" and enables us to "think about them so as to make the best decisions about their acquisition and use". And why is that? One (abductive) reason is that this view enables us to understand the impact of computers along the same lines as we understand the impact of "books and printing [and] ...movable type.... [C]omputers are not just super calculators. They make available a new informational medium...just as with printing." This may seem obvious or not, but Barwise was writing in 1989, way before the advent of the World Wide Web, Kindles, or iPads, and his prediction certainly seems to be coming true. But why is information processing the key, rather than, say, symbol manipulation? Arguably, information processing is nothing but symbol manipulation. Arguably, however, it is *interpreted* symbol manipulation; moreover, not all symbol manipulation is necessarily information in some sense. So, perhaps, although computers are nothing but symbol manipulators, it is as information processors that they will have (are having) an impact.

29. Computer science is a branch of engineering:

- Just as with the claim that computer science is a science, there are two versions of this, a strong and a weak version: The weak version is that computer science is at least a branch of engineering, or that part of computer science is a branch of engineering. This allows for the possibility that computer science might be engineering together with being something else (such as a science) or that computer science can be divided into subdisciplines (such as a science subdiscipline and an engineering subdiscripline).

  The strong version is that computer science is nothing but engineering (and is not also a science).

- [Loui, 1987, 175] argues against [Krantz, 1984] that computer science is a legitimate academic discipline (not another discipline, such as math, and not something that will disappear or dissolve into other disciplines, like microscopy).

- And, apparently against people like [Newell et al., 1967], he argues that it is not the study of "the use of computers" [Loui, 1987, 175]. But that word 'use' is important. What Loui has in mind is certainly includes the study of hardware [Loui, 1987, 176]; what he is rejecting is that computer science, as an academic discipline, is the study of *how to use* computers (in the way that driving schools teach how to drive cars).

- His definition is this: "Computer science is the theory, design, and analysis of algorithms for processing [that is, for storing transforming, retrieving, and trasmitting] information, and the implementations of these algorithms in hardware and in software" [Loui, 1987, 176]. He thus seems to side with [Denning, 1985] and, perhaps, [Knuth, 1974b].

- He then goes on to argue that computer science is an engineering discipline [Loui, 1987, 176]:

  (a) Engineering...

     i. is concerned with what *can* exist (as opposed to what *does* exist.

     ii. "has a scientific basis"

     iii. applies "principles in design"

     iv. analyzes "trade-offs"

     v. has "heuristics and techniques"

  (b) "Computer science has all the[se] significant attributes of engineering"

  (c) ∴, computer science is a branch of engineering.

  We will delay investigation of the first premise to Ch. 5.

His justification that computer science is not concerned with what exists is related to the claim that others have made CITE that computer science is not a natural science, but a science of human-made artifacts. There are two possible objections to this: First, insofar as procedures are natural entities, computer science—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.

The "scientific basis" of "traditional engineering disciplines suchas mechanical engineering and electrical engineering" is physics, but the scientific basis of computer science is, according to Loui, mathematics. He thus considers computer science to be "a new kind of engineering". Mathematical engineering, perhaps? [Abrahams, 1987, 472] explicitly makes this claim, and [Halpern et al., 2001] can be read as making a case for computer science as being based more on logic than on mathematics, so—if it is a kind of engineering—perhaps it is logical engineering?. (We'll return to computer science as mathematical engineering below.) This assumes, of course, that you are willing to consider mathematics to be a natural science, or else that science is not limited to studying natural

entities. But in the latter case, why worry about whether computer science is concerned with what can, rather that what does, exist?

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, 176]. Hence, presumably, computer science is a branch of engineering. 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 computer science is mathematics, then why does Loui say that computer "specialists" apply apply "the priniciples of computation"? I would have expected him to say "the principles of mathematics". Perhaps he sees "computation" as being a branch of mathematics (but that's inconsistent with his objections to Krantz). Or perhaps he doesn't think that the abstract mathematical theory of computation is part of computer science (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.

"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, 177]. This is true, but doesn't support his argument as well as it might. For one thing, not only system designers evaluate such trade-offs; so do theoreticians—witness the abstract mathematical theory of complexity. And, as noted above, not all computer scientists design such systems. So, at most, those who do are doing a kind of engineering.

As for heuristics, Loui seems to have in mind rough-and-ready "rules of thumb" rather than formally precise theories. Insofar as engineers rely on such heuristics (which we will discuss in Ch. 5 when discuss Koen's definition), and insofar as some computer scientists also do, then those computer scientists are doing something that engineers also do. But so do many other people: Writers surely rely on such heuristics ("write simply and clearly"); does that make them engineers? This is probably his weakest argument.

- So, despite Loui's belief that he is arguing for the strong view of computer science as nothing but engineering, I think he's better interpreted as arguing for the weak view, namely, that there is an engineering component to the discipline. Indeed, towards the end of his essay, after approvingly citing [Denning, 1985]'s definition, he says: "It is impossible to define a reasonable boundary between the disciplines of computer science and computer engineering. They are the same discipline" [Loui, 1987, 178].

But that seems to contradict the title of his essay!

30. Computer science is the study of complexity [Ceruzzi, 1989, 268]

    - Ceruzzi 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, 1989, 269]). But even if computer science is part of a larger discipline that studies complexity, it doesn't follow that computer science 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]. But, again, at most this makes the claim that *part* of computer science is the study of complexity. And really all that it says is that programming is (the best) way to *avoid* complexity, not that it studies it. What's missing, 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. (And only [Khalil and Levy, 1978] makes that claim.)

    - And [Denning et al., 1989, 11] point out that viewing " 'computer science [as] the study of abstraction and the mastering of complexity'...also applies to physics, mathematics, or philosophy." But, for a contrasting view of abstraction as the key to computing, see [Kramer, 2007].

    - [Gal-Ezer and Harel, 1998, 80] also suggest complexity as a unifying theme, if not the basic subject matter, of computer science: "a beneficial way to view CS is by the kinds of complexity it deals with[:] computational complexity; system, or behavioral, complexity; and cognitive complexity."

31. Is computer science coextensive with (if not identical to) electrical (more precisely, electronic) engineering?

    - [Ceruzzi, 1989] doesn't explicitly say this, but he comes close: First, "Electronics emerged as the 'technology of choice' [over those that were used in mechanical calculators or even early electric-relay-based computers] for implementing the concept of a computing machine.... ...This activity led to the study of 'computing' independently of the technology out of which 'computers' were built. In other words, it led to the creation of a new science: 'Computer Science' " [Ceruzzi, 1989, 257]. Second, "As computer science matured, it repaid its debt to electronics by offering that engineering discipline a body of theory which served to unify it above the level of the physics of the devices themselves. In short, computer science provided electrical engineering a pardigm, which I call the 'digital approach,' which came to define the daily activities of electrical engineers in circuits and systems design" [Ceruzzi, 1989, 258].

- One problem with trying to conclude from this that computer science is (nothing but) electrical engineering is that there are now other technologies that are beginning to come into use: quantum computing [Grover, 1999, Aaronson, 2008, Monroe and Wineland, 2008, Aaronson, 2011, Hayes, 2014b] and DNA computing [Adleman, 1998, Shapiro and Benenson, 2006, Qian and Winfree, 2011]. Assuming that those methods achieve some success, then it becomes clear how computer science goes beyond any particular implementation technique or technology, and becomes a more abstract science (or study, or whatever) in its own right. And Ceruzzi himself declares, "The two did not become synonymous" [Ceruzzi, 1989, 273].

32. [Licklider and Taylor, 1968] fairly accurately predicted what computers would look like, and how they would be used, today (nearly 50 years in the future). What they were writing about is clearly part of "computer science", yet equally clearly not (directly) part of the abstract, mathematical theory of computation. This strongly suggests that it would be wrong to treat computer science as being primarily about algorithms or primarily about computers. It is about both. We'll see this more clearly when we trace the parallel histories of computers (as calculating machines) and computing (as it evolved from the search for a foundation for mathematics).

33. HARTMANIS 1993 SHOULD PROBABLY BE REQUIRED/VERY STRONGLY RECOMMENDED, BUT RE-READ HIS 3 PAPERS (1992 WITH LIN, 1993, AND 1995 TO SEE WHICH SHOULD BE REQUIRED, WHICH V.S.R, ETC.

I quoted this earlier (§3.3.2):

> [Computer science differs from the known sciences so deeply that it has to be viewed as a new species among the sciences. [Hartmanis, 1993, 1].

Let's consider this claim carefully. First, Hartmanis comes down on the side of computer science being a science: It is a "new species *among the sciences*". (A tiger is a different species from a chimpanzee "among the animals", but they are both animals.)

Second, 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 (pretty distant) or lions to tigers (pretty 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, 1]

This strongly suggests that computer science 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, 6], [Hartmanis, 1995a, 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 computer science 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 social sciences or mathematics, or else in a brand-new camp of its own, i.e., *sui generis*.

And immediately after his claim that he will not define computer science (see the motto to §3.4), he gives this characterization:

> At the same time [that is, despite his immediately preceding refusal to define it], 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 computer science 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, 5] (cf. [Hartmanis, 1995a, 10])

Although it may be "clear" to Hartmanis that information, an "abstract subject", is (one of) the "object[s] of study in computer science", he does not share his reasons for that clarity. Since, as we have seen, others seem to disagree that computer science is the study of information (others have said that it is the study of computers or the study of algorithms, for instance), it seems a bit unfair for Hartmanis not to defend his view. But he cashes out this promissory note in [Hartmanis, 1995a, 10], where he says that "what sets it [computer science] 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, 1995a, 8]. This makes computer science sound very much like mathematics (which is not unreasonable, given that it was this aspect of computer science that led Hartmanis to his ground-breaking work on computational complexity, an almost purely mathematical area of computer science)

But it's not just information that is the object of study; it's also information-processing machines, that is, computers. Computers, however, don't deal directly with information, because information is abstract, that is, non-physical. For one thing, this suggests that, insofar as computer science is a new species of non-physical science, it is not a species of social science: Despite its name,

the "social" sciences seem deal with pretty physical things: societies, people, speech, etc. (This, by the way, is controversial. After all, one of the main problems of philosophy is the problem of the relation of the mind (which seems to be non-physical and is studied by the social science of psychology) to the brain (which is clearly physical and is studied by the natural sciences of biology and neuroscience). And philosophers such as John Searle [Searle, 1995] have investigated the metaphysical nature of social institutions such as money, which seem to be neither purely abstract (many people cash a weekly paycheck and get real money) nor purely natural or physical (money wouldn't exist if people didn't exist). So, if computer science is a science, but is neither physical nor social, then perhaps it is more like mathematics, which is sometimes called a "formal" science (http://en.wikipedia.org/wiki/Formal_science).

For another thing, to say that computers don't deal directly with information, but only with *representations* of information suggests computer science has a split personality: Part of it deals directly with something abstract (information), and part of it deals with directly with something real but that is (merely?) a representation of that abstraction (hence dealing indirectly with information). Such real (physical?) representations are called "implementations"; we will look at that notion in more detail in Ch. 14 and at the relation of computers to the real world in Ch. 17.

Finally, although Hartmanis's description of the goal of computer science sounds like he is talking about AI (and he might very well be), another way to think about that goal is this: Historically, we have "endowed" calculating machines with the ability to do both simple and complex mathematics. What other abilities can we give to such machines? Or, phrased a bit differently, what can be automated— what can be computed? GIVE EITHER A FORWARD OR A BACKWARD CITATION TO FORSYTHE.

HARTMANIS HAS INTERESTING THINGS TO SAY ABOUT COMPLEXITY AND LEVELS OF ABSTRACTION ON PP.5–6. IF I DISCUSS THIS ELSEWHERE, I SHOULD CITE HARTMANIS.

> [Computer science is concentrating more on the *how* than the *what*, which is more the focal point of physical sciences. In general the *how* is associated with engineering, but computer science is not a subfield of engineering. [Hartmanis, 1993, 8]

Here we have another reason why computer science is not a physical science (and probably also why it is not a social science). But there are two ways to understand "how": Algorithms are the prime formal entity that codify *how* to accomplish some goal. But, as Hartmanis quickly notes, engineering is the prime discipline that is concerned with how to do things, how to build things (which we'll look at in more detail in Ch. 5). The first kind of "how" is mathematical

and abstract (indeed, it is *computational*!); the second is more physical. One way to see this as consistent with Hartmanis's description of the objects of study of computer science is to say that, insofar as computer science studies abstract information, it is concerned with how to process it (that is, it is concerned with algorithms), and, insofar as computer science studies computers, it is concerned with how to process representations (or implementations) of information (that is, it is concerned with the physical devices).

But that latter task would seem to be part of engineering (perhaps, historically, electrical engineering; perhaps, in the future, quantum-mechanical or bioinformatic engineering; certainly computer engineering!). So why does he say that "computer science is *not* a subfield of engineering"? In fact, he seems to regret this strong statement, for he next says that "the engineering in our field has different characterizations than the more classical practice of engineering" [Hartmanis, 1993, 8]: So, computer science certainly overlaps engineering, but, just as he claims that computer science is a new species of science, he also claims that "it is a new form of engineering" [Hartmanis, 1993, 9]. In fact, he calls it "[s]omewhat facetiously…the engineering of mathematics"; but he also says that "we should not try to draw a sharp line between computer science and engineering" [Hartmanis, 1993, 9].

To sum up so far, Hartmanis views computer science as a new species both of science and of engineering. This is due, in part, to his view that it has two different objects of study: an abstraction (information) as well as its implementations. But isn't it also reasonable to think that, perhaps, there are really two different (albeit new) disciplines, namely, a new kind of science and a new kind of engineering? If so, do they interact in some way more deeply and integratively than, say, chemistry and chemical engineering, so that it makes sense to say that "they" are really a single discipline? Hartmanis suggests two examples that show a two-way interaction between these two (halves of the same?) discipline: Alan Turing's interest in the mathematical nature of computation led to his development of real computers and AI; and John von Neumann's interest in building computers led to his theoretical development of the structure of computer architecture [Hartmanis, 1993, 10].

Hartmanis explicitly says that computer science *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.

HARTMANIS ALSO HAS A NICE CHARACTERIZATION OF THE DUAL HISTORY OF CS AS BOTH A CULMINATION OF THE HISTORY OF THE FOUNDATIONS OF MATH AND OF THE HISTORY OF CALCULATING DEVICES–SEE P.9

34.   One of the defining characteristics of computer science is the

> immense difference in scale of the phenomena computer science deals with. From the individual bits of programs and data in the computers to billions of operations per second to the highly complex operating systems and various languages in which the problems are described, the scale changes through many orders of magnitude. [Hartmanis, 1993, 5] (cf. [Hartmanis, 1995a, 10])

[Brooks, 1996, 62, col. 2] echoes this when he says that "It is this new world of complexity that is our peculiar domain." In what way is this a "defining characteristic"? Let's say, following Hartmanis, that computer science is the science of information processing and representation. Is it *also* the science of scale differences? Not quite; rather, it seems that scale differences are a feature of information and its processing and representation. So, insofar as there might be some other science that also studies information (just as biology, chemistry, and physics all study physical phenomena), then that science would also, ipso facto, study such scale differences. Indeed, Hartmanis himself, in his replies to commentaries on this paper [Hartmanis, 1995b, 60], says that physics also deals with scale differnences. However, it is interesting to note that Herbert Simon [Simon, 1962], [Simon, 1996b] also focused on the study of such complexity and the usefulness of abstraction in studying it (cf. [Hartmanis, 1993, 6], [Hartmanis, 1995a, 11]).

35. What is information? Answering this question is important, but would take us too far afield. Lots of work has been done on the nature of information and on the philosophy of information. See, especially, the writings of Floridi, cited in §1.4, as well as [Piccinini and Scarantino, 2011] and the commentary on [Hartmanis, 1995a] by Denning (1995), cited below in §3.14.

36. What Do Computer Scientists Do?

One way to answer this new question is to look at some of the "advertising" mentioned in §3.3.1.4. Here is a more or less random selection; some of these "definitions" of computer science were originally found on departmental websites or from introductory course materials:

(a) Because the word 'compute' comes from a Latin expression meaning "to reckon together", perhaps computer science is the "science of how to calculate" [Jacob, 1999]. As we will see, this is not entirely unreasonable, but a more plausible characterization is that computer science is the "science of processing information by computers" [Jacob, 1999].

But this raises further questions: What is a "science"? What is "information"? What does it mean to "process" information? And what is a "computer"?

(b) According to [Roberts, 2006, 5], computer science is "the science of problem solving in which the solutions happen to involve a computer."

As Alexander the Great knew when he sliced through the Gordian knot instead of untying it, there are at least two ways to solve problems.[32] Presumably, a "science" of problem solving—based on theories of algorithms, data structures, and complexity—is to be contrasted with mere "hacking" in the sense of "playing around". But one might also wonder how "solving problems" might be related to "calculating" or "processing information", and how the computer is supposed to help.

37.   Hartmanis calls computer science "a new species among the sciences" [[Hartmanis, 1995a]; but see also [Hartmanis, 1993, 1]].... I have argued that it would be more accurate to call computer science a new species of engineering [Loui, 1987]. [Loui, 1995, 31]

But his argument here [Loui, 1995, 31] is interestingly different from his earlier one:

(a) "The goal of engineering is the production of useful things"

(b) "[C]omputer science is concerned with producing useful things"

(c) ∴ Computer science is a species of engineering.

This is interesting, *because it is invalid!* Just because two things share a common property, it does not follow that one is subsumed under the other. For one thing, Loui could equally well have concluded that engineering was a species of computer science. But, more importantly, just because both cats and dogs are household pets, it doesn't follow that either is a species of the other.

But there are better arguments for considering computer science to be a branch of engineering, as we have seen (and will look at more closely in Ch. 5. If it is engineering, what kind of "new" engineering is it? Here's Loui's argument [Loui, 1995, 31] for that:

(a) "[E]ngineering disciplines have a scientific basis".

(b) "The scientific fundamentals of computer science...   are rooted...in mathematics."

(c) ∴ "Computer science is...a new kind of engineering."

This argument is interesting for better reasons than the first one. First, there are a few missing premises:

A Mathematics is a branch of science.

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

Recall that we can assume that computer science *is* a kind of engineering (if not from Loui's invalid argument, then, at least, from some of the better ones given by others). So, from that and 37a, we can infer that computer science (an an

---

[32]http://en.wikipedia.org/wiki/Gordian_Knot, accessed 20 September 2011.

engineering discipline) must have a scientific basis. We need premise 37A so that we can infer that the basis of computer science, which, by 37b is mathematics, is indeed a scientific one. (Alternatively, one can view 37b as the conjunction of "The fundamentals of computer science are rooted in mathematics" and 37A.) Then, from 37B, we can infer that computer science must differ from all other branches of engineering. It is, thus, mathematical engineering (as others have noted).

But this reminds me of the dialogue between [Newell et al., 1967] and [Knuth, 1974b]. Both Loui and Hartmanis agree that computer science is a new kind of something or other; each claims that the scientific/mathematical aspects of it are central; and each claims that the engineering/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.

38. Put this somewhere in §3.4:

> Let's remember that there is only one nature—the division into science nd engineering, and subdivision into phyiscs, 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, 56]

His last two sentences suggest that, rather that consider computer science to be either science or else engineering (or even sui generis), it should be thought of as an interdisciplinary discipline. As such, its goal would be to bring to bear what is special to it—namely, algorithms and computers—to problems in other disciplines. This is summarized in his final paragraph:

> I reject the title question ["Are We Scientist or Engineers?"]. …[Computer science] spans a multidimensional spectrum from deep and elegant mathematics to crafty programming, from abstraction to solder joints, form deep truth to elusive human factors, from scholars motivated purely by the desire for knowledge or practitioners making my everyday life better. It embraces the ethos of the scholar as well as that of the professional. To answer the question would be to exclude some portion of this spectrum, and I would be poorer for that. [Wulf, 1995, 57]

That would be a great quote to end either this chapter or the chapter on engineering (before going on to other questions).

39. More comments on [Brooks, 1996]; maybe some or all of this should go in Ch. 5?

Brooks's argument:

(a) "[A] science is concerned with the *discovery* of facts and laws." [Brooks, 1996, 61, col. 2]

(b) "[T]he scientist *build in order to study*; the engineer *studies in order to build*. [Brooks, 1996, 62, col. 1]

(c) The purpose of engineering is to build things.

(d) Computer scientists "are concerned with *making things*, be they computers, algorithms, or software systems. [Brooks, 1996, 62, col. 1]

(e) ∴ "the discipline we call 'computer science' is in fact not a science but a *synthetic*, an engineering, discipline." [Brooks, 1996, 62, col. 1]

As I say earlier, whether or not Brooks's notion of what science is is accurate will be our focus in Chapter 4. So, let's assume the truth of the first premise for the sake of the argument.

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

To sum up, the point of the second premise is to support the next premise, which defines engineering as a discipline whose goal is to build things, that is, a "synthetic"—as opposed to an "analytic"—discipline.[33] Again, whether or not Brook's notion of what engineering is is accurate will be our focus in Ch. 5. So, let's assume the truth of the second and third premises for the sake of the argument.

Clearly, if the third 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). So, is it the case that computer scientists are (only? principally?) concerned with building or "making things"? And, if so, what kind of things?

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. Here's why: He says that "Even when we build a computer the computer scientist designs only the abstract properties—its architecture and implementation.

---

[33]To analyze is to pull apart; to synthesize is to put together. "We speak of engineering as concerned with 'synthesis,' while science is concerned with 'analysis'." [Simon, 1996b, 4]

Electrical, mechanical, and refrigeration engineers design the realization" [Brooks, 1996, 62, col. 1]. I think this passage is a bit confused or sloppy, but it makes an interesting point: Brooks seems to be saying that computer scientists only design abstractions, and other (real?) engineerings implement them in reality. This is reminiscent of the distinction between the specifications of an algorithm and its detailed implementation in a computer program (we'll look into this in Ch. 10). Where Hartmanis and others have suggested that computer science is mathematical engineering, Brooks (following [Zemanek, 1971]) calls it "the engineering of abstract objects": If engineering is a discipline that builds, then what computer science qua engineering builds is implemented abstractions (see Ch. 14 for further discussion).

Moreover, what computer scientists build, unlike what other engineers build, are not things for direct human benefit (at least, not in 1977 when he first wrote these words and when very few people other than scientists, engineers, and business people had access to (gigantic) computing machines—certainly, there were no personal computers, laptops, tablets, or smartphones) but, rather, things that in turn can be used to build such directly beneficial things. Put more simply, his answer to the question "What is a computer?" seems to be: A computer is a tool (and a computer scientist, who makes such tools, is a "toolsmith" [Brooks, 1996, 62, col. 1].

But much of what he says against computer science being considered a science smacks of a different battle, one between science and engineering, with scientists belittling engineers. Brooks takes the opposite position: "as we honor the more mathematical, abstract, and 'scientific' parts of our subject more, and the practical parts less, we misdirect young and brilliant minds away from a body of challenging and important problems that are our peculiar domain, depriving the problems of the powerful attacks they deserve" [Brooks, 1996, 62, col. 2].

40. Notes on [Simon, 1996b]:

- Argument that computer science is a science of the artificial:

    (a) "A natural science is a body of knowledge about some class of things... in the world" [Simon, 1996b, 1] (Presumably, a natural science of *X* is a body of knowledge about *X*s in the world. Note that he does not say that the *X*s need to be "natural"!)

    (b) "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, 1]

    (c) "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, 2] (Again, this allows artifacts to be among the values of *X*. His justification for this premise consists of examples: the use of artificial heating and air-conditioning

to maintain temperature, "[a] forest may be a phenomenon of nature; a farm certainly is not. ...A plowed field is no more part of nature than an asphalted street—*and no less*", etc. [Simon, 1996b, 2–3, my emphasis]. All artifacts, he also notes, are subject to natural laws (gravity, etc.) [Simon, 1996b, 2].

Now, Simon doesn't, in this chapter, explicitly draw the conclusion that there can be sciences of artifacts or, more to the point, that—because computers are symbol systems [Simon, 1996b, 17ff] (cf. [Newell and Simon, 1976]) and symbols are "strings of artifacts" [Simon, 1996b, 2]—computer science is an "artificial science" of computers. For one thing, that's what his whole book is designed to argue. But he doesn't have to draw that conclusion explicitly: If it doesn't follow from the first premise that computer science can even be considered a natural science, it follows from these premises that any artifacts that impinge upon, or are produced by, nature or natural objects can be studied scientifically (in the manner of the second premise).

- It's almost as if he really wants to say that artificial sciences are natural sciences.

41.   Just as the scientific basis of electrical engineering is primarily physics, the scientific basis of software engineering is primarily computer science. This paper contrasts an education in a science with an education in an engineering discipline based on the same science. [Parnas, 1998, 2].

The first sentence, when set alongside the claims that computer science is mathematical engineering, suggest that software engineering is mathematical engineering engineering! So, both claims probably can't be true, and Parnas's second sentence shows why: He believes that computer science is a science, *not* an engineering discipline. Why?

Part of the reason concerns his definition of 'engineering': "Engineers are professionals whose education prepares them to use mathematics, science, and the technology of the day, to build products that are important to the safety and well-being of the public" [Parnas, 1998, 2]. As usual, we will defer analysis of this to Ch. 5. But note, first, the emphasis on education and, second, the idea that engineers *use* science to *build* things.

To complete his argument, he needs a premise that states that computer science is not engineering, that is, that computer science education doesn't prepare computer scientists to use computer science to build things or just that computer scientists don't build things. (That leaves open the possibility that computer science might be a branch of math or a "technology of the day", but it's pretty clear from the first quote that he thinks that it is a science.) This missing premise is the gist of his entire article. But at least one part of his argument is this: Proper training in software engineering ("designing, building, testing, and 'maintaining'

software produces" [Parnas, 1998, 2]) requires more than a course or two offered in a computer science curriculum; rather, it requires an "accredited professional programme. . . modelled on programmes in traditional engineering disciplines" [Parnas, 1998, 2].

But we still haven't got a clear statement as to why he thinks that computer science is a science and is not engineering. Well, maybe the latter: It's not engineering, because there is no "rigid accreditation process. . . [hence, no] well documented 'core body of knowledge'. . . for computer science" [Parnas, 1998, 2]. Such accreditation might be necessary, but is surely not sufficient: One might force such a core body of knowledge and such an accreditation process on, say, physics, but that wouldn't make physics an engineering discipline.

Some clarity arises here: "It is clear that two programmes are needed [for example, both physics and electrical engineering, or both computer science and software engineering], not because there are two areas of science involved [for example, physics and EE], but because there are two very different career paths. *One career path is that of graduates who will be designing products for others to use. The other career path is that of graduates who will be studying the phenomena that interest both groups and extending our knowledge in this area*" [Parnas, 1998, 3, my emphasis]. So: scientists study phenomena and extend knowledge; engineers design products. So: computer science studies phenomena and extends knowledge; software engineers design software products. (What phenomena does Parnas think that computer scientists study?) The distinction between science and engineering, for Parnas, is that between learning and building [Parnas, 1998, 4].

42. Yet another definition (in 3 different versions):

   (a) "Computer science [is] the science of solving problems with the aid of a computer" [Roberts, 1996]

   (b) "For the most part, computer science is best thought of as the science of problem solving in which the solutions happen to involve a computer." [Roberts, 2006, 5]

   (c) "Computer Science is the science of using computers to solve problems." [of Computer Science, 2003]

   Because these definitions don't limit the kind of problems being solved, they have the advantage of illuminating the generality and interdisciplinarity of computer science. And, because they implicitly include the software (algorithmic) side of computing—after all, you can't use a computer to solve a problem unless it has been appropriately programmed—they nicely merge the computer-vs.-algorithm aspects of the possible definitions. (They could just as easily have said something even more neutral: Computer science is the science of solving problems computationally, or algorithmically—after all, you can't solve

a problem that way without executing its algorithmic solution on a computer.) But can there really be a *science* of problem solving?

43. An interesting definition from an invalid argument:

> "Computer science is philosophy.   Logic is the foundation of philosophy.   It's also the foundation of Computer Science." [Rupp, 2003]

The first sentence is the title of Rupp's essay. The next two sentences are the first sentences of it, and the only support that he offers for his definition. But this is an invalid argument: Just because two disciplines share a common foundation, it does not follow that one of them "is" the other (and which one?) or that they are identical.

44. [Hammond, 2003],   quoting,   possibly,   Michael   R.   Fellows   (see http://en.wikiquote.org/wiki/Computer_science), says that computer science is not about computers, because "science is not about tools. It is about how we use them, and what we find out when we do" and because "Theoretical Computer Science doesn't even use computers, just pencil and paper". In other words, it is wrong to define a subject by its tools (see microscopy, above), and it is wrong to define a subject in such a way that it omits an important part of it.

   Rather, she says, "computer science is the study of algorithms, including their...Hardware Realizations".   In other words, although the "surrounding phenomena" of algorithms includes the computers that execute them, it's the algorithms that are the real focus.

   Even if this is true, it can be argued that a new tool can open up a new science or, at least, a new scientific paradigm (in the sense of [Kuhn, 1962]): "Paradigm shifts have often been preceded by 'a technological or conceptual invention that gave us a novel ability to see things that ocult not be seen before' " [Mertens, 2004, 196], quoting [Robertson, 2003]. Although computer science may not be *about* computers any more than astronomy is *about* telescopes, "The computer is to the naked mind what the telescope is to the naked eye, and it may well be that future generations will consider all precomputer science to be as primitive as pretelescopic astronomy" [Mertens, 2004, 196].

45. Another   science   of   an   artifact: bicycle   science [Wilson and Papadopoulos, 2004]. But it's really not clear if this is a science or a branch of engineering!

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

This isn't exactly a definition of computer science, but it could be turned into one: Computer science is the study of (choose one): (a) how to make computers

(at least) as "intelligent" as humans (how to get computers to do what humans can do); (b) 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.

Turing's motivation for the TM also supports this: The TM, as a model of computation, was motivated by how humans compute.

47. On Euclid (for this chapter, or elsewhere):
Euclid's geometry was procedural in nature; his proofs can be viewed as programs that construct certain geometrical figures using only compass and straightedge, much as computer programs construct (mathematical) objets using only recursive functions. [Toussaint, 1993] (see also [Goodman, 1987, §4])

48. On microscopy:
Computation is itself a methodology. [Denning, 2013, 37], citing unnamed others, uses the phrase "a new method of thought and discovery in science".

49. On computer science as a natural science:
[Denning, 2013, 37] argues that, insofar as computer science deals with information processes, it is a natural science, because many information processes are natural processes, e.g., in biology. (This supports Shapiro's argument.) [Easton, 2006] makes a similar argument.

Denning 2007 offers support for Shapiro's argument by citing 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. He concludes that "computing is *now* a natural science", not (or no longer?) "a science of the artificial".

An earlier statement by him to this effect is: "Computer science... is the science of information processes and their interactions with the world", together with the later comment that "There are many *natural* information processes" [Denning, 2005, 27, my emphasis].

50. Rather than wonder whether computer science is a science or not, or a branch of engineering or not, [Lindell, 2001] suggests that it is a liberal art. Today, when one thinks of the "liberal arts", one tends to think of the humanities rather than the sciences, but, classically, there were seven liberal arts: the linguistic liberal arts (grammar, rhetoric, and logic) and the mathematical liberal arts (arithmetic, music, geometry, and astronomy). (For more on this, see the *Wikipedia* article, "Liberal Arts Education" at https://en.wikipedia.org/wiki/Liberal_arts_education (accessed 1 July 2013).) Thought of this way, it becomes more reasonable to consider computer science as a modern version of these.

51. Minsky has also suggested that computer science is concerned with ways to "describe complicated processes" (SCS notes from Minsky's talk at UB, personal communication, 25 July 2005).

52.   Pedagogically, computer programming has the same relation to studying computer science as playing an instrument does to studying music or painting does to studying art. [Tucker et al., 2003, V]

53. Besides trying to say what CS is (a science/study/engineering ... of algorithms/computers/information ...), another approach is to say that it is a "way of thinking", that "algorithmic thinking" (about anything) is what makes CS unique. See [Anthes, 2006], especially the comments by Bernard Chazelle.

54. On the def of 'procedure' (such that computer science might be defined, à la Minsky and Shapiro, as a science of (computational) procedures:
    As [Bringsjord, 2006] argues, talk of procedures can be replaced by talk of "first-order logic, and proofs and interpretations"; consider, for example, the dual interpretations of statements of Prolog as either procedural or declarative statements, or Lisp programs, which appear to be merely definitions of recursive functions. So, we need to be clearer about what we mean by 'procedure' (as well as phrases like 'computational thinking' or 'algorithmic thinking'). This is a philosophical issue worthy of discussion (and we'll return to it in Ch. 7).

    On 'computational thinking':
    [Wing, 2006]'s notion of "computational thinking" may offer a compromise: This term avoids a procedural-declarative controversy, by including both concepts, as well as others. She defines computer science as "the study of computation—what can be computed and how to compute it". This definition 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. Computational thinking includes problem solving, recursive thinking, abstraction, and more.

    Denning 2009 finds fault with the notion, primarily on the grounds that it is too narrow: "Computation is present in nature even when scientist are not observing it or thinking about it. Computation is more fundamental than computational thinking. For this reason alone, computational thinking seems like an inadquate characterization of computer science" (p. 30). Note that, by 'computation', Denning means TM-computation. For his arguments about why it is "present in nature", see Denning 2007.

55. FOR SOME OF THE MORE MINOR POSITIONS, PERHAPS HAVE A SECTION AT THE END OF (EACH?) CHAPTER PRESENTING THE POSITIONS AND RAISING QUESTIONS FOR DISCUSSION ABOUT THEM. The next two items are prime candidates for this.

56. [Eden, 2007] seeks to bring clarity to the science-vs.-math-vs.-engineering controversy by taking up a distinction due to Peter Wegner [Wegner, 1976] among three different "Kuhnian paradigms" (see Ch. 4 for an explanation of "Kuhnian paradigms"): a view of computer science as (1) a "rationalist" or "mathematical" discipline, (2) a "technocratic" or "technological" discipline, and (3) a "scientific" discipline. (Tedre & Sutinen 2008 also discuss these three paradigms.) Eden then argues in favor of the scientific paradigm.

    But must there be a single paradigm? Are there no other disciplines with multiple paradigms? Does the existence of multiple paradigms mean that there is no unitary discipline of computer science? Or can all the paradigms co-exist?

57. Wing 2008 offers "Five Deep Questions in Computing":

    (a) $P = NP$?[34]

    (b) What is computable?

    (c) What is intelligence?

    (d) What is information?

    (e) (How) can we build complex systems simply?

    All but the last, it seems to me, concern abstract, scientific or mathematical issues: If we consider the second question to be the fundamental one, then the first can be rephrased as "What is efficiently computable?", and the second can be rephrased as "How much of (human) cognition is computable?". The fourth can then be seen as asking an ontological question about the nature of what it is that is computed ('0's and '1's? Information in some sense (and, if so, in which sense)?). The last question can be viewed in an abstract, scientific, mathematical way as asking a question about the structural nature of software (the issues concerning the "goto" statement and structural programming would fall under this category). But it can also be viewed as asking an engineering question: How should we—literally—build computers?

    If so, then her 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 be build devices to perform these computations?

    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).

58. [Lohr, 2008] almost inadvertently points out a confusion (among educators, in particular) about the nature of computer science. He quotes a high-school math

---

[34]See [Austin, 1983, Mycielski, 1983, Fortnow, 2009, Walsh, 2014] for brief, semi-formal discussions of what this means.

and computer science teacher as saying, "I do feel that computer science really helps students understand mathematics.... And I would use computers more in math, if I had access to a computer lab." The confusion is that computer science is seen as the use of a physical tool, rather than as the study of (as well as the use of) a method of thinking ("computational thinking"). For that, a computer is useful, but not necessary, as demonstrated by the "Computer Science Unplugged" project (http://csunplugged.org/ (accessed 1 July 2013)).

59. [Vardi, 2011] derives a "definition of computing, as an 'instrument for the human mind' " from Leibniz's statement that his *lingua characteristica universalis* (or universal formal language) and his *calculus ratiocinator* (or formal logic) will offer "mankind... a new instrument that will enhance the capabilities of the mind to a far greater extent than optical instruments strengthen the eyes". This is similar to Daniel C. Dennett's suggestion that the computer is a "prosthesis" for the mind (see, e.g., [Dennett, 1982]). Is that a reasonable definition of computer science?

60. [Cerf, 2012] says, not that computer science *is* a science, but that "there really is science to be found *in* computer science" (my emphasis). And, presumably, there is engineering to be found in it, and mathematics, too. This is a slightly different metaphor from the "spectrum".

61. For section on "CS is what computer scientists do":
    Need to add an organized list of what they do! On the science-engineering spectrum, we would have something like this:

    - abstract, mathematical theory of computation
    - abstract, mathematical theory of computational complexity
    - abstract, mathematical theory of program development
    - software engineering
    - …
    - operating systems
    - …
    - AI
    - …
    - computer architecture
    - …
    - VLSI, etc.

62. If "all true classification... [is] genealogical" [Darwin, 1872, Ch. 14, §"Classification", p. 437], then, to the extent that computer science is a single discipline, it is *both* mathematics *and* engineering. We'll see why in Ch. 6, when we explore the history of computers.

63. Relevant to the above: Computer science is a "portmanteau" discipline![35]

64. On complexity: One way in which computer science might be able to get a better handle on complexity than merely by scientific experimentation, analysis, and theorizing is by means of computational modeling, "taking into account the rules of interaction [of the parts of a complex system], the natures of the agents [that is, of the parts], and the way the agents, rules, and ultimately whole systems came about" [Adami, 2012, 1421]; cf. [Holland, 2012].

65. On interaction between the scientific and engineering aspects of computer science:
[McMillan, 2013] suggests that faster computers in the future will use a different kind of computation (such as "analog" computation). If so, this will be an engineering accomplishment. But, if so, it will also give rise to new mathematical analyses of this kind of "computation". Something similar, only in the opposite direction, is happening with quantum computing: mathematical theories of quantum computing are giving rise to physically implemented quantum computers. So, here, the science is giving rise to engineering. (We'll explore the nature of computation in Chs. 7 and 8, and the nature of different kinds of computing ("hypercomputation") in Ch. 11. On quantum computing, see [Hayes, 1995, Aaronson, 2008, Hayes, 2014b], [Grover, 1999]. For more on the digital-analog distinction, see [Montague, 1960], [Pour-El, 1974], [Haugeland, 1981], [Piccinini, 2004a], [Piccinini, 2007c], [Piccinini, 2008], [Piccinini, 2009], [Piccinini, 2010b], [Piccinini, 2011], [Piccinini and Craver, 2011], [Piccinini, 2012].)

66. On microscopy:
There used to be a British *Quarterly Journal of Microscopical Science* (1853–1965), affiliated with "the Microscopical Society of London". From its inaugural Preface:

> Recent improvements in the Microscope having rendered that instrument increasingly available for scientific research, and having created a large class of observers who devote themselves to whatever department of science may be investigated by its aid, it has been thought that the time is come when a Journal devoted entirely to objects connected with the use of the Microscope would contribute to the advancement of science, and secure the co-operation of all interested in its various applications.
>
> The object of this Journal will be the diffusion of information relating to all improvements in the construction of the Microscope, and to record the most recent and important researches made by its

---

[35]Inspired by the philosopher Hilary Putnam's comment that his "picture of [language] use was a portmanteau affair. There was the computer program in the brain, and there was the description of the external causes of the language user's words" [Putnam, 1994, 458]. A "portmanteau word" is one with "two meanings packed up into one word", like 'slithy' (meaning "lithe and slimy") or 'smog' (meaning "smoke and fog"); the term was coined by the great logician and storyteller Lewis Carroll [Carroll, 1871].

> aid in different departments of science, whether in this country or on
> the continent.. . .
>
> It is, perhaps, hardly necessary to apologise for the title of the
> Journal, as the term "Microscopical," however objectionable in its
> origin, has acquired a conventional meaning by its application to
> Societies having the cultivation of the use of the Microscope in view,
> and so fully expresses the objects of the Journal, that it immediately
> occurred as the best understood word to employ. It will undoubtedly
> be a Journal of Microscopy and Histology; but the first is a term but
> recently introduced into our language, and the last would give but a
> contracted view of the objects to which the Journal will be devoted.

One could replace 'microscope' with 'computer' (along with their cognates) and,
perhaps, 'histology' with something like 'mathematical calculations', and this
could read like a manifesto for the ACM.

The first issue included, besides many articles on what we now call microbiology,
a paper on "Hints on the Subject of Collecting Objects for Microscopical
Examination" and a review of a book titled *The Microscopist; or a Complete
Manual on the Use of the Microscope*.

Here is a passage from that review:

> As cutting with a sharp instrument is better than tearing with the nails,
> so vision with the microscope is better than with the naked eye. Its
> use is, therefore, as extensive as that of the organ which it assists, and
> it cannot be regarded as the property of one branch of science more
> than another. [Anonymous, 1853, 52]

And here is a paraphrase:

> As vision with the microscope is better than with the naked eye, so
> thinking with the computer is better than with the mind alone. Its use
> is, therefore, as extensive as that of the organ which it assists, and it
> cannot be regarded as the property of one branch of science more than
> another. [Anonymous, 1853, 52]

This is reminiscent of Daniel Dennett's arguments for the computer as a
"prosthesis" for the mind [Dennett, 1982].

In the March 1962 issue, the editors announced a change in focus, based on
the nature of many of their articles, to cytology, thus apparently changing their
interest from the tool to what can be studied with it. The change officially
occurred in 1966, when the journal changed its name to the *Journal of Cell
Science* (and restarted its volume numbers at 1).

67. [Denning, 2003, 15] makes an offhand comment that has an interesting implication. He says, "Computer science was born in the mid-1940s with the construction of the first electronic computers." This is no doubt true. But it suggests that that the answer to the question of what computer science is has to be that it is the study of computers. The study of algorithms is much older, of course, dating back at least to Turing's 1936 formalization of the notion, if not back to Euclid's geometry or ancient Babylonian mathematics [Knuth, 972a]. Yet the study of algorithms is clearly part of modern computer science. So, modern computer science is the result of a merger (a marriage) between the engineering problem of building better and better automatic calculating devices (itself an ancient endeavor) and mathematical problem of understanding the nature of algorithmic computation. And that implies that modern computer science is both engineering and science. (And that can be in two ways: that there are separate branches of computer science, or that they blend into each other.)

68. Question for end of chapter to think about:
In §3.11, I said that it makes no—or very little—sense to have a program without a computer to run it on. But the some of the earliest AI programs (for playing chess) were executed by hand (CITE Turing, Shannon, NewellShawSimon, IJGood). So, did these programs "have a computer to run on"? (Were the humans, who hand-executed them, the "computers" that these programs "ran on"?) When you debug a computer program, do you do the debugging by hand?[36]

69. On Knuth's ideas about teaching computers:

    (a) One potential restriction on Knuth's theory that we teach computers how to do something (more specifically, that, insofar as computer science is the study of what tasks are computable, it is the study of what tasks are teachable) is that teaching is propositional, hence *explicit* or conscious. But one way of getting a computer to do something is by *training* it (via a connectionist algorithm, or via a machine-learning algorithm—note that 'learning', in this sense of 'machine learning', is more than being (propositionally) *taught*). And such training is *implicit* or *un*conscious. We, as external, third-person observers still don't know *how* to do it consciously or explicitly. Knowing *how* is not necessarily the same as knowing *that*.

    (b)   It is a commonplace that a computer can do anything for which precise and unambiguous instructions can be given. [Mahoney and Haigh (ed.), 2011, 80].

        i. That is, a computer can do anything for which an *algorithm* can be given (for, after all, isn't an algorithm merely "precise and unambiguous instructions"?). Thought of this way, the comment is almost trivial.

---

[36]Thanks to Stuart C. Shapiro for this suggestion.

ii. But consider that to give such instructions (to give an algorithm) is to be able to explicitly *teach* the computer (or the executor, more generally) how to do that thing.

iii. And such explicit teaching is to make the task completely "conscious" (to make it a "System 2" task [Kahneman, 2011]). An algorithm in the form of an artificial neural network is akin to building in to the computer the ability, as if "innate", to do that thing. Such a computer could not tell us how it was doing it; it would have no "conscious" access to its algorithm. An algorithm in the form of an explicit machine-learning program that would enable the computer to learn how to do that thing is somewhere in the middle; it would be conscious of its ability to learn, but not necessarily of how to do the thing; it might not necessarily be able to teach someone or something else how to do it, unless it could observe itself doing it and develop a theory of how to do it (which theory would be expressed in an algorithm).

iv. On what is computable: Something is computable just in case "precise and unambiguous instructions can be given" for it. So, the question becomes: What tasks are amenable to "precise and unambiguous instructions"? Does this mean that chess is computable but vision is not? After all, one cannot give "precise and unambiguous instructions" that would enable someone to see. So there seems to be another notion of computability: A behavior is computable if it can be *described* in terms of such instructions". The entity that exhibits that behavior naturally might not use, or be able to use, those instructions in order to behave that way. But we might be able to give those instructions to another system that *could* use them to exhibit that behavior. So, for instance, the human brain might not literally compute in the sense of executing an algorithm im order to see, but a computer using that algorithm might be able to exhibit visuall behavior. (Whether it "sees", phenomenologically, is a philosophical question!) Similarly, the solar system might not be executing Kepler's laws, but an artificial solar system might.

70. [Knuth, 1985, 170–171] more emphatically defines computer science as "primarily the study of algorithms" *simpliciter*, primarily 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 suggests 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, 172]. We will see what it means to "think algorithmically" in Ch. 7.

71. CS as science of procedures:
Kay argues that computational linguistics is the branch of linguistics that studies languages computationally, that is, that tries to devise a computational theory

of the nature of language (and is distinct from "natural language processing", or the use of computers to process linguistic data. He considers the former to be scientific and the latter to be engineering, where "scientists try to understand their subject mainly for its own sake, though they are gratified when some of what they do proves useful. Engineers seek only the understanding needed to reach practical ends" [Kay, 2010, 1]. "[C]omputational linguists...look to computer science for insight into their problems. If communication is...about building structures by remote control in the heads of others, then it is all about *process*, and **computer science is the science of process, conceived in its most fundamental and abstract way**" [Kay, 2010, 2; italics in original; my boldface].

72. There can be algorithmic, that is, computational, theories (or models) of biological phenomena such as cells. See [Covert, 2014]. Or computational theories of plants; see [Pollan, 2013, 104–105].

73. Turing and von Neumann not only played leading roles in the design and construction of the first working computers, but were also largely responsible for laying out the general logical foundations for understanding the computation process, for developing computing formalisms, and for initiating the methodology of programming: in short, for founding computer science as we now know it. [Robinson, 1994, 5]

74. The ascendancy of logical abstraction over concrete realizaion 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.

   Of course no one should underestimate the enormous importance of the role of engineering in the history of the computer. Turing and von Neumann did not. They themselves had a deep and quite expert interest in the very engineering details from which they were abstracting, but they knew that the logical role of computer science is best played in a separate theater. [Robinson, 1994, 12]

75. Just as philosophy can be about anything (for any *x*, there is a philosophy of *x*), so computer science can be about pretty much anything. This can happen in two ways: For any *x*, there can be a computational theory of *x*. (Come to think of it, there are other disciplines like this, such as history.) But one can also use computational methods to study pretty much any *x* (see comments by the computer scientist Mehran Sahami in [Reese, 2014a]).

76. On "computational thinking":

> The ability to program or to understand different aspects of computing systems really involves clarity of thought. If you're going to program a computer, you need to specify a set of instructions for the computer to execute, and those have to be sufficiently clear to get your problem solved. Otherwise, you have errors. Just having the clarity of thinking to lay out instructions is an analytical skill that's useful for everyone. (Mehran Sahami, cited in [Reese, 2014a].)

> Algorithmic thinking is an interpretation of the world in which a person understands and formulates actions in terms of step-by-step procedures that give unambiguous results when carried out by anyone (or by a suitable machine). It resembles scientific thinking, which seeks to invent standard ways of observing that allow anyone to see and reproduce physical effects. Algorithmic thinking emphasizes the standard procedure[,] and scientific thinking the standard observer. [Denning, 1999, 3]

77. Another good source on complexity is [Pylyshyn, 1992]. It is also a good source on computer science as the science of information processing.

78. On "computational thinking":
    An idea due to Alan Perlis:

> ...procedural literacy—the ability to read and write processes—...a fundamental skill that everyone should learn at the university, regardless of field of study. ... It is not fluency with computer technology per se but rather is about how one can rationally navigate the tower of abstraction that exists within any computation system.
>
> The purpose of a course in programming is to teach people how to construct and analyze processes.
>
> [Wheeler, 2013, 296], quoting [Perlis, 1962, 209]

   Also:

> A course in programming..., if it is taught properly, is concerned with abstraction: the abstraction of constructing, analyzing, and describing processes.
> I personally feel that the ability to analyze and construct processes is av ery important ability, one which the student has to acquire sooner or later. I believe that he [sic] does acquire it in a rather diluted way during four years of an engineering or science program. I consider it also important to a liberal arts program.
> 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 ths 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 abilites better than any other course. [Perlis, 1962, 210]

And:

> Programming is the art of stating procdures. (John McCarthy, in [Perlis, 1962, 211])

79. Taught right, programming... is the hands-on, inquiry-based way that students learn what we call "computational thinking": probglem analysis and decomposition, abstraction, algorithmic thinking, algorithmic expression, stepwise fault isolation (we call it "debugging") and modeling. [Lazowska, 2014, A26]

80. Computer science is actually a study of things that happen. It's not only a study of a basic theory, and it is not just the business of making things happen. It's actually a study of how things happen. So the advice [to students] is: Don't lose the link ["between practice and theory"]! [Frenkel, 1993, 97, quoting Robin Milner]

81. The primary subject matter of computer science is *computational processes*, or processes that can be specified and studied independently of any physical realization. Processes, whether computational or not, are composed of interactions among objects, so we can also characterize the subject matter of computer science as interaction patterns. [Colburn and Shute, 2011, 249]

The main claim is stated without argument. The "or" clause is a bit hard to interpret. On one possible reading, it is an elaboration of the main claim. But surely being "independent of any physical realization" is not the only feature of being "computational". For instance, the halting problem is not computational, yet is independent of any physical realization. And surely some processes (a better term, of course, would be 'procedures'), perhaps recipes (as we will discuss in Ch. 10), would also be counterexamples. Another possible reading is that computer science has *two* kinds of processes as subject matter: computational ones and ones that are realization independent. But that leaves open the question of which ones are "computational". It is also a bit odd to say that computer science should *not* be interested in the realization of such processes (or is that more properly the concern of computer *engineering*?); we'll return to this topic in Ch. 14.

82. On complexity and (software) engineering:

> [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 elelments 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, 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 the data structures of '0' and '1') 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.

83.　　　Computer science offers a standard way to handle complexity: hierarchical structure. [Lamport, 2012, 16]

84. The following material might go here, or else in Ch. 4 (or both):

   As we mentioned in Ch. 2, [McGinn, 2015] argues that philosophy is a science just like physics (which is an empirical science) or mathematics (which is a "formal" science), likening it more to the latter than the former. To make his argument, he has to come up with a characterization of science, which we will look at in more details in Ch. 4. But an interesting exercise for the reader is to take that characterization, and see whether computer science fits it. [McGinn, 2015, 85] thinks that it is, like math, a formal science.

85. If not used in this chapter, then put in Ch. 4:

   > As we enter life, we all struggle to understand the world. Some of us continue this struggle with *dogged* determination. These are the *scientists*. Some of them realize that computation provides a privileged perspective to understand the world outside and the one within. These are the *computer scientists*. [Micali, 2015, 52].

86. On carving nature at its joints §3.4:
   (There may be a better place to put this observation, perhaps in a section on what a computer is or what thinking is.) When we define or formalize something, we often find that our definition or formalization includes something that was not previously thought to fall under the concept or excludes something that was thought to fall under it. We can then either reformulate the definition or bite the bullet about the inclusions and exclusions.

   Here is an observation about this point:

   > In mathematics, tentative proofs are sometimes 'refuted' not by logical contradiction..., but instead by a logically-sound consequence...that is unexpected and unwanted.... Lakatos [1976] argues that in such situations mathematicians should not just implicitly reject the unwelcome result ('monster barring'), but

> should instead either change their acceptance of the result ('concept stretching'...), or change the statement of the conjecture either by explicitly incorporating a condition that disallows the unwanted consequence ('lemma incorporation'...), or by inventing a wholly new conjecture. [Staples, 2015, §3.1]

87. On the first occurrence (here or earlier) of 'procedure', note that this is a term of art that is being used relatively informally here but that will be studied in more detail in later chapters, especially Ch. 7.

88. compare and contrast "computer science" with "library science", political science? [Mahoney and Haigh (ed.), 2011, 195] has a relevant point to make here: Is wondering whether computer science is a science, given its name,

> ...laboring under a misapprehension rooted in the English use of "computer science" to denote a subject other languages refer to as "informatics."

But he then goes on to point out that textbooks and courses from "informatics" departments cover exactly the same material as textbooks and courses from "computer science" departments.

> So I must wonder, as Shakespeare's Juliet once did, "What's in a name?" Here too the rose smells the same.

89. [Mahoney and Haigh (ed.), 2011, 26; my italics] suggests the spectrum view of the nature of computer science, given its historical origins in both mathematics and engineering:

> Between the mathematics that makes the device theoretically possible and the electronics that makes it practically feasible lies the programming that makes it intellectually, economically, and socially useful. *Unlike the extremes, the middle* remains a craft, technical rather than technological, mathematical only in appearance.

90. > Computing has no nature. It is what it is because people have made it so. [Mahoney and Haigh (ed.), 2011, 109]

This is not exactly of the form "computer science is what computer scientists do", though it bears a superficial resemblance. But I think that it is more sophisticated. Mahoney's point is that, unlike the other natural sciences (for example, physics, chemistry, biology, and so on), computer science only came into existence when its two histories (logical-mathematical and engineering) began to intersect in the 1940s, so its "nature" only came to be what those logicians, mathematicians, and engineers were doing.

91. [Mahoney and Haigh (ed.), 2011, 159ff] cites the argument that computer science is not a natural science because "the computer is an artifact, not a natural phenomenon, and science is about natural phenomena" (p. 159). This goes straight against [Newell et al., 1967], though it is consistent with Simon's

claim that computer science is a "science of the artificial". It is also orthogonal to [Shapiro, 2001]'s claim that computer science *is* a natural science, because Shapiro's argument is based on a different premise, namely, that computer science is the study of procedures, and procedures *are* natural phenomena.

But does not subscribe to this viewpoint, not because he thinks that computers *are* natural phenomena, but because he thinks that there is no sharp dividing line "between nature and artifact" (p. 159). And he thinks that that is the case because we *use* artifacts to study nature (p. 159)—"[w]e know about nature through the models we buid of it" (p. 161) and because "[a]rtifacts work by the laws of nature, and by working reveal those laws" (p. 160). In other words, artifacts are part of nature.

He goes on to suggest that nature itself might be ultimately computational in nature (so to speak). We will explore some of these ideas in Ch. 9 when we consider whether the universe might be a computer.

92.   Academically and professionally, computer engineering took charge
        of the hardware, while computer science concerned itself with the
        software.... [Mahoney and Haigh (ed.), 2011, 108]

To the extent, then, that software and hardware can*not* easily be distinguished (which we will begin to investigate in Ch. 12), then neither can computer engineering and computer science.

93.   Computation is sure to become one of the most important *of the*
        sciences. This is because it is the science of how machines can be
        made to carry out intellectual processes. [McCarthy, 1963, 1, my
        italics]

First, note that he thinks that it is a science, presumably just like other sciences (else it wouldn't be destined "to become one of the most important of" them). Second, the nature of this science is akin to the view that computer science is the study of what is computable: "Machines can be made to carry out intellectual processes" if those processes are computable. Why "intellectual processes"? Well, surely mathematical processes are intellectual, and to the extent that other intellectual processes are expressible mathematically—for example, by being codable into symbolic notation (and, ultimately, into binary notation)—those other intellectual processes are mathematical, hence potentially computable. Why is it a science? McCarthy doesn't say, but I would guess that it is a science in the same sense that mathematics is.

3.14 Further Sources of Information

Items in **boldface** are especially interesting.

   NOTE FOR NEXT DRAFT:
FACTOR OUT REFS TO: DNA, MAGIC, & QUANTUM COMPUTING (ETC.) IN SEPARATE SECTION.

1. Books and Essays:

   - Abrahams, Paul (1987), "What Is Computer Science?", *Communications of the ACM* 30(6) (June): 472–473.

   - Abrahams, Paul W.; & Lee, Graham (2013), "Computer Science Is Not a Science" (two separate letters to the editor), *Communications of the ACM* 56(1) (January): 8.

   - Anthes, Gary (2006), "Computer Science Looks for a Remake", *Computerworld* (1 May).

   - Bornat, Richard (2006), "Is 'Computer Science' Science?", *4th European Conference on Computing and Philosophy (ECAP'06, 22–24 June, Trondheim, Norway).*

   - Carey, Kevin (2010), "Decoding the Value of Computer Science", *Chronicle of Higher Education* 58(12) (12 November): A88.

   - Cerf, Vinton G. (2012), "Where Is the Science in Computer Science?", *Communications of the ACM* 55(10) (October): 5.

     – Argues that, even if computer science might once have focused on computing machines, it should now be more focused on "predict[ing] likely outcomes based on models[, which] is fundamental to the most central notions of the scientific method".

   - The College Board (2012), "AP Computer Science: Principles; Course Annotations", online at:
     http://www.collegeboard.com/prod_downloads/computerscience/1_Annotations-08-21-12_ld.pdf (accessed 8 July 2013)

   - Cook, Stephen A. (1983), "An Overview of Computational Complexity", *Communications of the ACM* 26(6) (June): 400–408.

   - Crowcroft, Jon (2005), "On the Nature of Computing", *Communications of the ACM* 48(2) (February): 19–20.

     – Argues that computer science is not the study of either the natural or the artificial but of the *virtual*.
     – Another essay on computer science and "the virtual" is [Pylyshyn, 1992].

   - CSTA [Computer Science Teachers Association] Curriculum Improvement Task Force (2005), "The New Educational Imperative: Improving High School Computer Science Education" (New York: Association for Computing Machinery); online at:
     http://csta.acm.org/Communications/sub/DocsPresentationFiles/White_Paper07_06.pdf (accessed 1 July 2013).

     – Relevant to this chapter, see §1.1 (pp. 17–19), "What Is Computer Science Anyway?".

   - Denning, Peter J. (1980), "What Is Experimental Computer Science?", *Communications of the ACM* 23(10) (October): 543–544.

- Denning, Peter J. (1995), "Can There Be a Science of Information?", *ACM Computing Surveys* 27(1) (March): 23–25.

- Denning, Peter J. (1999), "Computer Science: The Discipline", in Anthony Ralston & David Hemmindinger (eds.) (2000), *Encyclopedia of Computer Science*: PAGES.

- Denning, Peter J. (2005), "Is Computer Science Science?", *Communications of the ACM* 48(4) (April): 27–31.

- Denning, Peter J. (2007), "Computing Is a Natural Science", *Communications of the ACM* 50(7) (July): 13–18.

- Denning, Peter J. (2009), "Beyond Computational Thinking", Communications of the ACM 52(6) (June): 28–30.

- **Denning, Peter J.; Comer, Douglas E.; Gries, David; Mulder, Michael C.; Tucker, Allen; Turner, A. Joe; & Young, Paul R. (1989), "Computing as a Discipline",** *Communications of the ACM* **32(1) (January): 9–23.**

- Denning, Peter J.; & Rosenbloom, Paul S. (2009), "Computing: The Fourth Great Domain of Science", *Communications of the ACM* 52(9) (September): 27–29.

- **Denning, Peter J.; & Freeman, Peter A. (2009), "Computing's Paradigm",** *Communications of the ACM* **52(12) (December): 28–30.**

- Dijkstra, E.W. (1987), "Mathematicians and Computing Scientists: The Cultural Gap", *Abacus* 4(4): 26–31.

- Easton, Thomas A. (2006), "Beyond the Algorithmization of the Sciences", *Communications of the ACM* 49(5) (May): 31–33.

- Ensmenger, Nathan (2011), "Computing as Science and Practice", *Science* 333 (9 September): 1383.

- Fant, Karl M. (1993), "A Critical Review of the Notion of Algorithm in Computer Science", *Proceedings of the 1993 ACM Annual Computer Science Conference (Indianapolis)* (New York: ACM): 1–6;
  http://tinyurl.com/qdaojox[37] (accessed 6 December 2013).

- Feitelson, Dror G. (guest ed.) (2007), Special Issue on Experimental Computer Science, *Communications of the ACM* 50(11) (November): 24–59.

- Forsythe, George F. (1967), "A University's Educational Program in Computer Science", *Communications of the ACM* 10(1) (January): 3–8.

- Green, Penelope (2014), "Magic in the Mundane", *New York Times* (17 July): D1, D4.

  – Accessed 26 July 2014 from:
    http://www.nytimes.com/2014/07/17/garden/putting-magic-in-the-mundane.html
  – "An inventor of 'enchanted objects' [embedded with computers] sees a future where we can all live like wizards."

- Guzdial, Mark (2008), "Paving the Way for Computational Thinking", *Communications of the ACM* 51(8) (August): 25–27.

- Hartmanis, Juris (1995), "On Computational Complexity and the Nature of Computer Science", *ACM Computing Surveys* 27(1) (March): 7–15.

---

[37]http://staffweb.worc.ac.uk/DrC/Courses%202006-7/Comp%204070/Reading%20Materials/FAnt.pdf

– This paper covers much of the same ground, and in many of the same words, as [Hartmanis, 1993], but it is more easily accessible.

– Among several follow-up commentaries, see esp.:

∗ Denning, Peter J. (1995), "Can There Be a Science of Information?", *ACM Computing Surveys* 27(1) (March): 23–25.

∗ Loui, Michael C. (1995), "Computer Science Is a New Engineering Discipline", *ACM Computing Surveys* 27(1) (March): 31–32.

∗ Plaice, John (1995), "Computer Science Is an Experimental Science", *ACM Computing Surveys* 27(1) (March): 33.

∗ Stewart, N.F. (1995), "Science and Computer Science", *ACM Computing Surveys* 27(1) (March): 39–41; a longer version is online at: http://www.iro.umontreal.ca/∼stewart/science_computerscience.pdf (accessed 5 June 2013).

∗ **Wulf, W.A. (1995), "Are We Scientists or Engineers?",** *ACM Computing Surveys* **27(1) (March): 55–57.**

– And Hartmanis's replies:

∗ Hartmanis, Juris (1995), "Response to the Essays 'On Computational Complexity and the Nature of Computer Science' ", *ACM Computing Surveys* 27(1) (March): 59–61.

• Hayes, Brian (2015), "Cultures of Code", *American Scientist* 103(1) (January-February): 10–13, http://www.americanscientist.org/issues/pub/cultures-of-code (accessed 24 March 2015)

– Distinguishes "three communities in the world of computation": computer science, computational science, and software development.

• Hendler, James; Shadbolt, Nigel; Hall, Wendy; Berners-Lee, Tim; & Weitzner, Daniel (2008), "Web Science: An Interdisciplinary Approach to Understanding the Web", *Communications of the ACM* 51(7) (July): 60–69.

• Hsu, Francis (2013), "Is Computing Science?", *Communications of the ACM* 56(8) (August): 9.

– Argues that "there are no clear boundaries" between branches of knowledge.

• Kroes, Peter (2010), "Engineering and the Dual Nature of Technical Artefacts", *Cambridge Journal of Economics* 34: 51–62.

– Argues that "technical artefacts are physical structures with functional properties"

• Kukla, A. (1989), "Is AI an Empirical Science?", *Analysis* 49: 56–60.

– The answer is: No; it is an *a priori* science or discipline like mathematics.

– For the opposite point of view, see:
Burkholder, Leslie (1999), "Are AI and Mechanics Empirical Disciplines?", *Journal of Experimental and Theoretical Artifical Intelligence* 11: 497–500.

• Lloyd, Seth (1990), "The Calculus of Intricacy: Can the Complexity of a Forest Be Compared with That of *Finnegans Wake*?", *The Sciences* (September/October): 38–44.

– An informal discussion of the nature of complexity and how it can be measured using information theory or ideas from computing.

- Lohr, Steve (2006), "Computing 2016: What *Won't* Be Possible?", *New York Times* (31 October): F3.

- Naur, Peter (1995), "Computing as Science", Appendix 2 of *An Anatomy of Human Mental Life* (Gentofte, Denmark: naur.com Publishing): 208–217; online at http://www.naur.com/naurAnat-net.pdf (accessed 1 July 2013) via http://www.naur.com/Nauranat-ref.html (accessed 1 July 2013).

- Parnas, David Lorge (1998), "Software Engineering Programmes Are Not Computer Science Programmes", *Annals of Software Engineering*.

  – See also:
    Dewar, Robert; & Astrachan, Owen (2009), "CS Education in the U.S.: Heading in the Wrong Direction?", Point/Counterpoint, *Communications of the ACM* 52(7) (July): 41–45.

- Pylyshyn, Zenon W. (1987), "Computers and the Symbolization of Knowledge", in Ralph Morelli, W. Miller Brown, Dina Anselmi, Karl Haberlandt, & Dan Lloyd (eds.), *Minds, Brains and Computers: Perspectives in Cognitive Science and Artificial Intelligence* (Norwood, NJ: Ablex, 1992): 82–94; portions online at: http://tinyurl.com/Morelli-etal1987 (accessed 19 July 2013); preprint online at: http://ruccs.rutgers.edu/ftp/pub/papers/suffolk.pdf (accessed 19 July 2013).

- Shneiderman, Ben (2007), "Web Science: A Provocative Invitation to Computer Science", *Communications of the ACM* 50(6) (June): 25–27.

- Simon, Herbert A. (1996), "Understanding the Natural and Artificial Worlds", Ch. 1, pp. 1–24 of Simon's *The Sciences of the Artificial, Third Edition* (Cambridge, MA: MIT Press).

  – Originally written in 1969; updated in 1981.
  – In this chapter, Simon also has interesting things to say about whether computer programs are scientific theories, which we will investigate in Ch. 15

- Tedre, Matti (2015), *The Science of Computing: Shaping a Discipline* (Boca Raton, FL: CRC Press).

  – An investigation of the philosophical issues around the nature and history of computer science, examining whether it is a science, and, if so, what kind of science it might be.

- Tedre, Matti; & Moisseinen, Nella (2014), "Experiments in Computing: A Survey", *Scientific World Journal*, Vol. 2014, Article ID 549398, http://dx.doi.org/10.1155/2014/549398 (accessed 3 April 2015).

  – A survey of the nature of experiments in science, and whether computer science is experimental in nature.

- Turner, Ray (2010), "Programming Languages as Mathematical Theories", in Jordi Vallverdú (ed.), *Thinking Machines and the Philosophy of Computer Science: Concepts and Principles* (IGI Global): 66–82.

- Vardi, Moshe Y. (2014), "Boolean Satisfiability: Theory and Engineering", *Communications of the ACM* 57(3) (March): 5

  – Accessed 14 March 2014 from:
    http://cacm.acm.org/magazines/2014/3/172516-boolean-satisfiability/fulltext
  – An editorial on the difference between computer *science* approaches to an important theoretical problem and computer *engineering* approaches to a practical version of the same problem.

- Wing, Jeannette M. (2006), "Computational Thinking", *Communications of the ACM* 49(3) (March): 33–35.

  – For a contrasting view, see:
  Denning, Peter J. (2009), "Beyond Computational Thinking", *Communications of the ACM* 52(6): 28–30.

- Wing, Jeannette M. (2008), "Five Deep Questions in Computing", *Communications of the ACM* 51(1) (January): 58–60.

2. Websites:

- Chazelle, Bernard (2006), "The Algorithm: Idiom of Modern Science"
  http://www.cs.princeton.edu/∼chazelle/pubs/algorithm.html
  accessed 5 November 2012.

- Denning, Peter J., "Great Principles of Computing",
  http://cs.gmu.edu/cne/pjd/GP/GP-site/welcome.html (accessed 8 July 2013)

- Denning, Peter J. (ed.), Symposium on "The Science in Computer Science", *Ubiquity*, http://ubiquity.acm.org/symposia.cfm
  (accessed 28 June 2013)

- Department of Computing Sciences, Elon University, Elon, NC (2007), "What Is Computer Science?"
  http://www.elon.edu/e-
  web/academics/elon_college/computing_sciences/curriculum/cs.xhtml
  accessed 5 November 2012.

  – This is from the departmental website of a small college's "computing sciences" (note the name!) department. It discusses pretty much all the issues we've been looking at: Is it a science? What does it study? Is it an engineering discipline?

- Shapiro, Stuart C. (2001), "Computer Science"
  http://www.cse.buffalo.edu/∼shapiro/Courses/CSE115/notes2.html
  accessed 5 November 2012.

- Stairs, Allen (2014), Response to question about the definition of 'magic',
  http://www.askphilosophers.org/question/5735
  (accessed 5 December 2014).

- Tedre, Matti (2007), "Lecture Notes in the Philosophy of Computer Science", esp.
  "Introduction to the Course"
  (http://cs.joensuu.fi/∼mmeri/teaching/2006/philcs/files/lecture_notes1.pdf),
  "Part I: Struggling for Status"
  (http://cs.joensuu.fi/∼mmeri/teaching/2006/philcs/files/lecture_notes2.pdf),
  and "Part II: Emerging Interdisciplinarity"
  (http://cs.joensuu.fi/∼mmeri/teaching/2006/philcs/files/lecture_notes3.pdf)
  (all accessed 3 July 2013).

- Tedre, Matti; & Sutinen, Erkki (2008), "Three Traditions of Computing: What Educators Should Know", *Computer Science Education* 18(3) (September): 153–170.

Figure 3.3: http://abstrusegoose.com/215

Figure 3.4: http://xkcd.com/894/
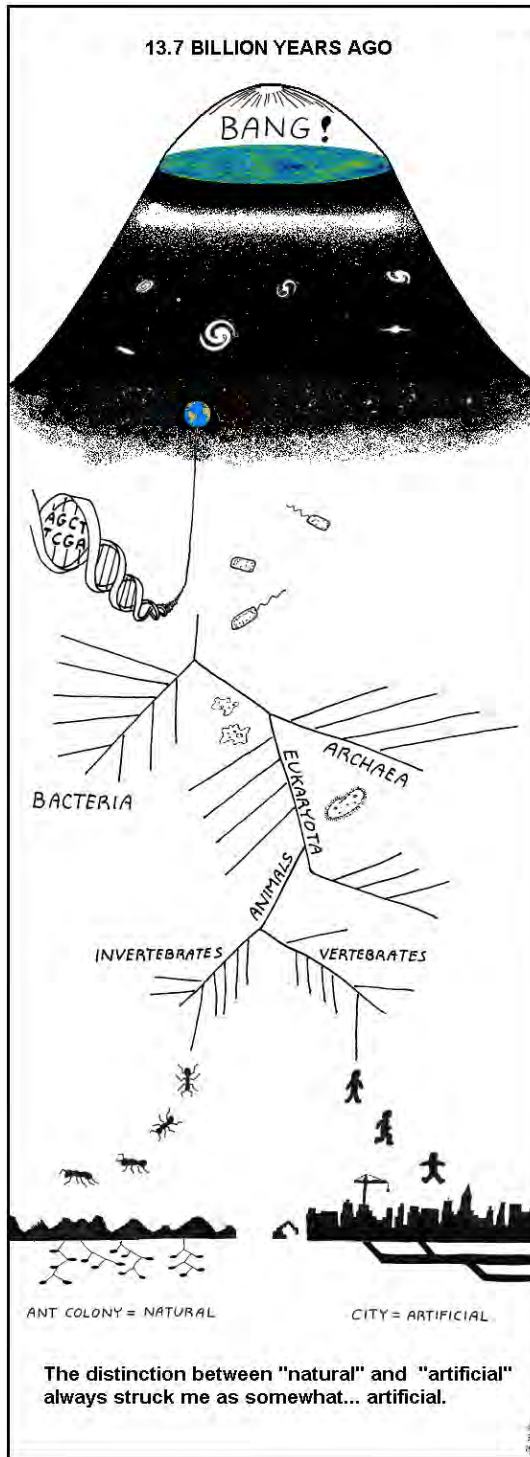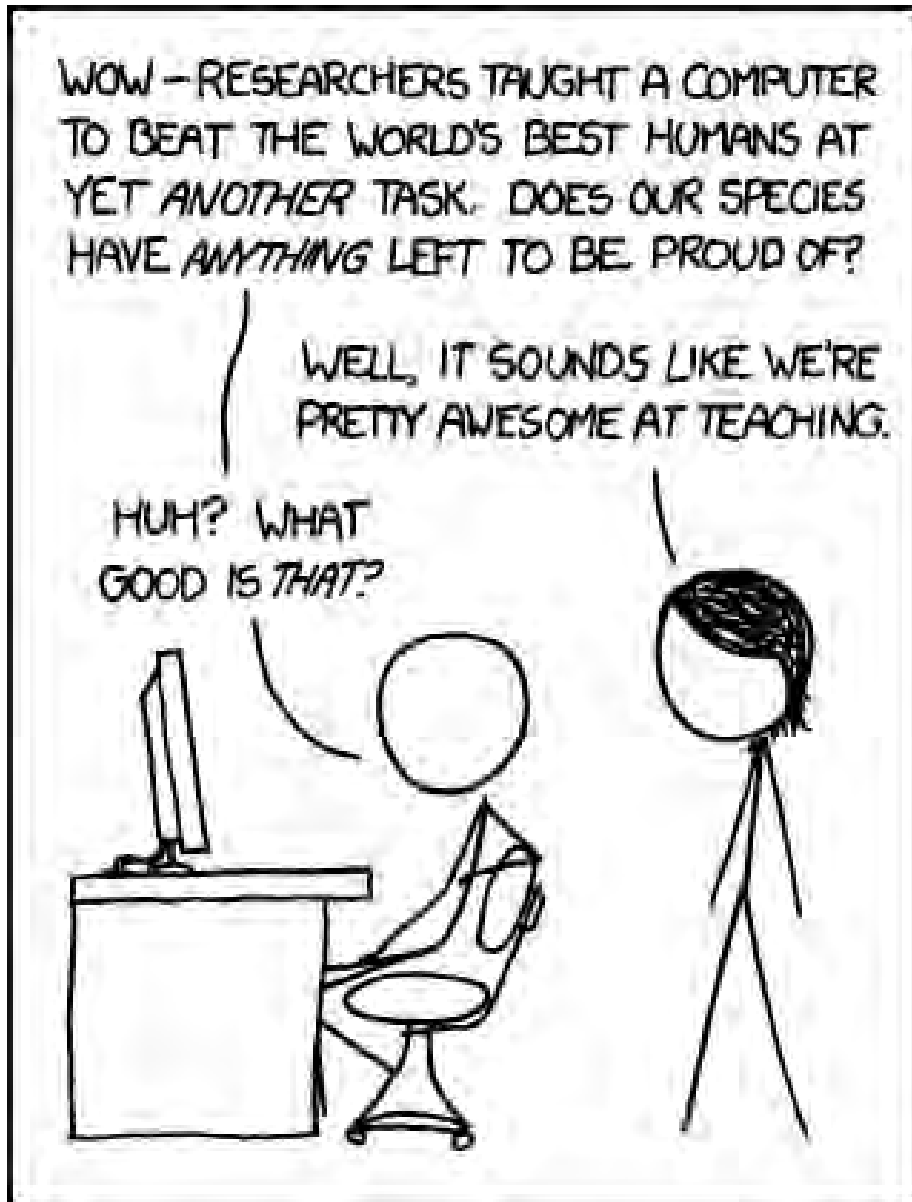
7/3/13 Obj→ / Math↓	computers (& their algorithms)	algorithms (& computers that execute them)	computer processes	procedures	programming	automation	information (processing)	complexity	mathematics	abstractions	problem solving	intelligence
science	newell,perlis,simon67 parnas1998	parnas1998					hartmanis93				roberts1996	
art & sci							forsythe1967					
art					knuth1974b							feigenbaum2003
study	licklider1968	knuth74; denning1985? loui1977 galezer-harel1998	fant 1993		khalil-levy1978	Forsythe 1968; Arden 1980; denning1985	denning1985? loui1977 barwise1989? baj/csyeral92	dijkstra (ceruzzi1989) galezer-harel1998	dijkstra (ceruzzi1989)			
compml/math		wins2006 dijkstra1974		shannon50								
empirical study/ artificial science			newell-simon76	shapiro01								
natural science	simon96											
sci & engg		knuth74 loui87					hartmanis-lin92 hartmanis93 loui87	hartmanis93	hartmanis93 loui1995	(*)brooks96		
engg	brooks96(*)											
CS is (nothing but):	mathematics EE philosophy	krantz1984 ceruzzi1989										

Figure 3.5: Outline of Positions on What CS Is

Chapter 4

What Is Science?

The most remarkable discovery made by scientists is science itself. The discovery must be compared in importance with the invention of cave-painting and of writing. Like these earlier human creations, science is an attempt to control our surroundings by entering into them and understanding them from inside. And like them, science has surely made a critical step in human development which cannot be reversed. We cannot conceive a future society without science.
— Jacob Bronowski, *Scientific American* (1958)

Science is the field in which rationality reigns.
— John Worrall [Worrall, 1989, 109]

[A] scientific education is, to a considerable extent, an exercise in taming the authority of one's intuition.
— H. David Orr [Orr, 2013, 27]

4.1   Recommended Readings:

In doing these readings, remember that our ultimate question is whether computer science is a science.

1. Required:

   - Kemeny, John G. (1959), *A Philosopher Looks at Science* (Princeton: D. van Nostrand).

     – Introduction, pp. ix–xii
     – Ch. 5, "The [Scientific] Method", pp. 85–105.
     – You can skim Ch. 10, "What Is Science?", pp. 174–183, because his answer is just this: A science is any study that follows the scientific method.

   - Kolak, Daniel; Hirstein, William; Mandik, Peter; & Waskan, Jonathan (2006), *Cognitive Science: An Introduction to Mind and Brain* (New York: Routledge), §4.4.2. "The Philosophy of Science"

   - Papineau, David (1996), "Philosophy of Science", in Nicholas Bunnin & E.P. Tsui-James (eds.), *The Blackwell Companion to Philosophy* (Oxford: Blackwell): 290–324.

     – §§1.1–1.3 ("The Problem of Induction", "Popper's Falsificationism", and "The Failings of Falsificationism"), pp. 290–294
     – §§1.5–2.1 ("Instrumentalism versus Realism", "Theory, Observation and Incommensurability", "The Underdetermination of Theory by Evidence", "The Pessimistic Meta-Induction", "Naturalized Epistemology of Science", "Causation"), pp. 298–310
     – Skim the rest.

2. Very Strongly Recommended:

   - Quine, Willard van Orman (1951), "Two Dogmas of Empiricism", *Philosophical Review* 60: 20–43, http://www.ditext.com/quine/quine.html
     (accessed 5 November 2012).

   - Popper, Karl R. (1953), "Science: Conjectures and Refutations", in Karl R. Popper, *Conjectures and Refutations: The Growth of Scientific Knowledge* (New York: Harper & Row, 1962).

   - Kuhn, Thomas S. (1962), *The Structure of Scientific Revolutions* (Chicago: University of Chicago Press), Ch. IX, "The Nature and Necessity of Scientific Revolutions"; online at:
     http://www.marxists.org/reference/subject/philosophy/works/us/kuhn.htm
     (accessed 5 November 2012).

3. Strongly Recommended:

   - Hempel, Carl G. (1966), *Philosophy of Natural Science* (Englewood Cliffs, NJ: Prentice-Hall), "Scope and Aim of this Book".

     – On empirical vs. non-empirical sciences.

   - Kyburg, Henry E., Jr. (1968), *Philosophy of Science: A Formal Approach* (New York: Macmillan), Ch. 1, "The Concept of a Formal System".

- Ziman, John M. (1968), "What Is Science?", in John M. Ziman, *Science Is Public Knowledge* (Cambridge, UK: Cambridge University Press), reprinted in Alex C. Michalos (ed.) *Philosophical Problems of Science and Technology* (Boston: Allyn & Bacon, 1974): 1–27.

- Salmon, Wesley C. (1984), *Scientific Explanation and the Causal Structure of the World* (Princeton: Princeton University Press), Ch. 1, "Scientific Explanation: Three General Conceptions".

4. Recommended:

- Okasha, Samir (2002), *Philosophy of Science: A Very Short Introduction* (Oxford: Oxford University Press).

  - This is the very best introduction to philosophy of science, although it's an entire (but short) book. You may read it instead of any of the above.

4.2   Introduction

We began by asking, "What is computer science?", and we have seen that one answer is that it is a science (or that parts of it are science). Some say that it is a science of computers, some that it is a science of algorithms or procedures, some that it is a science of information processing. And, of course, some say that it is not a science at all, but that it is a branch of engineering; we will explore that option in Chapter 5. In the present chapter, we will explore what it means to be a science, so that we can decide whether computer science is one (or whether parts of it are).

In keeping with my definition of philosophy as the *personal search* for truth by rational means, I won't necessarily answer the question, "Is computer science a science?". But I will provide considerations to help *you* find *and defend* an answer that *you* like. It is more important for *you* to determine an answer for *yourself* than it is for me to present you with my view; this is part of what it means to do philosophy *in* the first person *for* the first person. And it is very important for you to be able to *defend* your answer; this is part of what it means to be rational (it is the view that philosophy is intimately related to critical thinking).

The word 'science' originally simply meant "knowledge" or "knowing"; it derives from the Latin verb *scire*, which meant "to know"[1] But, of course, it has come to mean much more than that.

Let's begin by contrasting the term 'science' with some other terms. First, science is sometimes opposed to "art", not in the sense of the fine arts (such as painting, music, and so on) but in the sense of an informal body of experiential knowledge, or tricks of the trade: information that is the result of personal experience, perhaps unanalyzable (or, at least, unanalyzed), and creative. This is "art" in the sense of "the art of cooking". By contrast, science is formal, objective, and systematic; scientific knowledge is the result of analysis.

This contrast can be seen in the titles of two classic texts in computer science: Donald Knuth's *The Art of Computer Programming* [Knuth, 1973] and David Gries's *The Science of Programming* [Gries, 1981]. The former is a (multi-volume) handbook of different techniques, catalogued by type, but analyzed (albeit incompletely by today's standards). The latter is a compendium of formal methods for program development and verification, an application of logic to programming. (For a detailed defense of the title of Knuth's work, see [Knuth, 1974a].)

Second, as we have seen, science is also sometimes opposed to engineering. Because this will be our focus in Chapter 5, I won't say more about it here.

Finally, science is opposed (in more ways than one) to "pseudo-science" (such as astrology). We will return to this in §4.7.1, because to explain the contrast between science and pseudo-science (disciplines that masquerade as science, but are not science) is part of the philosophical exploration of what science is.

One might think that the philosophy of science would be the place to go to find out what science is, but philosophers of science these days seem to be more interested in questions such as the following (the first two of which are the closest to our question):

---

[1] According to the *Oxford English Dictionary*, http://www.oed.com/view/Entry/172672 (accessed 16 July 2013).

- What is a scientific theory? (Here, the emphasis is on the meaning of the term 'theory'.)

- What is scientific explanation? (Here, the emphasis is on the meaning of the term 'explanation'.)

- What is the role of probability in science?

- What is the nature of induction? (Why) will the future resemble the past?

- What is a theoretical term? (That is, what do the terms of (scientific) theories mean? Do they necessarily refer to something in the real world? For example, there used to be a scientific concept in the theory of heat called 'phlogiston', but we no longer think that this term refers to anything.)

- How do scientific theories change? When they do, are their terms "commensurable": that is, do they mean the same thing in different theories? (For example, what is the relationship between 'phlogiston' and 'heat'? Does 'atom', as used in ancient Greek physics, or even 19th-century physics, mean the same as 'atom' as used in 21st-century physics?)

- Are scientific theories "realistic" (do they attempt to describe the world?) or merely "instrumental" (are they just very good predicting-devices that don't necessarily bear any obvious resemblance to reality, as sometimes seems to be the case with our best current theory of physics, namely, quantum mechanics)?

And so on.

These are all interesting and important questions, and it is likely that a good answer to our question, "What is science?", will depend on answers to all of these. If so, then a full answer will be well beyond our present scope, and the interested reader is urged to explore a good book on the philosophy of science (such as those listed in the Recommended and Further Readings sections of this chapter). Here, we will only be able to consider a few of these questions.

4.3 Early Modern Science

Francis Bacon (1561–1626), a contemporary of Shakespeare, who lived about 400 years ago, devised one of the first "scientific methods". He introduced science as a *systematic study*. (So, when you read about computer scientists who call computer science a "study" rather than a "science", maybe they are not trying to deny that computer science is a science but are merely using a euphemism.)

To study something, X, systematically is:

- to find positive and negative instances of X—to find things are *are X*s and things that are *not X*s;

- to make changes in Xs or their environment (that is, to do experiments);

- to observe Xs and to observe the effects of experiments performed with them;

- to find correlations between *X*s, their behavior, and various aspects of their environment.

One important question in the history of science has concerned the nature of these correlations. Are they (merely) *descriptions*, or are they *explanations*? In other words, is the goal of science to describe the world, or is it to explain the world?

4.4   The Goals of Science

4.4.1   Description as the Goal of Science

Ernst Mach (1838–1916) was a physicist and philosopher of science who lived about 130 years ago, at the time when the atomic theory was being developed. He was influenced by Einstein's theory of relativity and is probably most famous for having investigated the speed of sound (which is now measured in "Mach" numbers, "Mach 1" being the speed of sound).

For Mach, the goal of science was to discover regular patterns among our sensations in order to enable the prediction of future sensations. Thus, Mach was a realist, because, in order to discover something, it must really be there. Perhaps it seems odd to you that a physicist would be interested in our sensations rather than in the world outside of our sensations. This makes it sound as if science should be done "in the first person, for the first person", just like philosophy! That's almost correct; many philosophically oriented scientists at the turn of the last century believed that science should begin with observations, and what are observations but our sensations? The great 18th-century philosopher Immanual Kant distinguished between what he called 'noumena' (or "things in themselves") and what he called 'phenomena' (or things as we perceive and conceive them); he claimed that we could only have knowledge about phenomena, not noumena, because we could not get outside of our first-person, subjective ways of conceiving and perceiving the world.

For Mach, scientific theories are (merely) shorthand descriptions of how the world appears to us. According to the philosophy of science known as "physicalism", sensory perception yields reliable (but corrigible)[2] knowledge of ordinary, medium-sized physical objects and events. For Mach, because atoms were not observable, there was no reason to think that they exist.

4.4.2   Explanation as the Goal of Science

By contrast, the atomic theory was an attempt to *explain* why the physical world appears the way it does. Such a goal for science is to devise theories that explain observed behavior. Such theories are not merely summaries of our observations, but go beyond our observations to include terms that refer to things (like atoms) that might not be able to be observed (yet).

Here is a question for you to think about: Are some computer programs theories? In particular, consider an AI program that allows a robot to "see" or to use natural

---

[2]I.e., "correctable".

language. Does such a program constitute a psychological (hence scientific) *theory* of vision or language? If so, would it be a *descriptive* theory or an *explanatory* one? (We'll return to this topic in Ch. 15.)

4.5 Instrumentalism vs. Realism

Suppose we have an explanatory scientific theory of something, say, atomic theory. Such theories, as we have seen, often include "unobservables"—terms referring to things that we have not (yet) observed but whose existence would help explain things that we *have* observed. One way of looking at this is to think of an experiment as taking some input (perhaps some change deliberately made to some entity being studied) and observing what happens after the experiment is over—the output of the experiment. Between the input and the output, something happens, but we don't necessarily know what it is. It is as if what we are studying is a "black box", and all we can observe are its inputs and outputs. A scientific theory (or, for that matter, a computer algorithm!) can be viewed as an explanation of what is going on inside the black box. (Can it be viewed merely as a description of what's going on inside? Probably not, because you can only describe what you can observe and, by hypothesis, we can't observe what's going on inside the black box.) Such a theory will usually involve various unobservables structured in various ways.

Do the unobservables that form part of such an explanatory theory exist? If you answer 'yes', then you are a "realist"; otherwise, you are an "instrumentalist". A realist believes in the real existence of explanatory unobservables. An instrumentalist believes that they are merely useful tools (or "instruments").

In Mach's time, it was not clear how to treat the atomic theory. Atoms were clearly of instrumental value, but there was no observable evidence of their existence. But they were so useful scientifically that it eventually became unreasonable to deny their existence, and, eventually, they were observed.

In our time, quantum mechanics poses a similar problem. If the world really is as quantum mechanics says that it is (and quantum mechanics is our best current theory about how the world is), then the world is really weird. So, possibly quantum mechanics is merely a useful calculating tool for scientific prediction and shouldn't be taken literally as a description of the real world. (On the other hand, recent research in cognitive science suggests that quantum-mechanical methods applied at the macrosopic level might provide better explanations of certain psychological findings about human cognition than more "standard" methods [Wang et al., 2013].)

The great 20th-century philosopher Willard van Orman Quine, in his classic paper "On What There Is" [Quine, 1948] argued that "to be is to be the value of a bound variable". In other words, if your best theory talks about Xs—that is, postulates the existence of Xs—then, according to that theory, Xs exist.

4.6   "The" Scientific Method

People often talk about "the scientific method". There probably isn't any such thing; rather, there are many scientific methods of studying something. But let's look at one version of a scientific method, a version that is interesting in part because it was described by the mathematician John Kemeny, who was also a computer scientist. (He was the inventor of the BASIC computer programming language and helped develop time sharing. (He also worked with Einstein and was president of Dartmouth College.)

His book, *A Philosopher Looks at Science* [Kemeny, 1959, Chs. 5, 10] presents the scientific method as a cyclic procedure. Because cyclic procedures are called 'loops' in computer programming, I will present Kemeny's version of the scientific method as an algorithm that does not halt (an infinite loop):[3]

> **Algorithm Scientific-Method**
> **begin**
>
>> **while** there is a new fact to observe, **do:**
>> {i.e., **repeat until** there are no new facts to observe,
>> which will never happen, so we have *permanent inquiry*}
>>
>>> **begin**
>>> 1. *observe* things & events;
>>>
>>>> {Express these observations as statements about
>>>> particular objects a, b, ...:
>>>>
>>>> $$Pa \rightarrow Qa,$$
>>>> $$Pb \rightarrow Qb,$$
>>>> $$\ldots$$
>>>>
>>>> Note: observations may be "theory-laden",
>>>> that is, based on assumptions}
>>>
>>> 2. *induce* general statements;
>>>
>>>> $$\{\forall x[Px \rightarrow Qx]\}$$
>>>
>>> 3. *deduce* future observations; {prediction}
>>>
>>>> $$\{Pc/\therefore Qc\}$$
>>>
>>> 4. *verify* predictions against observations;
>>>
>>>> {**if** Qc
>>>>   **then** general statement confirmed
>>>>   or consistent with theory
>>>>   **else** revise theory (or... }
>>>
>>> **end**
>>
>> **end.**

---

[3]This "algorithm" is written in an informal pseudocode. Terms in **boldface** are control structures. Expressions in {braces} are comments.

Kemeny's version of the scientific method is a cycle (or "loop") consisting of observations, followed by inductive inferences, followed by deductive predictions, followed by verifications. The scientist begins by making individual observations of specific objects and events, and expresses these in language: Object *a* is observed to have property P, object *a* is observed to have property Q, object *a*'s having property P is observed to precede object *a*'s having property Q, and so on. Next, the scientist uses inductive inference to infer from a series of events of the form P*a* precedes Q*a*, P*b* precedes Q*b*, etc., that whenever any object *x* has property P, it will also have property Q. So, the scientist who observes that object *c* has property P will deductively infer (that is, will predict) that object *c* will also have property Q *before observing it*. The scientist will then perform an experiment to see whether Q*c*. If Q*c* is observed, then the scientist's theory that $\forall x[Px \rightarrow Qx]$ will be verified; otherwise, the theory will need to be revised in some way. (Similarly, [Denning, 2005, 28] says that "The scientific paradigm... is the process of forming hypotheses and testing them through experiments; successful hypotheses become models that explain and predict phenomena in the world." He goes on to say, "Computing science follows this paradigm in studying information processes".)

For Kemeny, explanation in science works this way: An observation is explained by means of a deduction from a theory. This is the philosopher Carl Hempel's Deductive-Nomological Theory of Explanation. It is "deductive", because the statement that some object *c* has property Q (Q*c*) is explained by showing that it can be validly deduced from two premises: that *c* has property P (P*c*) and that all Ps are Qs ($\forall x[Px \rightarrow Qx]$). And it is "nomological", because the fact that all Ps are Qs is lawlike or necessary, not accidental: Anything that is a P *must* be a Q. (This blending of induction and deduction is a modern development; historically, Bacon (and other "empiricists", chiefly in Great Britain) emphasized experimental "induction and probabilism", while Descartes (and other "rationalists", chiefly on the European continent) emphasized "deduction and logical certainty" [Uglow, 2010, 31].)

Finally, according to Kemeny, any discipline that follows the scientific method is a science. This rules out astrology, on the grounds that astrologers never verify their predictions.[4]

4.7 Alternatives to "The Scientific Method"

This "hypothetical-deductive method" of "formulating theoretical hypotheses and testing their predictions abainst systematic observation and controlled experiment" is the classical, or popular, view of what science and the scientific method are. And, "at a sufficiently abstract level", all sciences "count as using" it [Williamson, 2011]. But there are at least two other views of the nature of science that, while generally agreeing on the distinctions between science as opposed to art, engineering, and pseudo-science, differ on the nature of science itself.

---

[4]Or that their predictions are so vague that they are always trivially verified. See §4.7.1, below.

4.7.1   Falsifiability

According to the philosopher Karl Popper (1902–1994), the scientific method (as propounded by people like Kemeny) is a fiction. The real scientific method sees science as a sequence of *conjectures* and *refutations*:

1. Conjecture a theory (to explain some phenomenon).

2. Compare its predictions with observations
   (i.e., perform experiments to test the theory).

3. **If** an observation differs from a prediction,
   then the theory is *refuted* (or *falsified*)
   else the theory is *confirmed*.

There are two things to note about the last part of this: First, by 'falsifiable', Popper meant something like "capable of being falsified *in principle*", not "capable of being falsified with the techniques and tools that we *now* have available to us". Seond, 'confirmed' does not mean "true"! Rather, it means that the theory is *not yet falsified*! This is because there might be some other explanation for the predicted observation. Just because a theory T predicts that some observation O will be made, and that observation is indeed made, it does not follow that the theory is true! In other words, the argument:

$$T \rightarrow O$$
$$O$$
$$\therefore T$$

is an invalid argument (called the Fallacy of Affirming the Consequent). If T is false, then the first premise is still true, so we could have true premises and a false conclusion. This might also be called the fallacy of circumstantial evidence, where T is the (possibly false) circumstantial evidence that could explain O, but there might be other evidence that also explains O and which *is* true.

So, what is science according to Popper? A theory or statement is scientific if and only if it can be falsified. This is why astrology (and any other superstition) is only pseudo-science: It cannot be falsified, because it is too vague. (As the physicist Freeman Dyson once wrote, "Progress in science is often built on wrong theories that are later corrected. It is better to be wrong than to be vague" [Dyson, 2004, 16].) For Popper, falsifiability also ruled out two other candidates for scientific theories: Freudian psychotherapy and Marxism as an economic theory. The vaguer a statement or theory is, the harder it is to falsify. When I was in college, one of my friends came into my dorm room, all excited about an astrology book he had found that, he claimed, was really accurate. He asked me what day I was born; I said "September 30th". He flipped the pages of his book, read a horoscope to me, and asked if it was accurate. I said that it was. He then smirked and told me that he had read me a random horoscope, for April 16th. The point was that the horoscope for April 16th was so vague that it also applied to someone born on September 30th!

But is astrology really a Popperian pseudo-science? Although I agree that the popular newspaper style of astrology no doubt is (on the grounds of vagueness),

"real" astrology, which might be considerably less vague, might actually turn out to be testable and, presumably, falsified, hence falsifiable. But that would make it scientific (albeit false)!

It is worthwhile to explore the logic of falsifiability a bit more. Although the invalid argument form above seems to describe what goes on, it can be made more detailed. It is not the case that scientists deduce predictions from theories alone. There are usually background beliefs that are independent of the theory being tested (for example, beliefs about the precision of one's laboratory equipment). And one does not usually test a complete theory T but merely one new hypothesis H that is being considered as an addition to T. So it is not simply that the argument above should begin with a claim that the theory T implies prediction O. Rather, theory T, conjoined with background beliefs B, conjoined with the actual hypothesis H being tested is supposed to logically predict that O will be observed.

$$(T \, \& \, B \, \& \, H) \rightarrow O$$

Suppose that O is not observed:

$$\neg O$$

What follows from these two premises? By the rule of inference called 'modus tollens', we can infer:

$$\neg(T \, \& \, B \, \& \, H)$$

But from this, it follows (by DeMorgan's Law) that:

$$\neg T \, \vee \, \neg B \, \vee \, \neg H$$

That is, either T is false, or else B is false, or else H is false, or any combination of them is false. What this means is that if you strongly believe in your hypothesis H that seems to be inconsistent with your observation O, you do not need to give up H. Instead, you could give up the rest of your theory T (or simply revise some part of it), or you could give up (some part of) your background beliefs B (for example, you could blame your measuring devices as being too inaccurate). As [Quine, 1951] pointed out, you could even give up the laws of logic if the rest of your theory has been well confirmed; this is close to the situation that obtains in contemporary quantum mechanics.

4.7.2 Scientific Revolutions

Thomas Kuhn (1922–1996), a historian of science, rejected both the classic scientific method and Popper's falsifiability criterion ("a plague on both your houses", he might have said). Based on his studies of the history of science, Kuhn claimed that the real scientific method works as follows [Kuhn, 1962, Ch. 9]:

1. There is a period of "normal" science, based on a "paradigm", that is, on a generally accepted theory. During that period of normal science, a Kemeny-like (or Popper-like) scientific method is in operation.

2. If that theory is challenged often enough, there will be a "revolution", and a new theory—a new paradigm—will be established, completely replacing the old one.

3. A new period of normal science follows, now based on the new paradigm, and the cycle repeats.

[Dyson, 2004, 16] refers to the "revolutionaries" as "those who build grand castles in the air" and to the "normal" scientists as "conservatives...who prefer to lay one brick at a time on solid ground".

   The most celebrated example was the Copernican revolution in astronomy [Kuhn, 1957]. "Normal" science at the time was based on Ptolemy's "paradigm" of an earth-centered theory of the solar system. But this was so inaccurate that its advocates had to keep patching it up to make it consistent with observations. Copernicus's new paradigm—the heliocentric theory that we now believe—overturned Ptolemy's paradigm. Other revolutions include those of Newton (who overthrew Aristotle's physics), Einstein (who overthrew Newton's), Darwin (whose theory of evolution further "demoted" humans from the center of the universe), Watson and Crick ("whose discovery of the...structure of DNA...changed everything" in biology [Brenner, 2012, 1427]), and Chomsky in linguistics (even though some linguists today think that Chomsky was wrong [Boden, 2006]).

4.8   Is Computer Science a Science?

These are only three among many views of what science is. Is computer science a science according to any of them? This is a question that I will leave to the reader to ponder. But here are some things to consider:

- Does computer science follow Kemeny's scientific method? For that matter, does *any* science (like physics, chemistry, or biology) really follow it? Does *every* science follow it (what about astronomy or cosmology)? Or is it just an idealized vision of what scientists are supposed to do?

- Even if it is just an idealization, does computer science even come close? What kinds of theories are there in computer science? How are they tested?

- Are any computer-science theories ever refuted?

- Have there been any revolutions in computer science? (Is there even a current Kuhnian paradigm?)

   And there are other considerations: What about disciplines like mathematics? Math certainly seems to be scientific in some sense, but is it a science like physics or biology? Is computer science, perhaps, more like math than like these (other) sciences? This raises another question: Even if computer science is a science, what kind of science is it?

4.9 What Kind of Science Might Computer Science Be?

Hempel once distinguished between two kinds of science: *empirical* sciences and *non-empirical* sciences.

The former explore, describe, explain, and predict various occurrences in the world. Such descriptions or explanations are empirical statements that need empirical (that is, experimental) support. The empirical sciences include the *natural* sciences (physics, chemistry, biology, some parts of psychology, etc.) and the *social* sciences (other parts of psychology, sociology, anthropology, economics, political science, history, etc.). Does computer science fit in here? (If so, where?)

The non-empirical sciences are logic and mathematics. Their non-empirical statements don't need empirical support. Yet they are true of, and confirmed by, empirical evidence (though exactly how and why this is the case is still a great mystery; see [Wigner, 1960], [Hamming, 1980]).[5] So, does computer science fit here, instead?

Computer science arose from logic and math (we will see how in Chapter 6, when we look into the nature of computers and algorithms). But it also arose from the development of the computer as a tool to solve logic and math problems. This brings it into contact with the empirical world and empirical concerns such as space and time limitations on computational efficiency (or "complexity" [Loui, 1996, Aaronson, 2013]).

One possible way of adding computer science to Hempel's taxonomy is to realize that psychology doesn't neatly belong to just the natural or just the social sciences. So, perhaps computer science doesn't neatly belong to just the empirical or just the non-empirical sciences, but that parts of it belong to both. And it might even be the case that the non-empirical aspects of computer science are not simply a third kind of non-empirical science, on a par with logic and math, but are themselves parts of both logic and of math.

Or it might be the case that we are barking up the wrong tree altogether. What if computer science isn't a science at all? This possibility is what we now turn to.

4.10 NOTES FOR NEXT DRAFT

1. Cham's PhD cartoon about scientific method.

2. Calvin and Hobbes cartoon on horoscopes.

3.   Natural science is knowledge about natural objects and phenomena. We ask whether there cannot also be "artificial" science—knowledge about artificial objects and phenomena. Simon 1996:3

4. xkcd cartoon about difference between scientists and normal people.

5. discuss para on Galileo and Descartes in Dyson 2006, bottom of p2 to top of p4

---

[5]A related question about whether scientists need to study mathematics at all has recently been discussed in [Wilson and Frenkel, 2013].

6. Steedman 2008 has some interesting things to say on the differences between a discipline such as physics and a discipline such as computer science (in general) and computational linguistics (in particular), especially in §1, "The Public Image of a Science".

7. Petroski 2008: The first two sections argue that all scientists are sometimes engineers and all engineers are sometimes scientists.

8. Vardi 2010 argues that computing (or "computational science") is not a new kind of science or a new way of doing science, but just a more efficient way of doing the same kind of science that humans have always done.

9. Non Sequitur cartoon on "pre-conceptual science".

10. Maybe put this in an "exercises for the reader" section at the end of the chapter:

    Read [Johnson, 2001]. Then try to reconstruct and evaluate his argument for the proposition that "All Science Is Computer Science".

11. Perhaps science is merely *any* systematic activity, as opposed to a *chaotic* one. There is a computer program called 'AlphaBaby', designed to protect your computer from young children who want to play on your computer but who might accidentally delete all of your files while randomly hitting keys. AlphaBaby's screen is blank; when a letter or numeral key is hit, a colorful rendition of that letter or numeral apears on the screen, and when any other key is hit, a geometric figure or a photograph appears. Most children hit the keys randomly ("chaotically") rather than systematically investigating which keys do what ("scientifically").

    [Williamson, 2011] suggests something similar when he characterizes the "scientific spirit" as "emphasizing values like curiosity, honesty, accuracy, precision and rigor".

12. Here's a simplfied way of thinking about what a scientific theory is: We can begin by considering two things: the world and our beliefs about the world (or our descriptions of the world). Those beliefs or descriptions are theories—theories about the world, about what the world is like. Such theories are *scientific* if they can be tested by emmpirical or rational evidence, in order to see if they are "good" beliefs or descriptions, that is, beliefs or descriptions that are true (that correspond to what the world is really like). The testing can take one of two forms: verification or refutation. A theory is verified if it can be shown that it is consistent with the way the world really is. And a theory is refuted if it can be shown that it is not the way the world really is. A picture can help:

A —————————————————————————————————

B —   —   —   —   —   —   —   —

C —————————————————————————————————

Line *A*—a continuous line—is intended to represent the world, a continuum. Line *B*—a line with gaps—is intended to represent observations that we can make about the world; some parts of the world we can observe, but others we can't (those are the gaps). The solid lines in *B* represent things that we believe about the world; the gaps represent things that we don't know (yet) about the world. Line *C* is intended to represent a scientific theory about the world (about line *A*); here, the gaps are filled in. Those fillings-in are predictions about what the world is like at those locations where we cannot observe it; they are guesses (hypotheses) about the world.

13. On "art" vs. science:
According to [Knuth, 1974a, 668], 'art' in the non-painting sense once "meant something devised by man's intellect, as opposed to activities derived from nature or instinct", as in the "liberal arts" (grammar, rhetoric, logic, arithmetic, geometry, music, and astronomy), and it later came to be "used...for the application of knowledge" (where 'science' was "used to stand for knowledge"):

> The situation was almost exactly like the way in which we now distinguish between "science" and "engineering."

But he later has something more interesting to suggest:

> 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. Since the notion of an algorithm or a computer program provides us with an extremely useful test for the depth of our knowledge about any given subject, the process of going from an art to a science means that we learn how to automate something.

(In [Knuth, 2001, 168], he adds this: "Every time science advances, part of an art becomes a science, so art loses a little bit. Yet, mysteriously, art always seems to register a net gain, because as we understand more we invent more new things that we can't explain to computers.")

14. Distinguish between the everyday sense of 'theory' and the technical sense. In the everyday sense, a "theory" is merely an idea which may or may not have any evidence to support it. In the technical sense, a "theory" is a set of statements that describe and explain some phenomenon, often formalized mathematically or logically; to be "scientific", such a theory must also be accompanied by confirming evidence (and, if Popper is right, its statements must be falsifiable). Note that both 'theory' and 'theorem' are etymologically related.

Anti-evolutionists (creationists and advocates of "intelligent design") sometimes criticize the scientific theory of evolution as "merely a theory". Anyone who does so is confusing the everyday sense (in which 'theory' is opposed to 'fact') with the technical sense—evolution *is* a theory is the technical sense.

For a wonderful discussion of the nature of science with respect to evolution, see the judge's opinion in the celebrated legal case of Kitzmiller v. Dover Area School District (especialy §4, "Whether ID [Intelligent Design] Is Science"), online at:
http://ncse.com/files/pub/legal/kitzmiller/highlights/2005-12-
20_Kitzmiller_decision.pdf
(accessed 22 July 2013).

Also note: Evolution can be considered as automated design, instead of intelligent design.

15.   [A] science is an evolving, but never finished, interpretive system. And fundamental to science... is its questioning of what it thinks it knows. ... Scientific knowledge... is a system for coming to an understanding. [Barr, 1985]

     Science is *all about* the fact that we don't know everything. *Science is the learning process.* [Dunning, 2007]

     [Science is not a collection of truths. It is a continuing exploration of mysteries. [Dyson, 2011b, 10]

16. Francis Bacon...

     told us to ask questions instead of proclaiming answers, to collect evidence instead of rushing to judgment, to listen to the voice of nature rather than to the voice of ancient wisdom. [Dyson, 2011a, 26]

17. In a commentary on a paper [Zhai and Kristensson, 2012] describing a new technology for entering text on a computer, [Buxton, 2012] lauds the combination of engineering and pure science involved in the develpment of this new technology, and comments that "it is an exemplary demonstration that reinforces that computer science is a science, and that today the nature of that science has a breadth that extends beyond its traditional roots". This, perhaps, reinforces the "portmanteau" nature of computer science.

18. More on Bacon:

     The concept of replicability in scientific research was laid out by Francis Bacon in the early 17th century, and it remains the principal epistemological tenet of modern science. Replicability begins with

> the idea that science is not private; researchers who make claims must allow others to test those claims. [Wainer, 2012, 358]

19. Besides the triumverate of Bacon's scientific method, Popper's falsificationism, and Kuhn's paradigms and revolutions, there are other approaches to the nature of science. For instance, the philosopher of science Michael Polanyi argued in favor of science as being "socially constructed", not purely rational or formal (see [Kaiser, 2012] for an overview). And another philosopher of science, Paul Feyerabend, also critiqued the rational view of science, from an "anarchic" point of view (see [Feyerabend, 1975], [Preston, 2012], and compare similar remarks in [Quillian, 1994, §2.2]). But exploration of alternatives such as these are beyond our scope.

20.   The history of science is partly the history of an idea that is by now so familiar that it no longer astounds: the universe, including our own existence, can be explained by the interations of little bits of matter. We scientists are in the business of discovering the laws that characterize this matter. We do so, to some extent at least, by a kind of reduction. The stuff of biology, for instance, can be reduced to chemistry and the stuff of chemistry cna be reduced to physics [Orr, 2013, 26]

The reductionist picture can be extended at both ends of the spectrum that Orr mentions: At one end, sociology can be reduced to psychology, and psychology to biology; at the other, perhaps physics can be reduced to mathematics, and mathematics to logic and set theory. CITE PHD CARTOON ON THIS. On this view, it's hard to see where computer science fits into this sequence. On the other hand, not everyone thinks that this chain of reductions is legitimate [Fodor, 1974]. For another thing, it's not obvious that computer science is "in the business of discovering the laws that characterize... matter". We might try to argue that the universe isn't made up of matter, but of information. Then, if you are also willing to say that computer science is the science of information (or of information processing), you could conclude that it is a science.

21. On instrumentalism:
    Galileo...

> ...and the Church came to an implicit understanding: if he would claim his work only as "*istoria,*" and not as "*dimonstrazione,*" the Inquisitors would leave him alone. The Italian words convey the same ideas as the English equivalents: a new story about the cosmos to contemplate for pleasure is fine, a demonstration of the way things work is not. You could calculate, consider, and even hypothesize with Copernicus. You just couldn't believe in him. [Gopnik, 2013, 107]

Gopnik goes on to observe, with respect to the tension between empirical observation and rational deduction:

> Though Galileo...wants to convince...[readers] of the importance
> of looking for yourself, he also want to convince them of the
> importance of *not* looking for yourself. The Copernican system
> is counterintuitive, he admits—the earth certainly doesn't seem to
> move. It takes intellectual courage to grasp the argument that it does.
> [Gopnik, 2013, 107]

22. [Colburn, 2000, 168] draws an analogy between the scientific method of
    formulating, testing, and (dis)confirming hypotheses and the problem-solving
    method of computer science consisting of formulating, testing, and accepting-
    or-rejecting an algorithm. Besides suggesting that computer science is (at least
    in part) scientific, this analogizes algorithms to scientific hypotheses or theories.
    (See Ch. 15 for further discussion.)

23. Question for end-of-chapter discussion:
    Computer science is not the only alleged science whose scientific credentials
    have been questioned. Even excepting such "pseudosciences" as astrology, the
    disciplines of psychiatry and political science have both come under attack. At
    the time of writing, economics is another such discipline whose status as a
    science (even as a "social" science) has been doubted by some critics. Find some
    essays that have been critical of the scientific status of one of these disciplines
    (or of some other discipline) and consider whether the arguments that have been
    used to justify or to challenge their status as a science can be applied to computer
    science. (For economics, you might consider [Rosenberg, 1994], [Leiter, 2004],
    [Leiter, 2005], [Leiter, 2009], [Chetty, 2013]. For psychiatry, take a look at the
    writings of Adolf Grünbaum at: http://tinyurl.com/lsstp7a[6]
    For astrology, [Thagard, 1978].)

24. On the "pessimistic meta-induction" that all statements
    of science are false, see http://plato.stanford.edu/entries/structural-realism/ and
    https://en.wikipedia.org/wiki/Pessimistic_induction

25. Probably need to add a short section on structuralism, to complement sections
    on realism and instrumentalism. Can then cite under Further Sources:
    Floridi, Luciano (2004), "Informational Realism", in J. Weckert & Y. Al-
    Saggaf (eds.), *Computing and Philosophy Conference (Canberra, Australia);
    Conferences in Research and Pratice in Information Technology, Vol. 37*: 7–
    12 (Australian Computer Society);
    accessed 18 June 2014 from:
    http://philsci-archive.pitt.edu/2538/1/ir.pdf
    which discusses the relevance of object-oriented programming to structuralism.

```
@misc{pittphilsci2538,
          month = {January},
```

```
        title = {Informational Realism},
       author = {Luciano Floridi},
         year = {2005},
         note = {published in ACS Conferences in Research and
Practice in Information Technology (CRPIT), 2004, (3), pp. 7-12.},
     keywords = {Structural realism, epistemic structural realism,
ontic structural realism, levels of abstraction, informational ontology,
object oriented programming},
          url = {http://philsci-archive.pitt.edu/2538/},
     abstract = {What is the ultimate nature of reality? This paper
defends an answer in terms of informational realism (IR). It does so in
three stages. First, it is shown that, within the debate about
structural realism (SR), epistemic (ESR) and ontic (OSR) structural
realism are reconcilable by using the methodology of the levels of
abstractions. It follows that OSR is defensible from a
structuralist-friendly position. Second, it is argued that OSR is also
plausible, because not all related objects are logically prior to all
relational structures. The relation of difference is at least as
fundamental as (because constitutive of) any relata. Third, it is
suggested that an ontology of structural objects for OSR can reasonably
be developed in terms of informational objects, and that Object Oriented
Programming provides a flexible and powerful methodology with which to
clarify and make precise the concept of ?informational object?. The
outcome is informational realism, the view that the world is the
totality of informational objects dynamically interacting with each
other.?}
}
```

26. [Cockshott and Michaelson, 2007,   §2.5,
    p. 235] suggest that the hypercomputation challenges to the CTCT are examples
    of Kuhnian revolutionary paradigmatic challenges to the "normal" science of
    computer science. (They don't think that the challenges are successful, however.)

27. At least one major philosopher has said something that suggests that engineering
    might be a part of science:

    > I have never viewed prediction as the main purpose of science,
    > although it was probably the survival value of the primitive precursor
    > of science in prehistoric times. *The main purposes of science are
    > understanding (of past as well as future),* **technology**, *and control of
    > the environment.* [Quine, 1988]

    If "technology" can be identified with engineering (and why shouldn't it be?),
    then this puts engineering squarely into the science camp, rendering the science-
    vs.-engineering debates moot (though still not eliminating the need to ask what
    engineering—or technology—is).

28.   In a word, the general public has trouble understanding the
        provisionality of science.   Provisionality refers to the state of
        knowledge at a given time. Newton's laws of gravity, which we all
        learn in school, were once thought to be complete and comprehensive.
        Now we know that while those laws offer an accurate understanding
        of how fast an apple falls from a tree or how friction helps us take
        a curve in the road, they are inadequate to describe the motion of
        subatomic particles or the flight of satellites in space. For these we
        needed Einstein's new conceptions. [Natarajan, 2014, 64]

        Einstein's theories did not refute Newton's; they simply absorbed
        them into a more comprehensive theory of gravity and motion.
        Newton's theory has its place and it offers an adequate and accurate
        description, albeit in a limited sphere. As Einstein himself once put
        it, "The most beautiful fate of a physical theory is to point the way
        to the establishment of a more inclusive theory, in which it lives as a
        limiting case." It is this continuously evolving nature of knowledge
        that makes science always provisional. [Natarajan, 2014, 65]

   Somewhere, need to mention this issue of provisionality, maybe in connection
   with Popper's falsificationism. Question to think about: In what sense, if
   any, is computer science "provisional"? Nelson Pole has suggested (private
   communication, 3/9/15) that "if there is a bug lurking in every moderately
   complex program, then *all* programs are provisional".

29. A great closing/summary quote:

        Science, after all, is simply a logical, rational and careful examination
        of the facts that nature presents to us. (The Amazing Randi, as quoted
        in [Higginbothan, 2014, 53].) Although Shapiro would be happy with
        the word 'nature' here, others might not be, but I think that it can
        be eliminated without loss of meaning and still apply to computer
        "science".

30. Comments on [Simon, 1996b]:

   (a)   A natural science is a body of knowledge about some class
            of things—objects or phenomena—in the world: about the
            characteristics and properties that they have; about how they
            behave and interact with each other.
                The central task of a natural science is to make the wonderful
            commonplace: to show that complexity, correctly viewed, is only
            a mask for simplicity; to find pattern hidden in apparent chaos.
            [Simon, 1996b, Ch. 1, p. 1]

        The emphasis here is on "natural", because he is ultimately going to discuss
        sciences of the "artificial". So, presumably, a science of the artificial would
        be such a body of knowledge about some class of *artificial* things (as
        opposed to "things in the world"). But, because artificial things are, surely,

things in the world (think of beehives and bird nests), I suspect that what he means is things that are not "naturally" in the world: Note that the word 'natural' does not appear in his characterization of natural science. But what difference is there between what might be called "natural artifacts" such as beehives and birdnests, on the one hand, and artifacts such as machines and computers, on the other? The only difference that I can see is in who made them: non-humans vs. humans (and Simon says things that are consistent with this (p. 2). In that case, a science of the artificial would merely be a science of things made by humans. But it would still be a science.

(b) For instance, he says that "A forest may be a phenomenon of nature; a farm certainly is not" (p. 3). Granted, there might not have been any farms had there not been humans (but what about ant "farms"?). But would he be equally willing to say that a beehive may be a pheomenon of nature; an log house certainly is not? Beehives are just as artificial or non-natural as log houses, differing only in who made them. (Another, better?, example: bird nests vs. shelters made of branches and leaves by some members of Amazonian or sub-Saharan African peoples.)

(c) But Simon is aware of this; he goes on to say:

> A plowed field is no more part of nature than an asphalted street—
> *and no less* [Simon, 1996b, 3, my italics].

And, indeed, he explicitly says:

> But you will have to understand me as using "artificial"... as meaning man-made as opposed to natural. [Simon, 1996b, 4].

I would only amend this slightly to read: "human-made as opposed to non-human-made"

(d) I suppose that there are echoes here of the more general issue of the generalization of terms, such as whether or not to apply 'thinking' to computers (in addition to humans) or whether to apply 'flying' to airplanes (in addition to birds) or even 'noodle' to non-wheat products (see http://www.askphilosophers.org/question/5738). (See also Simon's comments (pp. 15–16) about the meaning of 'moon'—are "artificial" satellites moons?)

(e) His description of the purpose of science is consistent both with Mach's view that science is (merely) a way to describe the world efficiently and with the more modern view that its purpose is to explain or help us understand the world. Note, too, the emphasis on complexity, which elsewhere (WHERE?) we saw/will see that Simon believes is one of the central roles of computer science.

(f) Nevertheless, he comes down later on the side of explanation:

> This is the task of natural science: to show that the wonderful is not incomprehensible, *to show how it can be comprehended...* [Simon, 1996b, 1, my italics]

So, the task is not, *pace* Mach, merely to describe the complexity of the world in simple terms, but to explain the world, to make it understandable.

(g) One of the general paradoxes of analysis or of explanation in general is that, by showing how something mysterious or wonderful or complex is really just a complex of things that are non-mysterious or mundane or simple, we lose sight of the original thing that we were trying to understand or analyze. (We saw/will see this again in Dennett's notion of Turing's "inversion".) Simon demurs:

> ...the task of natural science...[is] to show how it [the wonderful] can be comprehended—*but not to destroy wonder. For when we have explained the wonderful, unmasked the hidden pattern, a new wonder arises at how complexity was woven out of simplicity.* [Simon, 1996b, 1–2, my italics]

So, for instance, the fact (if it is a fact; we will explore this issue later WHERE?) that non-cognitive computers can exhibit (or merely simulate) cognitive behaviors is not necessarily a reductio of cognition as computation but is itself something worthy of wonder and further (scientific) explanation.

(h) He also goes on to suggest an analogy between the emergence of complexity from simplicity and the "emergence" of a representational picture or of a piece of music from "partially concealed patterns", presumably implemented by brush strokes or individual musical notes— although he seems to chalk this up to aesthetics (p. 2).

31.   Science is what we understand well enough to explain to a computer; art is everything else. [Knuth, 2001, 168]

Despite the sense of facetiousness, there is a serious point here: We must understand something well enough—in full detail—in order to express it in, or model it with, a computer program.

Nelson Pole (private communication, 3/9/15) points out that programs that learn need not begin by understanding things that well. I would respond by observing that the programmer of the learning program might not have to understand the topic to be learned that well, but would have to understand the nature of learning well enough to write a learning program.

32.   The goal of any scientific theory or model is to construct mechanisms that account for observable phenomena which can be detected by our sensory systems (possibly mediated by gauges of scientific apparati). [Peschl and Scheutz, 2001a, 1]

One ambiguity in this characterization is whether the constructed mechanism (which, of course, could be expressed as a computer program) is the one that actually caused the phenomena that we observed, and not merely one that could have caused it. In other words, we want our theores to be true. Many different

mechanisms might produce the same phenomena (think of certain kinds of optical illusions or distorted images).

33. As we saw in Chs. 2 and 3, [McGinn, 2015, 85] thinks that both philosophy and computer science are sciences, just as physics and chemistry are. Unlike the latter two, the formal sciences are "not empirical in the usual sense—they are not based on observation and experiment—but they are sufficiently rigorous, organized, and systematic to qualify as science". To make his argument, he has to come up with a characterization of science, but he is more concerned with a broad, almost ordinary-language (or dictionary-definition) characterization than a formal, philosophical analysis.

> I take it that what distinguishes a discourse as scientific are such traits as these: rigor, clarity, literalness, organization, generality (laws or general principles), technicality, explicitness, public criteria of evaluation, refutability, hypothesis testing, expansion of common sense (withe the possibility of undermining common sense), inaccessibility to the layman, theory construction, symbolic articulation, axiomatic formulation, learned journals, rigorous and lengthy education, professional societies, and a sense of apartness from naïve opinion" [McGinn, 2015, 86]

(Possibly add this article as required, or whatever, reading for this chapter.)

34. Put in this chapter, if not used in Ch. 3:

> As we enter life, we all struggle to understand the world. Some of us continue this struggle with *dogged* determination. These are the *scientists*. Some of them realize that computation provides a privileged perspective to understand the world outside and the one within. These are the *computer scientists*. [Micali, 2015, 52].

35. On the nature of theories:
Theories are syntactic in nature:

> When we say that theories are abstract, we mean that theoretical constructs are typically phrased in a vocabulary that is different from the vocabulary of the evidence that supports the theory. [Gopnik, 1996, §3.1, p. 496]

First, theories are "phrased in a vocabulary", hence syntactic. Second, this vocabulary is different from the semantic interpretation of the theory.

4.11   Further Sources of Information

Items in **boldface** are especially important, interesting, or useful.

1. Books and Essays:

   - Recommendations on texts…whose subject is the philosophy of science", AskPhilosophers.org http://www.askphilosophers.org/question/1515
     (accessed 5 November 2012).

   - Arden, Bruce W. (1980), "COSERS Overview", in Bruce W. Arden (ed.), *What Can Be Automated? The Computer Science and Engineering Research Study (COSERS)* (Cambridge, MA: MIT Press), Ch. 1, pp. 1–31; see esp. the section titled "About Names and Labels" (pp. 5–7).

   - Burkholder, Leslie (1999), "Are AI and Mechanics Empirical Disciplines?", *Journal of Experimental and Theoretical Artificial Intelligence* 11: 497–500; DOI:10.1080/095281399146373

     – Discusses the difference between "empirical experimental disciplines" (like mechanics, a branch of physics) and "*a priori* disciplines" (like mathematics).

   - Chaitin, Gregory J. (1968), "On the Difficulty of Computations", in Gregory J. Chaitin, *Thinking about Gödel and Turing: Essays on Complexity, 1970–2007* (World Scientific Publishing): 3–15; online at:
     http://www.worldscientific.com/doi/suppl/10.1142/6536/suppl_file/6536_chap01.pdf
     (accessed 22 July 2013).

     – See esp. §7, on "classical problems of the methodology of science".

   - Corcoran, John (DATE), "Scientific Revolutions", *Encyclopedia of American Philosophy*.

   - Denning, Peter J. (2005), "Is Computer Science Science?", *Communications of the ACM* 48(4) (April): 27–31.

   - Denning, Peter J.; Comer, Douglas E.; Gries, David; Mulder, Michael C.; Tucker, Allen; Turner, A. Joe; & Young, Paul R. (1989), "Computing as a Discipline", *Communications of the ACM* 32(1) (January): 9–23.

   - Dyson, Freeman (2006, 19 October), "Writing Nature's Greatest Book", *New York Review of Books* 53(16).

     – An interesting book review that discusses some of the origins of science. In particular, take a look at fifth and sixth paragraphs, which discuss Galileo and Descartes.

   - Fine, Arthur (1986), "Unnatural Attitudes: Realist and Instrumentalist Attachments to Science", *Mind* 95(378) (April): 149–179

     – A "state of the art" survey article on realism vs. instrumentalism in science.

   - Gordin, Michael D. (2012), "Separating the Pseudo from Science", *The Chronicle [of Higher Education] Review* 59(4) (21 September): B10–B12; online at:
     http://chronicle.com/article/Separating-the-Pseudo-From/134412/
     (accessed 23 July 2013).

   - Hartmanis, Juris, & Lin, Herbert (eds.?) (1992), "What Is Computer Science and Engineering?", in Juris Hartmanis & Herbert Lin (eds.), *Computing the Future:*

A Broader Agenda for Computer Science and Engineering (Washington, DC: National Academy Press), Ch. 6, pp. 163–216.[7]

- Hitchcock, Christopher (2007), "Conceptual Analysis Naturalized: A Metaphilosophical Case Study", *Journal of Philosophy* 103(9) (September): 427–451.

    - §I, "Theories of Causation", pp. 429–433, contains an excellent summary of various contemporary and historical theories of the nature of cause and effect.

- Lehoux, Daryn; & Foster, Jay (2012), "A Revolution of Its Own", *Science* 338 (16 November): 885–886.

    - Review of the 4th edition (2012) of [Kuhn, 1957].

- Lloyd, Seth (2007), "You Know too Much", *Discover Magazine* (28 April).

    - "Most of our exploding collection of science information is junk."

- Lohr, Steve (2006, 2 November), "Group of University Researchers to Make Web Science a Field of Study", *New York Times*: C6.

    - Is "Web science" a science?
    - How is it related to computer science?

- Pigliucci, Massimo; & Boudry, Maarten (eds.) (2013), *Philosophy of Pseudoscience: Reconsidering the Demarcation Problem* (Chicago: University of Chicago Press).

    - The "demarcation problem" is the problem of determining the dividing line between "real" science and "pseudo"science; for example, almost everyone will agree that astronomy is a "real" science and that astrology is not. But what is the difference between "real" and "pseudo" sciences?
    - Also see:
      (a) Pigliucci, Massimo; & Boudry, Maarten (2013), "The Dangers of Pseudoscience", Opinionator/The Stone, *New York Times* (10 October); accessed 18 October 2013 from:
      http://opinionator.blogs.nytimes.com/2013/10/10/the-dangers-of-pseudoscience/ (accessed 18 October 2013)
      (b) Curd, Martin (2014), Review of Pigliucci & Boudry's book, *Notre Dame Philosophical Reviews* (22 July); accessed 21 July 2014 from:
      https://ndpr.nd.edu/news/49425-philosophy-of-pseudoscience-reconsidering-the-demarcation-problem/

- Petroski, Henry (2008), "Scientists as Inventors", *American Scientist* 96(5) (September-October): 368–371.

- Popper, Karl R. (1959), *The Logic of Scientific Discovery* (New York: Harper & Row).

- Quillian, M. Ross (1994), "A Content-Independent Explanation of Science's Effectiveness", *Philosophy of Science* 61(3) (September): 429–448.

---

[7]This book was the subject of a petition (online at: http://www-formal.stanford.edu/jmc/petition/whysign/whysign.html (accessed 16 July 2013)) to withdraw it from publication "because we consider it misleading and even harmful as an agenda for future research". The petition was sponsored by Bob Boyer, John McCarthy, Jack Minker, John Mitchell, and Nils Nilsson. Commentaries on it appeared in [Kling et al., 1993].

– An explanation, in terms of the communication of information, of why the natural sciences are more "effective" than the social sciences, written by one of the early researchers in AI who later became a sociologist of science. Although written in the early days of the World Wide Web, this paper has some interesting implications for the role of social media in political discourse.

• Reed, Dan (2007), "Computing: All Our Constants Are Variables", *Computing Research News* 19(4) (September): 3.

– Makes an interesting distinction between science as "natural philosophy" and as *scientia* (= knowledge).

• Rosenberg, Alex (2000), *Philosophy of Science: A Contemporary Introduction* (London: Routledge).

• Simon, Herbert A. (1996), *The Sciences of the Artificial, 3rd edition* (Cambridge, MA: MIT Press).

– Ch. 1, "The Natural description Artificial Worlds", has sections on the nature of understanding by simulating and on computers as artifacts, as abstract objects, and as empirical objects.

• Steedman, Mark (2008), "On Becoming a Discipline", *Computational Linguistics* 34(1) (March): 137–144.

• Tedre, Matti (2011), "Computing as a Science: A Survey of Competing Viewpoints", *Minds and Machines* 21(3) (August): 361–387.

• Tedre, Matti; & Moisseinen, Nella (2014), "Experiments in Computing: A Survey", *Scientific World Journal*, Vol. 2014, Article ID 549398, http://dx.doi.org/10.1155/2014/549398 (accessed 3 April 2015).

– A survey of the nature of experiments in science, and whether computer science is experimental in nature.

• Thagard, Paul R. (1978), "Why Astrology Is a Pseudoscience", *PSA: Proceedings of the Biennial Meeting of the Philosophy of Science Association*: 223–234.

• Vardi, Moshe Y. (2010), "Science Has Only Two Legs", *Communications of the ACM* 53(9) (September): 5.

• Weinberger, David (2012), "Shift Happens", *The Chronicle [of Higher Education] Review* (27 April): B6–B9.

• Wigner, Eugene (1960), "The Unreasonable Effectiveness of Mathematics in the Natural Sciences", *Communications in Pure and Applied Mathematics* 13(1) (February); online at:
http://nedwww.ipac.caltech.edu/level5/March02/Wigner/Wigner.html
(accessed 5 November 2012).

– An interesting reply from a computer scientist:
Hamming, R.W. (1980), "The Unreasonable Effectiveness of Mathematics", *American Mathematical Monthly* 87(2) (February), online at:
http://www.dartmouth.edu/∼matc/MathDrama/reading/Hamming.html
(accessed 5 November 2012).

• Willingham, Daniel T. (2011), "Trust Me, I'm a Scientist", *Scientific American* (May): 12; online at:
http://www.scientificamerican.com/article.cfm?id=trust-me-im-a-scientist
(accessed 23 July 2013)

        – "Why so many people choose not to believe what scientists say"

2. Websites:

- Abelson, Harold, & Sussman, Gerald Jay, with Sussman, Julie (1996), "Preface to the First Edition", *Structure and Interpretation of Computer Programs*, http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-7.html#%_chap_Temp_4 (accessed 5 November 2012).

- Gopnik, Adam (2015), "The Evolution Catechism", *The New Yorker* online, http://www.newyorker.com/news/daily-comment/evolution-catechism (accessed 24 March 2015)

  – A discussion of the nature of "theory" as it is used in science and in ordinary language.

- Tedre, Matti (2007), "Introduction to the Course", Lecture Notes in the Philosophy of Computer Science, http://cs.joensuu.fi/~mmeri/teaching/2006/philcs/files/lecture_notes1.pdf (accessed 5 November 2012).

Chapter 5

What Is Engineering?

Figure 5.1: ©2009 by Universal UClick

5.1   Recommended Readings:

In doing these readings, remember that our ultimate question is whether computer science is an engineering discipline.

1. Required:

   - Davis, Michael (2009), "Defining Engineering from Chicago to Shantou", *The Monist* 92(3) (July): 325–338.

     – OR [Davis, 1995a] or [Davis, 1996]?? (and move Davis 2009 to further reading)

   - Petroski, Henry (2003), "Early [Engineering] Education", *American Scientist* 91 (May-June): 206–209.

   - Loui, Michael C. (1987), "Computer Science Is an Engineering Discipline", *Engineering Education* 78(3): 175–178.

2. Very Strongly Recommended:

   - Davis, Michael (1998), "Introduction to Engineering", Part I of *Thinking Like an Engineer: Studies in the Ethics of a Profession* (New York: Oxford University Press), esp.:

     – Introduction (p. 3)
     – Ch. 1, "Science, Technology, and Values" (pp. 3–17)
     – "Who Is an Engineer?" (from Ch. 2, pp. 25–28)
     – Introduction to Ch. 3, "Are 'Software Engineers' Engineers?" (pp. 31–32)
     – "The Standard Definition of Engineer" (from Ch. 3, pp. 32–34)
     – "Three Mistakes about Engineering" (from Ch. 3, pp. 34–36)
     – "Membership in the Profession of Engineering" (from Ch. 3, pp. 36–37)

   - Petroski, Henry (2008), "Scientists as Inventors", *American Scientist* 96(5) (September-October): 368–371.

     – The first two sections argue that all scientists are sometimes engineers and all engineers are sometimes scientists.

   - Brooks, Frederick P., Jr. (1996), "The Computer Scientist as Toolsmith II", *Communications of the ACM* 39(3) (March): 61–68.

5.2 Introduction

We began by asking what computer science is (Ch. 3), and we considered that it might be what it says it is: a science; so we then looked at what science is (Ch. 4).

We also considered that computer science might be a branch of engineering; so now it is time to ask what engineering is. What is its relationship to science? And what is the relationship of computer science to engineering?

Answering these questions depends even more on what people *say* about it than does answering the question, "What is science?". This is because the philosophy of engineering is much less well developed than the philosophy of science and, for some reason, there seem to be fewer attempts to try to define 'engineering'. For instance, if you go to various university websites for schools or departments of engineering, you will rarely find a definition. ([Koen, 1988, 307] makes a similar point.)

For engineering, more than for science or even computer science, it seems that engineering *is* what engineers *do*, and so we may need to look at what it is that they do. (As the computer scientist R.W. Hamming once pointed out [Hamming, 1968, 4], "the only generally agreed upon definition of mathematics is 'Mathematics is what mathematicians do', which is followed by 'Mathematicians are people who do mathematics'.")

Dictionary definitions of 'engineering' are even less helpful than usual. Actually, dictionary definitions are rarely useful: First, dictionaries don't always agree, and some are better than others. Second, dictionaries at best tell you how other people use a term, but, if people use a term incorrectly, dictionaries are duty bound to record that. Third, dictionaries can be misleading: *Webster's Ninth New Collegiate Dictionary* [Mish, 1983, 259] defines 'college' as "a body of clergy living together and supported by a foundation"! This may once have been true, and may even still be true in a very limited sense of the term, but why is it listed as the first definition? Because Merriam-Webster dictionaries list definitions in *historical order*!

[Mish, 1983] defines 'engineering' as:

1. The activities or function of an engineer.

2. The application of science and mathematics to make matter and energy useful to people.

The first of these cannot be understood without understanding 'engineer', which is defined thus:

1. A member of a military group devoted to engineering work.

2. A person trained in engineering.

So this dictionary agrees explicitly with Hamming: engineering is what engineers do; engineers are people who do engineering! Independently of the "military group" condition, both of these definitions require us to already understand 'engineering'. Only the second definition of 'engineering' holds out some hope for independent understanding, but it is not at all clear that computer scientists apply science and math

to make matter and energy useful. Some might, by a stretch of meaning, but surely not all.

It might help to look at the etymology of 'engineer'. (It might, but it won't.) The word comes from *in-* + *gen-* + *-or*, where *in-* means "in", *-or* means, roughly, "agent", and *gen-* can mean "mind, intellect, talent; to produce; etc.", none of which seems to help—the word has simply changed too much for us to be able to figure out what it means from its origins.

One philosopher who has tried to explain engineering [Bunge, 1974] does it this way: There are two kinds of science (where science is any discipline that applies the scientific method): pure and applied. Pure sciences apply the scientific method to increasing our knowledge of reality (e.g., cell biology). Applied sciences apply the scientific method to enhancing our welfare and power (e.g., cancer research). Among the applied sciences are operations research (mathematics applied to management), pharmacology (chemistry applied to biology), and engineering. Given this taxonomy, computer science would not be a branch of engineering, though it might be an applied science alongside engineering. Yet there is a "pure" component of computer science, namely, the mathematical theory of algorithms, computability, and complexity (on the first two, see Ch. 7; on the latter, see [Loui, 1996, Aaronson, 2013]).

5.3   A History of Engineering

> Rather than treat software engineering as a subfield of computer science, I treat it as an element of the set, {Civil Engineering, Mechanical Engineering, Chemical Engineering, Electrical Engineering, . . . }. this is not simply a game of academic taxonomy, in which we argue about the parentage of ownership of the field; *the important issue is the content and style of the education.* [Parnas, 1998, 1, my emphasis]

So let us turn to another philosopher for an insight as to what engineering might be [Davis, 1998] (cf. [Davis, 1995a], [Davis, 1996], which contain earlier versions of the same considerations). Davis begins with a history of engineering, which began some 400 years ago in France, where there were "engines"—that is, machines—and "engineers" who worked with them, such as soldiers who used "engines of war" such as catapults and artillery, or carpenters and stonemasons in civilian life who plied these trades as soldiers. From this background comes the expression "railroad engineer" and such institutions as the Army Corps of Engineers.

In 1676, the French army created a separate corps (separate from the infantry) of engineers who were charged with military construction. So, at least in 17th-century France, an engineer was someone who did whatever it was that those soldiers did. Forty years later, in 1716, there were civil engineers: soldiers who built bridges, roads, etc., for civilians.

A distinction was drawn between engineers and architects. The former were trained in math and physics, and were concerned with reliability and other practical matters. They were trained as army officers, hence more disciplined for larger projects. Architects, on the other hand, were more like artists, chiefly concerned with aesthetics.

Engineers in France at that time were trained at the École Polytechnique ("Polytechnic School"), whose curriculum began with a year of science and math, followed gradually by more and more applications to construction (e.g., of roads), culminating in a specialization.

So, at this time, engineering was the application of science "for the use and convenience of" people and for "improving the means of production" [Davis, 1998]. Engineering was not science: Engineers *use* science but don't create new knowledge. Nor is engineering applied science: Engineers are concerned with human welfare (and not even with generality and precision), wheras applied scientists are concerned with applying their scientific knowledge.

Davis concludes that engineering must be defined by its curriculum (by its specific knowledge) and by a professional commitment to use that knowledge consistent with an engineering code of ethics. So, rather than saying that engineering is what engineers *do*, Davis says that engineering is what engineers *learn* and how they *ought* (ethically) to use that knowledge. This, of course, raises the question: What is it that engineers learn?

One question we might raise concerns the one branch of computer science that has always explicitly called itself 'engineering': software engineering. Is software engineering engineering? For Davis, it could be, if and only if there is a professional curriculum for it, along with a code of professional ethics.

Another philosopher [Mitcham, 1994] echoes much of what Davis says. He notes that 'engineer' comes from the "Latin *ingenero*, meaning 'to implant', 'generate', or 'produce' " [Mitcham, 1994, 144]. And he rehearses the military history (albeit with different dates–CHECK THIS OUT). He also distinguishes between the engineer as designer and the technician or technologist as builder. ELABORATE ON THIS.

5.4   Engineering as Design

An alternative view of engineering is due to Henry Petroski [Petroski, 2003, 206], who says that its fundamental activity is *design*. So, engineering is not science, whose fundamental activity is *analysis* [Petroski, 2003, 207]. But a science and an engineering discipline can have the same object, can be about the same thing.

If so, then what is the common object of computer science and computer engineering? Is it computers? Algorithms? Information? Perhaps computer science studies algorithms and procedures, whereas computer engineering studies computers and computer systems. If so, then who studies the relations between these?

One aspect of design has been picked up by [Hamming, 1968]. When one designs something, one has to make choices. Hamming suggests that "science is concerned with *what* is possible while engineering is concerned with *choosing*, from among the many possible ways, *one* that meets a number of often poorly stated economic and practical objectives". This is interesting for a number of reasons.

First, it fits well with much of the work—even theoretical work—that is done by computer scientists. As we saw earlier (§§3.6,3.8, one definition of computer science is that it is concerned with what can be automated. One way of expressing this is as follows: For what tasks can there be an algorithm that accomplishes it? But there

can be many algorithms all of which accomplish the exact same task. How can we choose among them? We can ask which ones are more efficient: Which use less memory ("space")? Which take less time? So, in addition to asking what can be automated, computer science also asks: What can be automated *efficiently*? If that is computer *engineering*, so be it, but that would put one of the most abstract, theoretical, and mathematical branches of computer science—namely, the theory of computational complexity—smack dab in the middle of computer engineering, and that doesn't seem correct.

5.5   A New Kind of Engineering?

One possible answer comes from the computer scientist Michael Loui [Loui, 1987]. According to Loui, computer science is *not* the study of computers: Contrary to Newell, Perlis, and Simon, not all phenomena give rise to a science of that phenomenon (there are toasters, but no science of toasters). In any case, any given computer will soon be outdated.

Nor is computer science a branch of math, because math is "declarative" (mathematical theorems are true propositions), whereas computer science is "procedural": Algorithms tell you *how to do* things. I don't think that this distinction is quite as clearcut as Loui would make it out. For one thing, one of the earliest and most successful branches of math—Euclid's geometry—was procedural! [Toussaint, 1993] A typical statement would be "To construct an angle bisector with compass and straightedge, do the following:". (And, just as there are theorems in computer science about what can*not* be computed, there are theorems in Euclidean geometry about what cannot be constructed using only compass and straightedge, such as an angle *tri*sector.)

Nor is computer science a branch of electrical engineering, because computers need not be electronic. There can be mechanical computers, and researchers are studying optical, quantum, and biological computers.[1] (However, as my colleague Peter Scott, formerly of our university's electrical and computer engineering department, pointed out to me, today's electrical engineering departments don't study just electronics, but also optics.)

Rather, according to Loui, computer science is the theory, design, analysis, and software- and hardware-implementation of algorithms for processing information, where, by 'processing', he includes storing, transforming, retrieving, and transmitting. Note that this is a very complicated definition, seemingly designed to make everyone happy: There's a little bit of engineering (design, hardware implementation), a little bit of science (theory, analysis), and some information thrown in for good measure.

But, for Loui, this means that computer science *is* engineering, but a new kind of engineering. Whereas classical engineering has a basis in science, computer science has a basis in math (as [Hartmanis, 1993]? suggested, perhaps computer science is mathematical engineering). Whereas classical engineering applies scientific principles to design physical objects, computer science designs, not only physical computers, but also non-physical programs. Finally, whereas classical engineering analyzes tradeoffs

---

[1]"The computing scientist could not care less about the specific technology that might be used to realize machines, be it electronics, optics, pneumatics, or magic" [Dijkstra, 1986], quoted in [Loui, 1987, 177].

such as gain vs. bandwidth (in electrical engineering) or stability vs. aesthetics (in civil engineering), the tradeoffs that computer science deals with are of a different kind: software vs. hardware, and space vs. time efficiency.

5.6 The Engineering Method

Just as there is a "scientific method", some scholars have proposed an "engineering method". We saw in Chapter 4 one view of the scientific method, according to which it is a loop that cycles through observation of facts, induction of general statements, deduction of future observations, and verification of the deduced predictions against observations, before cycling back to more observations.

Similarly, [Malpas, 2000] describes the engineering method both linearly and as a cycle. It begins by inputting a set of requirements, followed by analysis, then synthesis, then evaluation and execution, and outputting a solution. The cycle comes in between the input and the output: The evaluation and execution cycles back both to the analysis and to the synthesis, as well as adding to a knowledge base that, along with a set of resources, interact with the analysis, synthesis and evaluation-execution.

But neither this nor the scientific method are carved in stone; they are more like guidelines or even after-the-fact descriptions of behavior rather than rules that must be slavishly followed.

Whether computer science is a science or an engineering discipline, it has to be the science or engineering of something. We have seen at least two possibilities: It studies computers, or it studies algorithms. To further explore which of these might be central to computer science, let us begin by asking, "What is a computer?" Later, we will inquire into what an algorithm is.

5.7 NOTES FOR NEXT DRAFT

1. It is generally recognized...that engineering is 'the art or science of making practical application of the knowledge of pure sciences.'... The engineer uses the logic of science to achieve practical results." (pp. x-xi.) of Florman 1994

2. Here, in a mere 66 words, [former President Herbert] Hoover captured the essence of engineering and its contributions to society and culture":

   > Engineering "is a great profession. There is the fascination of watching a figment of the imagination emerge through the aid of science to a plan on paper. Then it moves to realization in stone or metal or energy. Then it brings jobs and homes to men. Then it elevates the standards of living and adds to the comforts of life. That is the engineer's high privilege." (p.304 of Petroski 2003)

3. Perhaps end this chapter, before going on to the other topics in PHICS, with Wulf's quote from §38

4. On contrasts between science and engineering:

> In a scientific discipline you discover how things work; in an engineering discipline you learn how to build things. [Abrahams, 1987, 472]

From this, Abrahams concludes that "Computer science is both a scientific discipline and an engineering discipline." The conclusion appears to be valid. Is the argument sound? The premises seem to be true. Yet one way to discover how things work is to try to build them; so, is all engineering a kind of science? But is the first premise really true? Is life, or the universe, a "thing"? Do scientists really try to learn how the kinds of physical objects that engineers build work (and nothing else)? This seems overly simplistic. Nevertheless, this "analytic vs. synthetic" distinction (that is, a distinction between analyzing—taking something apart—in order to learn how it works, on the one hand, and synthesizing—putting things together—in order to build something, on the other hand) seems to be a feature of many characterizations of science vs. engineering.

5. Characteristics of engineering:

   (a) Engineering emphasizes "what *can* exist", as opposed to science (especially physics), which studies what *does* exist [Loui, 1987, 176]. At first glance, this appears odd: One might think that the sciences are more concerned with abstract possibilities, whereas engineering would be more concerned with nitty gritty reality. But another way to understand this distinction is that the sciences are only concerned with things that currently exist, whereas engineering is (also) concerned with creating new things that *could*—but do not *yet*—exist.

   (b) "Computer science has all the significant aspects of engineering: a scientific basis, application of principles in design, analysis of trade-offs, and heuristics and techniques" [Loui, 1987, 176]. This suggests the following argument:

       i. Any engineering discipline has a scientific basis, etc.
       ii. Computer science has a scientific basis, etc.
       iii. ∴ computer science is an engineering discipline.

   But this is invalid, having the form "All *P*s are *Q*s; *c* is *Q*; ∴ *c* is *P*", which is a variety of the fallacy of affirming the consequent. (Compare: All dogs have 4 legs; my cat has 4 legs; ∴, my cat is a dog.) Loui's second "aspect" ties in with the first characteristic in our present list; he elaborates it as follows: "The ultimate goal of an engineering project is a product...that benefits society" [Loui, 1987, 176], giving bridges and computer programs as sample "products". But not all computer programs benefit society (think of computer viruses); presumably, Loui meant something like "product that

is intended to benefit society. But does that mean, then, that a computer programmer who writes a spreadsheet program is an engineer (no matter how sloppily the programmer writes it), whereas a computer programmer who writes a computer virus is not an engineer (even if the program was designed according to the best software engineering principles)?

6. Notes on [Koen, 1988]:

- Rather than defining 'engineering', Koen defines 'the engineering method'; presumably, just as 'science' can be defined as any discipline that follows "the scientific method", so 'engineering' can be defined as any discipline that follows "the engineering method". Here is his definition:

    The engineering method is the use of engineering heuristics to cause the best change in a poorly understood situation within the available resources [Koen, 1988, 308].

  He then spends a lot of time defining 'heuristics', which we'll turn to in a moment. What he doesn't seem to define is the adjective modifying 'heuristics', namely, 'engineering'! Presumably, the engineering method is more specific than merely the general use of heuristics. So, what makes a heuristic an *engineering* heuristic? Without this, his definition isn't very useful for us.

- On 'heuristic':

    A heuristic is anything that provides a plausible aid or direction in the solution of a problem but is in the final analysis ...incapable of justification and fallible. It is anything that is used to guide, discover and reveal a possible, but not necessarily, correct way to solve a problem. Though difficult to define, a heuristic has four characteristics that make it easy to recognize: it does not guarantee a solution; it may contradict other heuristics; it reduces the search time for solving a problem; and its acceptance depends on the immediate context instead of on an absolute standard. [Koen, 1988, 308].

- On this definition of 'heuristic', the first two characteristics differentiate the use of them from science and mathematics, according to Koen. So, they demarcate engineering from science and math. The third and fourht characteristics make their use more practical than at least some scientific or mathematical theories.

- In a somewhat puzzling passage on p. 309, Koen seems to suggest that engineering simply *is identical with* the use of heuristics, or, perhaps, with the use of heuristics for "causing desirable change in an unknown situation within the available resources". If this interpretation is correct, then part of what Koen is saying that what engineers do is to cause changes. This does contrast with science (and math), whose goal is, presumably, to understood

things, not to change them.[2]

- Much of what Koen says suggests that the kind of heuristic reasoning used by engineers is akin to what Herbert Simon called "bounded rationality" and "satisficing" (which we discussed briefly in §2.6.1.4): being satisfied with having a reasonable answer to a question rather than the "correct" one. Bounded rationality is necessary in practical situations, given limits ("bounds") on our time and knowledge.

7. According to the National Research Council's Committee on the Education and Utilization of the Engineer, engineering is, by their definition,

> Business, government, academic, or individual efforts in which knowledge of mathematical and/or natural sciences is employed in research, development, design, maufacturing, systems engineering, or technical opeations with the objective of creating and/or delivering systems, products, processes, and/or services of a technical nature and content intended for use. [Florman, 1994, 174–175]

Even Florman admits that this is a mouthful! Perhaps it can be simplified to something like this: Efforts in which math and natural science are used in order to produce something useful. If so, then engineering is (simply?) applied science.

And [Davis, 1996] points out that this definition, because of its vagueness (the overuse of 'and/or'), includes too much (such as accountants, because they use mathematics). He does say that it emphasizes three important "elements" of engineering: the centrality of math and science, the concern with the physical world (which might, therefore, rule out software; but see §12.3 on that topic), and the fact that "unlike science, engineering does not seek to understand the world but to remake it". But he goes on to say that "those three elements…do not define" engineering. So, it would seem that they are necessary but not sufficient conditions.

8. [Florman, 1994, 175] and many others [Davis, 1995a] also cite an 1828 definition due to one Thomas Tredgold: "Engineering is the art of directing the great sources of power in nature for the use and convenience of man [sic]".

9. Florman prefers this as his characterization, not of 'engineering', but of "the engineering view":

> a commitment to science and to the values that science demands— independence and originality, dissent and freedom and tolerance; a comfortable familiarity with the forces that prevail in the physical universe; a belief in hard work, not for its own sake, but in the quest for knowledge and understanding and in the pursuit of excellence; a willingness to forgo perfection, recognizing that we have to get real and useful products "out the door"; a willingness

---

[2]Compare Ch. 2, §2.6.4, where we saw Marx saying that philosophers should change the world, not merely understand it. Does that mean that Marx was proposing a discipline of "philosophical engineering"?

> to accept responsiblity and risk failure; a resolve to be dependable; a commitment to social order, along with a strong affinity for democracy; a seriousness that we hope will not become glumness; a passion for creativity, a compulsion to tinker, and a zest for change. [Florman, 1994, 183]

There are clear echoes of Koen's emphasis on the role of heuristics, a recognition of the foundational role of science, and—perhaps of most significance for our purposes—an emphasis on products.

So, if the central feature of engineering is, let's say, the application of scientific (and mathematical) techniques for producing or building something, then surely part of computer science is engineering: especially those parts that build computers and that write programs. Here's something to think about: Just as (some) computer scientists write programs, so journalists and novelists write essays. Are journalists and novelists therefore engineers? Their products are not typically application of science and math, so perhaps they aren't. But might they not be considered to be, say, language engineers?

10. Computer science is engineering, because: writing a computer program is giving instructions how to do something, that is, how to build something. On the other hand computer science could also be considered as the science of engineering, because it has theoretical aspects that describe what it means to give instructions for how to build something.

11.   The goal of engineering is the production of useful things, economically, for the benefit of society. [Loui, 1995, 31]

Among the "things" that (some) engineers produce are (chemical) processes, which are "similar to a computational algorithm, which is a method for processing information" [Loui, 1995, 31]. So, because computer science produces useful things (including algorithms, but also "microprocessors, software, and databases"), economically, for the benefit of society, it is a kind of engineering.

12. According to Michael Davis, scientists are primarily interested in "discovering new knowledge" and only secondarily, if at all, interested in "inventing something useful", whereas engineers are primarily interested in the latter and hardly at all in the former [Davis, 1995a, 42–43] (possibly repeated verbatim in [Davis, 1998]; CHECK THIS)

13. Question for the reader to think about: In Ch. 3, we saw that Brooks argued that computer science was not a science, but a branch of engineering, in part because the purpose of engineering is to build things, and that that's what computer scientists do. How would you evaluate this argument now that you have thought more deeply about what engineering is?

14. On software engineering:
In addition to the question of whether computer science is a kind of engineering,

there is the question of the nature of software engineering. Computer scientists (whether or not they consider themselves to be scientists or engineers) often consider software engineering as a branch of computer science: Courses in software engineering are often, perhaps even usually, taught in computer-science departments. [Parnas, 1998] argues that software engineering is a branch of engineering, not of computer science (nor of science more generally). In fact, he says, "Just as the scientific basis of electrical engineering is primarily physics, the scientific basis of software engineering is primarily computer science" [Parnas, 1998, 2].

There are two interesting implications of this. First, it suggests that Parnas views computer science as a science, because he takes it to be the scientific basis of a branch of engineering. Second, this view of things is inconsistent with the view advocated by, for instance, Loui and Hartmanis, who take computer science (or parts of it) as being a kind of engineering whose scientific basis is primarily mathematics! (On the other hand, one might argue that if software engineering is based on computer science, which, in turn, is based on mathematics, then software engineering must ultimately be based on mathematics, too. And that's not far from the truth, considering that much of formal software engineering (such as the formal analysis of computer programs and their development, or the use of program-verification methods in the development of programs; see, for example, [Mili et al., 1986] and our discussion of program verification in Ch. 16) is based on (discrete) mathematics and logic.

Parnas says that "engineers are professionals whose education prepares them to use mathematics, science, and the technology of the day, to build products that are important to the safety and well-being of the public" [Parnas, 1998, 2]. This echoes Davis's claim about the central role of education in the nature of being an engineer, as well as Brooks's (and others') claim about the purpose of engineering to build humanly useful things.

15. Even if computer science qua engineering is based on mathematics and logic as its underlying science, there is still a difference between more classic engineering disciplines and computer science, because, historically, computers (which, if anything, would seem to fall clearly on the engineering side of the spectrum) were developed—"engineered", if you will—without very much benefit of mathematical or logical analysis. It took von Neumann to see that the engineering accomplishments of Eckert and Mauchly (the developers of ENIAC) could be combined with the mathematical insights of Turing [Davis, 2012, Introduction]. We will look into this more closely when we investigate the historical answer to the question of what a computer is (in Ch. 6).

16.   **Engineering** is the knowledge required, and the process applied, to conceive, design, make, build, operate, sustain, recycle or retire, something of significant technical content for a specified purpose;— a concept, a model, a product, a device, a process a system, a technology. [Malpas, 2000, 31]

This is another definition-by-committee (note the lists of verbs and nouns). But it comes down to much the same thing as others have said: designing or building useful things. It emphasizes two aspects to this: One is that the design or activity of building must be knowledge-based; this presumably rules out design or building that is based, not on scientific knowledge, but on experience alone. The other aspect is that engineering is a process, in the sense of "knowing how" to do something [Malpas, 2000, 5]; this has an algorithmic flair—after all, algorithms are methods of describing how to do something.

17. [Petroski, 2005, 304] notes that we speak of "the sciences" in the plural (as we do of "the humanities"), but of engineering in the singular, "even though there are many" "engineerings". Moreover, "engineering...is an activity that creates things". Note two points: First, it is creative; this is related to claims about engineering as designing and building things. But, second, it is an activity, even grammatically: The word is a gerund, which "expresses...action". Is science also an activity (or is engineering different from science in this respect)? Insofar as science is an activity, is an activity that produces "knowledge". Engineering is an activity that uses that scientific knowledge to design and build artifacts.

18. [Petroski, 2006, 304] quotes former US president Herbert Hoover, who was an engineer by profession, as having "captured the essence of engineering" as beginning with...

> a figment of the imagination [that] emerge[s] through the aid of science to a plan on paper. Then it moves to realization in stone or metal or energy. Then it bring jobs and homes to men [sic]. Then it elevates the standards of living and adds to the comforts of life.

This rehearses the "design and build useful things" mantra of engineering, but it also highlights the idea of engineering as *implementing* an abstract idea. For more on implementation and its relationship to computer science, see Ch. 14.

19. In a follow-up article to [Davis, 1996], Michael Davis cites four different conceptions of engineering:

 (a) "engineering as tending engines", e.g., railroad engineers, building-superindent engineers; clearly, neither computer scientists nor software engineers are engineers in this sense, but neither are electrical, civil, mechanical, or chemical engineers engineers in this sense [Davis, 2011, 32];

 (b) engineers in "the functional sense, engineering-as-invention-of-useful-objects" [Davis, 2011, 32]. He criticizes this sense as both "too broad" (including architects and accountants) and "anachronistic" (applying to inventors of useful objects before 1700, which is about when the modern sense of 'engineer' came into use). Note that this seems to be the sense of engineering used by many who argue that computer science is engineering (who view engineering as designing and building useful artifacts).

(c) "engineering-as-discipline" [Davis, 2011, 32–33]. Here, the issue concerns "the body of knowledge engineers are supposed to learn", which includes "courses concerned with the material world, such as chemistry and statistics". Again, this would seem to rule out both computer science and software engineering, on the grounds that neither needs to know any of the "material" natural sciences like chemistry or physics (though both software engineers and computer scientists probably need some statistics) and both need "to know things other engineers do not".

(d) "engineering-as-profession" [Davis, 2011, 33]. This is Davis's favored sense, which he has argued for in his other writings. And here his explicit argument against software engineering (currently) being engineering (and his implicit argument against computer science (currently?) being engineering) is that, although both are professions, neither is (currently) part of the profession of engineering as it is taught and licensed in engineering schools. (Even computer science departments that are academically housed in engineering schools typically do not require their students to take "engineering" courses, nor are their academic programs accredited in the same way, nor are their graduates required to become "professional engineers" in any legal senses.)

Interestingly, he does suggest that software engineers, at least, might eventually become "real" engineers when "real" engineering starts becoming more computational [Davis, 2011, 34].

20. Consider the validity and soundness of the following arguments:[3]

(a) Argument 1:

   i. Engineers are cognitive agents that build artifacts for some identifiable purpose.
   ii. Birds build nests for housing their young.
   iii. Beavers build dams because the sound of rushing water annoys them.[4]
   iv. Computer engineers build computers for computation.[5]
   v. ∴ Birds, beavers, and computer engineers are all engineers.

(b) Argument 2:

   i. Engineers are cognitive agents that build artifacts for some identifiable purpose *and* who know what that purpose is.
   ii. Birds and beavers do not *know* why they build nests and dams, resepectively; they are only responding to biological or evolutionary instincts.
   iii. Computer *do* know what the purpose of computation is.
   iv. ∴ computer engineers are engineers, but birds and beavers are not.

---

[3]With thanks to Albert Goldfain.
[4]http://naturealmanac.com/archive/beaver_dams/beaver_dams.html (accessed 21 October 2013).
[5]We will discuss what this means later; see Chs. 6, 7, and 9.

21. On declarative propositions vs. procedural statements:
    [Goodman, 1987] argues that "constructive" algorithms, which describe how something can be done, are intensional, hence go beyond the standard extensional statements of ordinary mathematics. (On the difference between "intensional" and "extensional", see [Rapaport, 2009].) One reason is that we need to be able to say "what a program is *for*", because, in turn, "The point of discussin a particular algorithm is usually that it solves some problem that interests us" [Goodman, 1987, 476].

22. Mark Staples (personal communication, 2015) observes that Davis's definition of engineering in terms of its curriculum "is circular.... How does engineering knowledge become accepted into engineering curricula?"

    Also: Contra Petroski, engineering is more than just design, because architects design, but are not engineers.

    Also: "the key objective of engineering is to provide justified assurance (using theories and evidence from empirical test) that artefact are suitable for use (because they meet requirements specifications)."

    For a reply to Koen, see [Staples, 2014, §6.2].

    On their being no widely accepted definition of engineering, see [Pawley, 2009].

    > ...engineering research is only similar to science, and is not part of science (§4.1).

    The comment that follows may be more relevant to Ch. 4: Perhaps it would be useful to distinguish between a "narrow" and a "wide" notion of science: Science in the narrow sense is what questions about "what is science" are focused on. Science in the wide sense includes any logical, detailed, systematic study or investigation of some phenomenon. It is in the latter sense that engineering is "similar to science". And one way that it "is not part of science" is in its use of "theories that would not be acceptable within science" (§4.1), namely, theories that are "phenomenological, ...highly approximate, and have a highly limited scope" but are not "fundamental or explanatory, ...as precise as possible, ... [or] as universal as possible" (§4.1).

    > Unlike science, it is sufficient for engineering theories to be phenomenological. [Staples, 2015, §5]

    Presumably, it is necessary that scientific theories go beyond the merely phenomenological (cf. Mach?).

    Note:

> A phenomenological theory is one which predicts phenomena, but whose elements are not intended to correspond ontologically to real entities in the world. [Staples, 2015, §2.2]

> science tries to understand the world, whereas engineering tries to change it. (§4.1)

Compare Marx/Feuerbach on philosophy! (see §2.6.4). Computer science tries to do both: to understand the world computationally, and to change the world by building computational artifacts.

> ...engineering theories express claims that an artefact (represented in the theory by a design) will perform in a way that satisfies its requirements for use (represented in the theory by a requirements specification). (§8)

> Empirical theories in engineering concern not just the behaviour of artefacts, but also specifications of requirements for use, deigns fro artefacts, and claims that artefacts meet requirements for use. [Staples, 2015, §1].

> I define 'engineering theories' as explicit falsifiable claims used to predict and analyse the performance of artefacts with respect to requirements. [Staples, 2015, §2.1]

5.8 Further Sources of Information

1. Books and Essays:

   - Abelson, Harold, & Sussman, Gerald Jay, with Sussman, Julie (1996), "Preface to the First Edition", *Structure and Interpretation of Computer Programs*
     http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-7.html#%_chap_Temp_4
     accessed 5 November 2012.

   - Abrahams, Paul (1987), "What Is Computer Science?", *Communications of the ACM* 30(6) (June): 472–473.

   - Arden, Bruce W. (1980), "COSERS Overview", in Bruce W. Arden (ed.), *What Can Be Automated? The Computer Science and Engineering Research Study (COSERS)* (Cambridge, MA: MIT Press), Ch. 1, pp. 1–31; see esp. the section titled "About Names and Labels" (pp. 5–7).

   - Bunge, Mario (1974), "Towards a Philosophy of Technology", in Alex C. Michalos (ed.), *Philosophical Problems of Science and Technology* (Boston: Allyn & Bacon): 28–47.

   - Davis, Martin (2000), "Introduction", in Martin Davis, *The Universal Computer* (New York: W.W. Norton).

   - Davis, Michael (1995), "Questions for STS [Science & Technology Studies] from Engineering Ethics", talk given at the Society for the Social Study of Science, Charlottesville, VA, October 22, 1995,
     http://ethics.iit.edu/publication/Questions_for_STS.pdf
     accessed 5 November 2012.

   - Denning, Peter J. (2013), "Design Thinking", *Communications of the ACM* 56(12) (December): 29–31.

   - Denning, Peter J.; Comer, Douglas E.; Gries, David; Mulder, Michael C.; Tucker, Allen; Turner, A. Joe; & Young, Paul R. (1989), "Computing as a Discipline", *Communications of the ACM* 32(1) (January): 9–23.

   - Denning, Peter J.; & Riehle, Richard D. (2009), "Is Software Engineering Engineering?", *Communications of the ACM* 52(3) (March): 24–26.

   - Fiske, Edward B. (1989), "Between the 'Two Cultures': Finding a Place in the Curriculum for the Study of Technology", *New York Times* (29 March): B8; online at:
     http://www.nytimes.com/1989/03/29/us/education-lessons.html
     (accessed 9 September 2013)

   - Florman, Samuel C. (1994), *The Existential Pleasures of Engineering, 2nd edition* (New York: St. Martin's Press).

     – esp. Ch. 14, "The Engineering View", pp. 174–184.

   - Hartmanis, Juris, & Lin, Herbert (eds.?) (1992), "What Is Computer Science and Engineering?", in Juris Hartmanis & Herbert Lin (eds.), *Computing the Future: A Broader Agenda for Computer Science and Engineering* (Washington, DC: National Academy Press), Ch. 6, pp. 163–216.[6]

---

[6]This book was the subject of a petition to withdraw it from publication "because we consider it misleading and even harmful as an agenda for future research" (http://www-formal.stanford.edu/jmc/petition/whysign/whysign.html).
The petition was sponsored by Bob Boyer, John McCarthy, Jack Minker, John Mitchell, and Nils Nilsson. Commentaries on it appeared in [Kling et al., 1993].

- Hoare, C.A.R. (2009), "Retrospective: An Axiomatic Basis for Computer Programming", *Communications of the ACM* 52(10) (October): 30–32.

  – Has some interesting comments on the complementary nature of pure academic research (science) and applied industrial research (engineering).

- Kaag, John; & Bhatia, Sujata K. (2014), "Fools for Tools: Why Engineers Need to Become Philosophers", *The Chronicle [of Higher Education] Review* 61(13) (28 November): B13–B15,
  http://chronicle.texterity.com/chronicle/20141128b?folio=B13#pg13
  (accessed 30 December 2015).

  – An essay on engineering ethics.

- Koen, Billy Vaughn (2009), "The Engineering Method and Its Implications for Scientific, Philosophical, and Universal Methods", *The Monist* 92(3) (July): 357–386.

  – Koen seeks a "universal method" (not merely a scientific method or a method used in the humanities); he finds it in the "engineering method", which he identifies with heuristics. cf. Simon's notion of "satisficing", which we discussed in §2.6.1.4.

- Kroes, Peter (2010), "Engineering and the Dual Nature of Technical Artefacts", *Cambridge Journal of Economics* 34: 51–62.

  – Argues that "technical artefacts are physical structures with functional properties"

- Loui, Michael C. (1995), "Computer Science Is a New Engineering Discipline", *ACM Computing Surveys* 27(1) (March): 31–32.

- Malpas, Robert (2000), "The Universe of Engineering: A UK Perspective" (London: Royal Academy of Engineering)
  http://tinyurl.com/Malpas00 (accessed 5 November 2012).

- Petroski, Henry (2006), "A Great Profession", *American Scientist* 94 (July-August): 304–307.

- Petroski, Henry (2007), "Lab Notes", *American Scientist* 95(2) (March-April): 114–117.

  – How "a theoretician develop[ed] his applied side".

- Popper, Karl R. (1972), *Objective Knowledge: An Evolutionary Approach* (Oxford: Oxford University Press).

  – Parts of this book discuss the relation of engineering to science.

- Simons, Peter M.; & Michael, Sir Duncan (eds.) (2009), Special Issue on Philosophy and Engineering, *The Monist* 92(3) (July).

- Staples, Mark (2014), "Critical Rationalism and Engineering: Ontology", *Synthese* 191(10): 2255–2279,
  http://www.nicta.com.au/pub?doc=7397
  (preprint accessed 10 February 2015).

  – Presents a deductive view of theories in engineering, arguing that they "express claims that an artefact...will perform in a way that satisfies its requirements for use" (§8). Definitions of engineering are discussed in §2. The relation of engineering to science is discussed in §4.

- Staples, Mark (2015), "Critical Rationalism and Engineering: Methodology", *Synthese* 192(1): 337–362,
  http://www.nicta.com.au/pub?doc=7747
  (preprint accessed 11 February 2015).

  – A sequel to Staples 2014. Contains a useful "taxonomy" (§3) of ways in which artifacts can fail to conform to specifications (a topic that is also relevant to our later discussion of implementation, Ch. 14).

- Tedre, Matti (2009), "Computing as Engineering", *Journal of Universal Computer Science* 15(8): 1642–1658.

- Vincenti, Walter G. (1990), *What Engineers Know and How They Know It: Analytical Studies from Aeronautical History* (Baltimore: Johns Hopkins University Press).

  – Argues that engineering is a kind of knowledge that is different from scientific knowledge.

2. Websites:

Exercise for the reader: Link to various engineering websites, and try to find a definition of 'engineer' or 'engineering'!

- "What Is Engineering?", Whiting School of Engineering, Johns Hopkins University
  http://www.jhu.edu/ virtlab/index.php ().accessed 5 November 2012

- "What is engineering and what do engineers do?", National Academy of Engineering of the National Academies
  http://www.nae.edu/About/FAQ/20650.aspx (accessed 5 November 2012).

Chapter 6

What Is a Computer?
A Historical Perspective

"Let us now return to the analogy of the theoretical computing machines. . . It can be shown that a single special machine of that type can be made to do the work of all. It could in fact be made to work as a model of any other machine. The special machine may be called the universal machine. . . —Alan Turing (1947), cited in [Davis, 2012]

"If it should turn out that the basic logics of a machine designed for the numerical solution of differential equations coincide with the logics of a machine intended to make bills for a department store, I would regard this as the most amazing coincidence I have ever encountered." —Howard Aiken (1956), cited in [Davis, 2012]

"Many people think that computation is for figuring costs and charges in a grocery store or gas station." [Hill, 2008]

One of my daughters looked at a pile of MacBooks in the living room, and asked, "Can you hand me a computer?" (28 March 2015)

Figure 6.1: ©2012 by ??

6.1   Recommended Readings:

1. Highly Desired (if you have enough time to read two full-length books!):

   - Davis, Martin (2012), *The Universal Computer: The Road from Leibniz to Turing; Turing Centenary Edition*) (Boca Raton, FL: CRC Press/Taylor & Francis Group).
     - This will give you a good background in the "logical" history of computers, written by one of the leading mathematicians in the field of theory of computation.
     - At least try to read the Introduction (http://tinyurl.com/Davis00 accessed 5 November 2012), which is only one page long!
     - See Davis 1987, below. See also Papadimitriou 2001 and Dawson 2001 in "Further Sources of Information" for reviews of Davis 2012.
   - Aspray, William (ed.) (1990), *Computing before Computers* (Ames, IA: Iowa State University Press), http://ed-thelen.org/comp-hist/CBC.html (accessed 5 November 2012).
     - This will give you a good background in the "engineering" history of computers.

2. Required:

   - Browse the linked websites at "A Very Brief History of Computers" http://www.cse.buffalo.edu/∼rapaport/584/history.html (accessed 5 November 2012).
   - O'Connor, J.J., & Robertson, E.F. (1998), "Charles Babbage", http://www-gap.dcs.st-and.ac.uk/ history/Mathematicians/Babbage.html (accessed 5 November 2012).
   - Simon, Herbert A., & Newell, Allen (1958), "Heuristic Problem Solving: The Next Advance in Operations Research", *Operations Research* 6(1) (January-February): 1–10.
     - Read the brief history of Babbage's work (pp. 1–3); skim the rest.
     - In this paper, Simon and Newell predicted that (among other things) a computer would "be the world's chess champion" (p. 7) within 10 years, i.e., by 1968.[1]
     - For more on Adam Smith's pin-factory story (told in Simon & Newell 1958 and discussed below in §6.4.3), see:
       * Lawson, Robert (2004), "Division of Labour", http://www.divisionoflabour.com/archives/000006.php (accessed 5 November 2012).
         · Reprints the passage from Adam Smith on the pin factory.
       * Kennedy, Gavin (2008, 11 May), "The Talmud on the Division of Labour", *Adam Smith's Lost Legacy* (blog), http://tinyurl.com/p52fh6e[2] (accessed 5 November 2012).
       * Cowen, Tyler (2008), "Division of labor in the Babylonian Talmud", *Marginal Revolution* (blog), http://tinyurl.com/m7e6s34[3] (accessed 5 November 2012).

---

[1]I once asked Simon about this; our email conversation can be found at: http://www.cse.buffalo.edu/∼rapaport/584/S07/simon.txt (accessed 5 November 2012).

[2]http://adamsmithslostlegacy.blogspot.com/2008/05/talmud-on-dvision-of-labour.html

[3]http://marginalrevolution.com/marginalrevolution/2008/05/division-of-lab.html

- Davis, Martin (1987), "Mathematical Logic and the Origin of Modern Computers", *Studies in the History of Mathematics*, reprinted in Rolf Herken (ed.), *Universal Turing Machine: A Half-Century Survey; Second Edition* (Vienna: Springer-Verlag, 1995): 135–158; online at:
  http://staff.science.uva.nl/∼edoh/COMPHIS/davispaper.pdf
  (accessed 11 November 2013).

  – Article-length version of Davis 2012.

- Ensmenger, Nathan (2003), "Bits of History: Review of A.R. Burks's *Who Invented the Computer? The Legal Battle that Changed Computing History*", in *American Scientist* 91 (September-October): 467–468;
  http://www.americanscientist.org/bookshelf/pub/bits-of-history
  (accessed 30 October 2013).

3. Very Strongly Recommended:

- Robinson, J. Alan (1994), "Logic, Computers, Turing, and von Neumann", in K. Furukawa; D. Michie; & S. Muggleton (eds.), *Machine Intelligence 13: Machine Intelligence and Inductive Learning* (Oxford: Clarendon Press): 1–35.

  – A personal history of the development of computers and the related history of logic, by the developer of the resolution method of automated theorem proving.

- Browse through:

  – Computer History Museum, http://www.computerhistory.org/
    (accessed 5 November 2012).
  – Copeland, Jack (2000), "A Brief History of Computing",
    http://www.alanturing.net/turing_archive/pages/Reference%20Articles/BriefHistofComp.html
    (accessed 5 November 2012).
  – Lee, J.A.N. (1995–2002), "The History of Computing",
    http://ei.cs.vt.edu/∼history/ (accessed 5 November 2012).
  – Hitmill.com (2004), "History of Computers",
    http://www.hitmill.com/computers/computerhx1.html
    (accessed 5 November 2012).
  – Hoyle, Michelle A. (1994–2006), "The History of Computing Science"
    http://lecture.eingang.org/ (accessed 5 November 2012).

6.2   Introduction

Let us take stock of where we are. We began by asking what computer science is; we saw that it might be a science, a branch of engineering, a combination of both, or something *sui generis*; so we then investigated the nature of science and of engineering.

Now we want to turn to the question, "What is computer science the science (or engineering, or study) *of*"? There are at least three options to consider: The subject matter of computer science might be computers (the physical objects that compute), or it might be computing (the algorithmic processing that computers do), or it might be something else, such as the information that gets processed. In this and a subsequent chapter, we will examine the first two options. Readers interested in exploring the third option—information—are encouraged to look at some of the readings cited at the end of this chapter [OR AT END OF CH. 1? IN ANY CASE, SEPARATE THEM OUT, SO THAT THEY'RE EASIER TO FIND]

To help answer the question "What is a computer?", we will look first at the history of computers and then at some philosophical issues concerning the nature of computers.

Let's begin our historical investigation with some terminology. Some 120 years ago, in the May 2, 1892, issue of *The New York Times*, the following ad appeared:[4]

> *A COMPUTER WANTED.*
> WASHINGTON*, May 1.—A civil service ex-*
> *amination will be held May 18 in Washington,*
> *and, if necessary, in other cities, to secure*
> *eligibles for the position of computer in the*
> *Nautical Almanac Office, where two vacancies*
> *exist—one at $1,000, the other at $1,400.*
> *The examination will include the subjects of*
> *algebra, geometry, trigonometry, and as-*
> *tronomy. Application blanks may be obtained*
> *of the United States Civil Service Commission.*

So, a century or so ago, the answer to the question, "What is a computer?", was: a human who computes! In fact, until at least the 1940s (and probably the 1950s), that was the meaning of 'computer':

> Towards the close of the year 1850, the Author first formed the design of rectifying the Circle to upwards of 300 places of decimals. He was fully aware, at that time, that the accomplishment of his purpose would add little or nothing to his fame as a Mathematician, though it might as a Computer;... —William Shanks (1853), as cited in [Hayes, 2014a, 342]

When people wanted to talk about a *machine* that computed, they would use the phrase 'computing machine' or (later) 'electronic (digital) computer'. (Later on, when we look at Alan Turing's foundational papers in computer science and artificial intelligence, this distinction will be important.) Interestingly, nowadays when one wants to talk about a human who computes, we need to use the phrase 'human computer' [Pandya, 2013].

---

[4]The original may be seen online at [http://tinyurl.com/NYT-computer].

For the sake of familiarity, in this book, I will use the word 'computer' for the machine and the phrase 'human computer' for a human who computes.

6.3   Two Histories of Computers

There seem to be two histories of computers; they begin in parallel, but eventually converge and intersect.

The goal of one of these histories was to build a machine that could compute (or calculate), that is, a machine that could duplicate, and therefore assist, or even replace, or eventually supersede human computers. This is an engineering goal!

The goal of the other history was to provide a foundation for mathematics. This is a scientific (or, at least, a mathematical) goal!

These histories were probably never really parallel but more like a tangled web, with two "bridges" connecting them: the first was a person who lived about 330 years ago and the other was someone who was active only about 80 years ago—Gottfried Wilhelm Leibniz (1646–1716) and Alan Turing (1912–1954). (There were, of course, other significant people involved in both histories, as we will see.) Both histories begin in ancient Greece, the engineering history beginning with the need for computational help in navigation and the scientific history beginning with Aristotle's study of logic.

6.4   The Engineering History

The engineering history concerns the attempt to create machines that would do certain mathematical computations. The two main reasons for wanting to do this seem to be (1) to make life easier for humans (let a machine do the work) and—perhaps of more importance—(2) to produce computations that are more accurate (both more precise and with fewer errors) than those that humans produce. It is worth noting that the goal of having a machine perform an intellectual task that would otherwise be done by a human is one of the motivations underlying AI.

Here, I will only sketch some of the highlights of the engineering history. Articles such as [Chase, 1980], books such as [Aspray, 1990], and websites such as [Carlson et al., 1996] go into much more detail. And [Sloman, 2002, §2] argues that even the engineering history of computers has "two strands": the "development of machines for controlling physical mechanisms and [the] development of machines for performing abstract operations, e.g. on numbers."

6.4.1   Ancient Greece

The very early history of the attempt to build machines that could calculate can be traced back at least to the second century B.C.E., when a device now known as the Antikythera Mechanism was constructed [Freeth, 2006], [Wilford, 2006], [Seabrook, 2007], [Wilford, 2008], [Freeth, 2009]. This was a device used to calculate astronomical information, possibly for use in agriculture or religion. Although the Antikythera Mechanism was discovered in 1900, a full understanding of what it was and how it worked was not figured out until the 2000s.

6.4.2   17th Century Adding Machines

Skipping ahead almost 2000 years to about 350 years ago, two philosopher-mathematicians are credited with more familiar-looking calculators: Blaise Pascal (1623–1662), who helped develop the theory of probability, also invented an adding (and subtracting) machine that worked by means of a series of connected dials, and Leibniz (who invented calculus, almost simultaneously but independently of Isaac Newton) invented a machine that could add, subtract, multiply, and divide. As we'll see later on, Leibniz also contributed to the scientific history of computing with an idea for something he called a "calculus ratiocinator" (loosely translatable as a "reasoning system").

6.4.3   Babbage Machines

Two of the most famous antecedents of the modern electronic computer were due to the English mathematician Charles Babbage, who lived about 175 years ago (1791–1871). The first of the machines he developed was the Difference Engine (1821–1832). A French mathematician named Gaspard de Prony, who later headed France's civil-engineering college, needed to construct highly accurate, logarithmic and trigonometric tables for large numbers. He realized that it would take him too long using "difference equations" by hand.[5] In the 1790s, he had read Adam Smith's 1776 text on economics, *The Wealth of Nations*, which discussed the notion of the "division of labor": The manufacture of pins could be made more efficient by breaking the job down into smaller units, with each laborer who worked on one unit becoming an expert at his one job. This is essentially what modern computer programmers call "top-down design" and "stepwise refinement" [Wirth, 1971]: To accomplish some task T, analyze it into subtasks T_1, …, T_n, each of which should be easier to do than T. This technique can be repeated: Analyze each T_i into sub-subtasks T_{i_1}, …, T_{i_m}, and so on, until the smallest sub…subtask is so simple that it can be done without further instruction (this is the essence of "recursion"; see §7.7). De Prony applied this division of labor to computing the log and trig tables, using two groups of human computers, each as a check on the other. (For a discussion of this, written by two pioneers in computer science and AI, see [Simon and Newell, 1958]; see also [Pylyshyn, 1992].)

   Babbage wanted a *machine* to replace de Prony's *people*; this was his Difference Engine. He later conceived of an "Analytical Engine" (1834–1856), which was intended to be a general-purpose, problem solver (perhaps more closely related to Leibniz's goal for his calculus ratiocinator). Babbage was unable to completely build either machine: The techniques available to him in those days were simply not up to the precision required. However, he developed techniques for what we would today call "programming" these machines, using a 19th-century version of punched cards (based on a technique invented by Joseph Marie Jacquard for use in looms). Lady Ada Lovelace (1815–1852), daughter of the poet Lord Byron, working with Babbage, wrote a description of how to program the (yet-unbuilt) Analytical Engine; she is, thus, considered to be the first computer programmer. (But, for a more historically accurate

---

[5]Difference equations are a discrete-mathematical counterpart to differential equations. They involve taking successive differences of sequences of numbers.

discussion of the history of programming, see [Randell, 1994].) A partial model of the Difference Engine was finally built around 1991 [Swade, 1993], and efforts are underway to build a version of the Analytical Engine (http://www.plan28.org/).

6.4.4   Electronic Computers

The modern history of electronic, digital computers is rather confusing and the source of many historical and legal disputes. Here is a brief survey:

1. John Atanasoff (1903–1995) and his student Clifford Berry (1918–1963), working at Iowa State University, built the ABC (Atanasoff-Berry Computer) in 1937–1942. This may have been the first electronic, digital computer, but it was not a general-purpose (programmable) computer, and it was never completed. It was, however, the subject of a patent-infringement suit, about which more in a moment.

2. Konrad Zuse (1910–1995), in Germany, developed the Z3 computer in 1941, which was programmable.

3. Howard Aiken (1900–1973), influenced by Babbage, built the Harvard Mark I computer in 1944; it was designed to compute differential equations.

4. John Presper Eckert (1919-1995) and his student John Mauchly (1907–1980), working at the University of Pennsylvania, built the ENIAC (Electronic Numerical Integrator And Computer) in 1946. This was the first *general-purpose*—that is, programmable—electronic computer. In 1945, with the collaboration of the mathematician John von Neumann (1903–1957)—who outlined an architecture for computers that is still used today—they developed the EDVAC (Electronic Discrete Variable Automatic Computer), which used binary arithmetic (rather than decimal). This evolved into the first commercial computer: the UNIVAC (UNIVersal Automatic Computer). UNIVAC became famous for predicting, on national TV, the winner of the 1952 US presidential election. The company that made UNIVAC evolved into the Sperry Rand Corporation, who owned the patent rights. The Honeywell Corporation, a rival computer manufacturer, successfully sued Sperry Rand in 1973, on the grounds that Mauchly had visited Atanasoff in 1941, and that it was Atanasoff (and Berry), not Mauchly (and Eckert) who had "invented" the computer, thus vacating Sperry Rand's patent claims. (A useful summary of some of the issues involved can be found in [Ensmenger, 2003], who observes that identifying Atanasoff as "*the* inventor of *the* computer" (my phrasing and italics) is an "answer to what is fundamentally the *wrong question*" [Ensmenger, 2003, italics in original], because "any particular claim to priority of invention must necessarily be heavily qualified: If you add enough adjectives, you can always claim your own favorite". The story of von Neumann's involvement in the development of computers can be found in [Dyson, 2012b].)

5. There were two significant developments in the UK:

(a) In 1943, the Colossus computer was developed and installed at Bletchley Park for use in cryptography during World War II. Bletchley Park was where Alan Turing worked on cracking the Nazi's code-making machine, the Enigma.

(b) After the war, in 1945, Turing decided to try to implement his "*a*-machine" (the Turing machine), and developed the ACE (Automatic Computing Engine) [Copeland, 1999]. (It was also around this time that Turing started thinking about AI and neural networks.)

6.5   The Logical and Mathematical History

> Logic's dominant role in the invention of the modern computer is not widely appreciated. The computer as we know it today was invented by Turing in 1936, an event triggered by an important logical discovery announced by Kurt Gödel in 1930. Gödel's discovery...decisively affected the outcome of the so-called Hilbert Program. Hilbert's goal was to formalize all of mathematics and then give positive answers to three questions about the resulting formal system: is it consistent? is it complete? is it decidable? Gödel found that no sufficiently rich formal system of mathematics can be both consistent and comlete. In proving this, Gödel invented, and used, a high-level symbolic programming language: the formalism of primitive recursive functions. As part of his proof, he composed an elegant modular functional program.... This computational aspect of his work...is enough to have established Gödel as the first serious programmer in the modern sense. Gödel's computational example inspired Turing...[who] disposed of the third of Hilbert's questions by showing...that the formal system of mathematics is not decidable. Although his original computer was only an abstract logical concept,...Turing became a leader in the design, construction, and oepration of the first real computers. [Robinson, 1994, 6–7]

The parallel historical story concerns, not the construction of a physical device that could compute, but the logical and mathematical analysis of what computation itself is.

This story begins, perhaps, with Leibniz, who not only constructed a computing machine, as we have seen, but who also wanted to develop a "calculus ratiocinator": a formalism in a universally understood language (a "characteristica universalis") that would enable its "speakers" to precisely express any possible question and then to rationally calculate its answer. Leibniz's motto (in Latin) was: Calculemus! (Let us calculate!). In other words, he wanted to develop an algebra of thought.

This task was taken up around 160 years later (around 160 years ago) by the English mathematician George Boole (1815–1864), who developed an algebra of logic, which he called *The Laws of Thought* [Boole, 2009]. This was what is now called propositional logic. But it lacked a procedure for determining the truth value of a given (atomic) statement.

Boole's work was extended by the German mathematician Gottlob Frege (1848–1925, around 120 years ago), who developed what is now called first-order logic (or the

first-order predicate calculus).[6] Frege was a follower of a philosophy of mathematics called "logicism", which viewed mathematics as a branch of logic. Thus, to give a firm foundation for mathematics, it would be necessary to provide a system of logic that itself would need no foundation.

Unfortunately, the English philosopher Bertrand Russell (1872–1970, around 90 years ago), upon reading Frege's book *The Foundations of Arithmetic* in manuscript, discovered a problem. This problem, now known as Russell's Paradox, concerned the logic of sets: A set that had as members all only those sets that do not have themselves as members would both have itself as a member and not have itself as a member. This inconsistency in Frege's foundation for mathematics began a crisis that resulted in the creation of the theory of computation.[7]

That story continues with work done by the German mathematician David Hilbert (1862–1943, around 100 years ago), another logicist who wanted to set mathematics on a rigorous, logical foundation, one that would be satisfactory to all philosophers of mathematics, including intuitionists and finitists. (Intuitionists believe that mathematics is a construction of the human mind, and that any mathematical claim that can only be proved by showing that its assumption leads to a contradiction should not be accepted. Finitists believe that only mathematical objects constructible in a finite number of steps should be allowed into mathematics.) Hilbert proposed the following "Decision Problem" (in German: *Entscheidungsproblem*) for mathematics: to devise a procedure according to which it can be decided by a finite number of operations whether a given statement of first-order logic is a theorem.[8] A mathematical statement that was decidable in this way was also said to be "effectively computable" or "effectively calculable", because one could compute, or calculate, whether or not it

---

[6]None of these things called 'calculus' (plural: 'calculi') are related to the differential or integral calculus. 'Calculus' just means "system for calculation".

[7]An enjoyable graphic-novel treatment of the Russell-Frege story, written by a well-known computer scientist, is [Doxiadis et al., 2009].

[8]There are varying statements and translations of Hilbert's statements about the decision problem. Here are some:

> . . . determining whether or not a given formula of the predicate calculus is universally valid. [Hilbert and Ackermann, 1928, 112]

> In the broadest sense, the decision problem can be considered solved if we have a method which permits us to decide for any given formula *in which domains of individuals it is universally valid (or satisfiable) and in which it is not.* [Hilbert and Ackermann, 1928, 115]

> The Entscheidunsproblem is solved if one knows a procedure that allows one to decide the validity (respectively, satisfiability) of a given logical expression *by a finite number of operations.* (Translation in [Sieg, 1994, 77]; my italics.)

An earlier version (dating from 1900) appeared in Hilbert's list of 23 math problems that he thought should be investigated during the 20th century. The 10th problem was this:

> Given a diophantine equation with any number of unknown quantities and with rational integral numerical coefficients: *to devise a process according to which it can be determined by a finite number of operations whether the equation is solvable in rational integers.* (English translation from http://aleph0.clarku.edu/~djoyce/hilbert/problems.html#prob10 (accessed 11 December 2013.)

(Like the Halting Problem, this turns out to be non-computable; that is, there is no such process, no such algorithm.) For more on the decision problem, see §8.3.

was a theorem in a finite number of steps.[9]

(A brief digression on what a "theorem" is: When you studied geometry, you may have studied it in a modern interpretation of Euclid's original presentation of it as an axiomatic system. Most branches of mathematics (and, according to some philosophers, most branches of science) can be formulated axiomatically. One begins with a set of "axioms"; these are statements that are assumed to be true (or are considered to be so obviously true that their truth can be taken for granted). Then there are "rules of inference" that tell you how to logically infer other statements from the axioms in such a way that the inference procedure is "truth preserving"; that is, if the axioms are true (which they are, by assumption), then whatever logically follows from them according to the rules of inference are also true. (See Appendix. A for more details.) Such statements are called 'theorems'.)

Many mathematicians took up Hilbert's challenge: In the US, Alonzo Church (1903–1995) analyzed the notion of "function" and developed the lambda-calculus, claiming that any function whoses values could be computed in the lambda-calculus was effectively computable. The Austrian logician Kurt Gödel (1906–1978), who had previously proved the incompleteness of first-order logic (and thus became the most respected logician since Aristotle)—essentially, the claim that there were true arithmetic statements that could never be proved—developed the notion of "recursive" functions, claiming that this was co-extensive with effectively computable functions. Emil Post, a Polish-born, American logician (1897–1954), developed "production systems", also claimed to capture the notion of effective computability. And the Russian A.A. Markov (1856–1922) developed what are now known as Markov algorithms. (We will look in more detail at some of these systems in Ch. 7.)

But central to our story was the work of the English mathematician Alan Turing (1912–1954), who, rather than trying to develop a mathematical theory of effectively computable functions in the way that the others approached the subject, gave an analysis of what human computers did. Based on that analysis, he developed a formal, mathematical model of a human computer, which he called an "*a*-machine", and which we now call, in his honor, a Turing machine. In his classic paper published in 1936 [Turing, 1936], Turing presented his informal analysis of human computation, his formal definition of an *a*-machine, his claim that functions computable by *a*-machines were all and only the functions that were "effectively computable", a (negative) solution to Hilbert's Decision Problem (by showing that there was a mathematical problem that was *not* decidable computationally, namely, the Halting Problem), a demonstration that a single Turing machine (a "universal Turing machine") could do the work of all other Turing machines, *and*—as if all that were not enough—a proof that a function was computable by an *a*-machine if and only if it was computable in the lambda-calculus. (We will look at Turing's work in much greater detail in Ch. 8. For a very elementary introduction to the lambda-calculus, see [PolR, 2009, §"Alonzo Church's Lambda-Calculus"].)

Later, others proved that both methods were also logically equivalent to all of the others: recursive functions, production systems, Markov algorithms, etc. Because all of these theories had been proved to be logically equivalent, this finally convinced

---

[9]For more on this notion of "effectiveness", see §7.5.

almost everyone that the notion of "effective computability" (or "algorithm") had been captured precisely. Indeed, Gödel himself was not convinced until he read Turing's paper, because Turing's was the most intuitive presentation of them all. (But we will look at the arguments of those who are not convinced in Chs. 10 and 11.)

6.6   The Histories Converge

At this point, the engineering and mathematical histories converge: The development of programmable, electronic, digital computers—especially the EDVAC, with its von Neumann architecture—began to look like Turing machines, and Turing himself decided to implement a physical computer based on this architecture. (For more on this, see [Carpenter and Doran, 1977].) But Turing machines are hopelessly inefficient and cumbersome ("register" machines, another Turing-equivalent model of computation, are closer to modern computers):

> What is the difference between a Turing machine and the modern computer? It's the same as that between Hillary's ascent of Everest and the establishment of a Hilton hotel on its peak. (Alan J. Perlis, "Epigrams in Programming" [http://www.cs.yale.edu/homes/perlis-alan/quotes.html].)

To clarify some of this history, it will be necessary for us to look more closely at the nature of "effective computation" and "algorithms", which we will do in the next chapter. Armed with the results of that investigation, we will return to the question of what a computer is (from a philosophical point of view).

6.7   NOTES FOR NEXT DRAFT

1. Make Henkin 1962, required for next chapter, at least recommended here (or maybe required here and recommended in next chap)

2. Use these quotes from Hilbert; the first might be better in Ch. 7:

> It remains to discuss briefly what general requirements may be justly laid down for the solution of a mathematical problem. I should say first of all, this: that it shall be possible to establish the correctness of the solution by means of a finite number of steps based upon a finite number of hypotheses which are implied in the statement of the problem and which must always be exactly formulated. This requirement of logical deduction by means of a finite number of processes is simply the requirement of rigor in reasoning. [Hilbert, 1900, 440–441]

> Occasionally it happens that we seek...[a] solution [to a mathematical problem] under insufficient hypotheses or in an incorrect sense, and for this reason do not succeed. **The problem then arises: to show the impossibility of the solution under the**

given hypotheses, or in the sense contemplated. Such proofs of impossibility were effected by the ancients, for instance when they showed that the ratio of the hypotenuse to the side of an isosceles right triangle is irrational. In later mathematics, the question as to the impossibility of certain solutions plays a preeminent part, and we perceive in this way that old and difficult problems, such as the proof of the axiom of parallels, the squaring of the circle, or the solution of equations of the fifth degree by radicals have finally found fully satisfactory and rigorous solutions, although in another sense than that originally intended. It is probably this important fact along with other philosophical reasons that gives rise to **the conviction (which every mathematician shares, but which no one has as yet supported by a proof) that every definite mathematical problem must necessarily be susceptible of an exact settlement, either in the form of an actual answer to the question asked, or by the proof of the impossibility of its solution and therewith the necessary failure of all attempts.** Take any definite unsolved problem,... However unapproachable these problems may seem to us and however helpless we stand before them, we have, nevertheless, **the firm conviction that their solution must follow by a finite number of purely logical processes.** Is this **axiom of the solvability of every problem** a peculiarity characteristic of mathematical thought alone, or is it possibly a general law inherent in the nature of the mind, **that all questions which it asks must be answerable**? This conviction of the solvability of every mathematical problem is a powerful incentive to the worker. We hear within us the perpetual call: There is the problem. Seek its solution. You can find it by pure reason, for in mathematics there is no *ignorabimus*. [Hilbert, 1900, 444-445, my boldface][10]

3. If computers can be defined "historically", then they are "machines which (i) perform calculations with numbers, (ii) manipulate or process data (information), and (iii) control continuous processes or discrete devices... in real time or pseudo real time" [Davis, 1977, 1096]. Note that (ii) can be considered a generalization of (i), because performing calculations with numbers is a kind of manipulation of data, and because numbers themselves are a kind of data. (It's also the case that computers really manipulate *numerals*—that is, symbols that represent numbers—not (abstract) numbers themselves.) And, because being continuous or being discrete pretty much exhausts all possibilities, criterion (iii) doesn't really seem to add much. So this characterization comes down to (iii) alone: A computer is a machine that manipulates or processes data (information).

According to [Davis, 1977, 1096–1097], computers had evolved to have the following "key characteristics" (perhaps among others):

---

[10]For more on impossibility proofs in mathematics, see [Stewart, 2000]; cf. [Toussaint, 1993]. The most famous impossibility proof in computer science, of course, is the Halting Problem; see §7.8.

(a) "digital operation" (thus focusing on only the discrete aspect of (iii), above);

(b) "stored program capability" (here understood as "the notion that the instructions for the computer be written in the same form as the data being used by the computer" and attributed to von Neumann; we will return to this issue in Ch. 8);

(c) "self-regulatory or self-controlling capability" (this is not merely the automaticity of any machine, but it seems to include the ideas of feedback and "automatic modifiable stored programs");

(d) "automatic operation" (singled out from the previous characteristic because of its emphasis on operating "independently of human opertors and human intervention")

(e) "reliance on electronics" (this is admitted to be somewhat parochial in the sense that electronic computers were, at the time of writing, the dominant way of implementing them, but Davis recognized that other kinds of computers would eventually exist)

So, ignoring the last item and merging the previous two, we come down to this: A (modern) computer is an automatic, digital, stored-program machine (for manipulating information).

4. On numerals vs. numbers:
   Are computers symbol-manipulation machines or number-manipulations machines? There are two versions of this question. The first version contrasts numerical symbols (numerals) with numbers themselves. The second version contrasts symbols more generally with either numerals in particular or with numbers.

   The first question is closely related to the philosophy of mathematics. Is math itself more concerned with numerals than with numbers, or the other way around? "Formalists" and "nominalists" suggest that it is only the symbols for numbers that we really deal with. "Platonists" suggest that it is numbers that are our real concern, but at least some of them admit that the only way that we can directly manipulate numbers is via numerals (although some Platonists, including Gödel, suggest that we have a kind of perceptual ability, called 'intuition', that allows us to access numbers directly). There are also related questions about whether numbers exist and, if so, what they are. But these issues are beyond our scope. Computers, pretty clearly, have to deal with numbers via numerals. So, "The mathematicians and engineers then [in the 1950s] responsible for computers [who] insisted that computers only processed *numbers*—that the great thing was that instructions could be translated into numbers" [Newell, 1980, 137] are probably wrong. But even if we modify such a claim so that we replace numbers by numerals, we are faced with the second question above.

   An answer to that question will depend in part on how we interpret the symbols that a computer deals with. Certainly, there are ways to build computers

that, apparently, can deal with more than merely numerical symbols; the Lisp machines of the 1980s are prime examples: Their fundamental symbols were Lisp lists. But, insofar as any computer is ultimately constructed from physical switches that are either in an "on/up" or "off/down" position, we are left with a symbol system that is binary—hence numerical—in nature. Whether we consider these symbols to be numerals or not may be more a matter of taste or convenience than anything more metaphysical.

5. On Adam Smith's division of labor:
   It should be noted that, besides its positive effects on efficiency, the division of labor has negative ones, too: It "would make workers as 'stupid and ignoratn as it is possible for a human creature to be.' This was because no worker needed to know how to make a pin, only how to do his part in the process of making a pin" [Skidelsky, 2014, 35], quoting Adam Smith, in *The Wealth of Nations*, Book V, Ch. I, Part III, Article II (https://www.marxists.org/reference/archive/smith-adam/works/wealth-of-nations/book05/ch01c-2.htm)

6. Nice quote, possibly for end of §6.4.4:

   > *There are currently over one hundred computers installed in American universities.* Probably two dozen or more will be added this year. In 1955 the number was less than twenty-five. ... [C]onsidering the costs involved in obtaining, maintaining, and expanding these machines, the universities have done very well in acquiring hardware with their limited funds. [Perlis, 1962, 181, my italics]

   Of course, a university with, say, 5000 students now probably has at least 5000 computers, if not double that amount (by including smart phones), not to mention the computers owned by the universities themselves! And each one of those 10,000 computers is at least as powerful, if not more so, than the 100 that there were a half-century ago.

   > The basic purpose [of computers at universities], at present [that is, in 1962], is to do computations associated with and supported by university research programs, largely government financed. ... *Sad to state, some uses occur merely because the computer is available, and seem to have no higher purpose than that.* [Perlis, 1962, 182, my italics]

   And I wouldn't be surprised if most uses (Candy Crush?, Skype?, Facebook?, Twitter?, Amazon?) of the 10,000 computers at an average contemporary university "have no higher purpose"!

7. By the Entscheidungsproblem of a system of symbolic logic is here understood the problem to find an effective method by which, given any epression Q in the notation of the system, it can be determined whether or not Q is provable in the system. [Church, 1936a, 41n6]

8. On the Antikythera:
   Relevant to questions about whether everything computes anything (the Searle-Putnam puzzle), we can ask the following questions about the Antikythera, a real-life example of the "computer found in the desert": Does it compute? What does it compute? Is what it computes determined by its creators? Can *we* determine it?

9. The two separate histories of computer science are noted by [Mahoney and Haigh (ed.), 2011]:

   > . . . it is really only in von Neumann's collaboration with the ENIAC team that two quite separate historical strands came together: the effort to achieve high-speed, high-precision, automatic calculation and the effort to design a logic machine capable of significant reasoning.
   >
   > The dual nature of the computer is reflected in its dual origins: hardware in the sequence of devices that stretches from the Pascaline to the ENIAC, software in the series of investigations that reaches from Leibniz's combinatorics to Turing's abstract machines. Until the two strands come together in the computer, they belong to different histories. . . . (p. 26)

   He also emphasizes the fact that these histories "converged" but were not "coincident" (p. 88)

6.8 Further Sources of Information

FOR NEXT DRAFT: SEPARATE OUT THINGS LIKE ALL ARTICLES ON ADA, OR ON BABBAGE, AND SEPARATE BY ENGG HISTORY VS. LOGICAL HIST, ETC.

1. Books and Essays:

   - Baranger, Walter R. (1995), "J. Presper Eckert, Co-Inventor of Early Computer, Dies at 76", *New York Times* (7 June): B12; online at:
     http://www.nytimes.com/1995/06/07/obituaries/j-presper-eckert-co-inventor-of-early-computer-dies-at-76.html
     (accessed 11 November 2013).

   - Baranger, Walter R. (1995), "John V. Atanasoff, 91, Dies; Early Computer Researcher", *New York Times* (17 June): 11; online at:
     http://www.nytimes.com/1995/06/17/obituaries/john-v-atanasoff-91-dies-early-computer-researcher.html
     (accessed 11 November 2013).

   - Campbell-Kelly, Martin (2009), "Origin of Computing", *Scientific American* (September): 62–69; online at:
     http://www.cs.virginia.edu/∼robins/The_Origins_of_Computing.pdf
     (accessed 12 November 2013).

     - See also "Readers Respond on the 'Origin of Computing' ", *Scientific American* (January 2010): 8; online at:
       http://www.scientificamerican.com/article.cfm?id=letters-jan-10
       (accessed 12 November 2013).

   - Campbell-Kelly, Martin (2010), "Be Careful What You Wish For: Reflections on the Decline of Mathematical Tables", *Communications of the ACM* 53(4) (April): 25–26.

   - Campbell-Kelly, Martin (2011), "In Praise of 'Wilkes, Wheeler, and Gill' ", *Communications of the ACM* 54(9) (September): 25–27;
     http://dx.doi.org/10.1145/1995376.1995386

     - "Reflections on the first textbook on programming."

   - Campbell-Kelly, Martin (2012), "Alan Turing's Other Universal Machine", *Communications of the ACM* 55(7) (July): 31–33.

     - "Reflections on the Turing ACE computer and its influence."

   - Care, Charles (2007), "Not Only Digital: A Review of ACM's Early Involvement with Analog Computing Technology", *Communications of the ACM* 50(5) (May): 42–45; http://dx.doi.org/10.1145/1230819.1230837

   - Chaitin, Gregory J. (2002), "Computers, Paradoxes and the Foundations of Mathematics", *American Scientist* 90 (March-April): 164–171; online at:
     http://www.umcs.maine.edu/∼chaitin/amsci.pdf
     (accessed 12 November 2013).

   - Chase, George C. (1980), "History of Mechanical Computing Machinery", *Annals of the History of Computing* 2(3) (July): 198–226.

     - An illustrated history, with a useful introduction by I. Bernard Cohen.

- Collen, Morris F. (1994), "The Origins of Informatics", *Journal of the American Medical Informatics Association* 1(2) (March/April): 91–107.
    - A useful survey of the history of computer science aimed at practitioners of medical informatics.
- Copeland, B. Jack (2004), "Computation", in Luciano Floridi (ed.), *The Blackwell Guide to the Philosophy of Computing and Information* (Malden, MA: Blackwell).
    - Esp. pp. 3–4 on "The Birth of the Modern Computer"
- Copeland, Jack; Flowers, Thomas; & 17 Bletchley Park Codebreakers (2010), *Colossus: The Secrets of Bletchley Park's Codebreaking Computers* (New York: Oxford University Press); online at:
  http://www.colossus-computer.com/contents.htm
  (accessed 6 November 2012).
- Davis, Martin (2000), "Overheard in the Park", *American Scientist* 88 (July-August): 366–367.
    - Accessed 19 March 2014 from:
      http://www.americanscientist.org/bookshelf/pub/overheard-in-the-park
    - A somewhat negative review of David Berlinksi, *The Advent of the Algorithm* (Harcourt, 2000), correcting some of the historical errors in that book.
- Davis, Martin (2003), "Paradoxes in Paradise", *American Scientist* 91 (May-June): 268–269; online at:
  http://www.americanscientist.org/bookshelf/pub/paradoxes-in-paradise
  (accessed 12 November 2013).
    - Review of Marcus Giaquinto, *The Search for Certainty: A Philosophical Account of Foundations of Mathematics* (Oxford University Press, 2002).
- **Davis, Martin (2004), "The Myth of Hypercomputation", in C. Teuscher (ed.), *Alan Turing: The Life and Legacy of a Great Thinker* (Berlin: Springer): 195–212.**
    - Accessed 15 May 2014 from:
      http://www1.maths.leeds.ac.uk/∼pmt6sbc/docs/davis.myth.pdf
    - Early sections contain a good summary of the history of computation.
- Dawson, John W. (2001), Review of Davis 2012, *Bulletin of Symbolic Logic* 7(1) (March): 65–66; online at:
  http://www.math.ucla.edu/∼asl/bsl/0701/0701-003.ps
  (accessed 12 November 2013).
- Dyson, George (2012), *Turing's Cathedral: The Origins of the Digital Universe* (New York: Pantheon).
    - Holt, Jim (2012), "How the Computers Exploded" (review of Dyson 2012), *New York Review of Books* (7 June): 32–34.
    - Mauchly, Bill; Bernstein, Jeremy; Dowson, Mark; Adams, David K.; & Holt, Jim (2012, 27 September), "Who Gets Credit for the Computer?: An Exchange", *New York Review of Books*: 96, 98.
- Ensmenger, Nathan (2011), "Building Castles in the Air", *Communications of the ACM* 54(4) (April): 28–30; http://dx.doi.org/10.1145/1924421.1924432
    - "Reflections on recruiting and training programmers during the early period of computing."

- **Forster, E.M. (1909), "The Machine Stops"**
  http://archive.ncsa.illinois.edu/prajlich/forster.html (accessed 6 November 2012).

  - "Anybody who uses the Internet should read E.M. Forster's *The Machine Stops*. It is a chilling, short story masterpiece about the role of technology in our lives. Written in 1909, it's as relevant today as the day it was published. Forster has several prescient notions including instant messages (email!) and cinematophotes (machines that project visual images). —Paul Rajlich"
    (from the above-cited website)

- George, Alexander (1983), "Philosophy and the Birth of Computer Science", *Robotics Age* (January/February): 26–31.

- Goldstine, Herman H. (1972), *The Computer from Pascal to von Neumann* (Princeton, NJ: Princeton University Press).

- Green, Christopher D. (2005), "Was Babbage's Analytical Engine Intended to Be a Mechanical Model of the Mind?", *History of Psychology* 8(1): 35–45,
  http://www.yorku.ca/christo/papers/babbage-HoP.pdf
  (accessed 5 November 2012).

  - Briefly, no (at least, not by Babbage).

- Grier, David Alan (2005), *When Computers Were Human* (Princeton, NJ: Princeton University Press).

  - Skinner, David (2006), "The Age of Female Computers", review of Grier 2005, *The New Atlantis* (Spring): 96–103

    * Accessed 21 March 2014 from:
      http://www.thenewatlantis.com/publications/the-age-of-female-computers
      and http://www.thenewatlantis.com/docLib/TNA12-Skinner.pdf

- Hafner, Katie (2002), "Happy Birthday :-) to You: A Smiley Face Turns 20", *New York Times* (19 September): G4; online at:
  http://www.nytimes.com/2002/09/19/technology/typographic-milestones-happy-birthday-to-you-a-smiley-face-turns-20.html
  (accessed 12 November 2013).

- Haigh, Thomas (2014), "Actually, Turing Did Not Invent the Computer", *Communications of the ACM* 57(1) (January): 36–41.

  - Discusses the (ir)relevance of the mathematical history of computation to the engineering history.

- Halmos, Paul R. (1973), "The Legend of John von Neumann", *American Mathematical Monthly* (April): 382–394.

  - A very readable, short biography, with a heavy emphasis on the humorous legends that have grown up around von Neumann.

- Hayes, Brian (1994), "The World Wide Web", *American Scientist* 82 (September-October): 416–420; online at:
  http://bit-player.org/wp-content/extras/bph-publications/AmSci-1994-09-Hayes-www.pdf
  (accessed 11 November 2013).

- Hayes, Brian (2000), "The Nerds Have Won", *American Scientist* 88 (May-June): 200–204; onine at:

http://www.americanscientist.org/issues/pub/the-nerds-have-won
(accessed 12 November 2013).

  – On the history of the Internet.

- Holt, Jim (2001), "The Ada Perplex", *The New Yorker* (5 March): 88–93.
  – "How Byron's daughter came to be celebrated as a cyber-visionary."

- Husbands, Philip; Wheeler, Michael; & Holland, Owen (2008), "Introduction: The Mechanical Mind", in Philip Husbands, Owen Holland, & Michael Wheeler (eds.), *The Mechanical Mind in History* (Cambridge, MA: MIT Press): 1–17.

- Hyman, Anthony (1982), *Charles Babbage: Pioneer of the Computer* (Princeton: Princeton University Press).
  – Reviewed in:
    O'Hanlon, Redmond (1982), "A Calculating Man", *New York Times Book Review* (14 November): BR26–BR27.

- Hyman, Paul (2012), "Lost and Found", *Communications of the ACM* 55(7) (July): 21.
  – "Researchers discover computer pioneer Konrad Zuse's long-forgotten Z9, the world's first program-controlled binary relay calculator using floating-point arithmetic."
  – See also:
    Winkler, Jürgen F.H. (2012), "Konrad Zuse and Floating-Point Numbers", letter to the editor, *Communications of the ACM* 55(10) (October): 6–7.

- Johnson, George (2001), "Claude Shannon, Mathematician, Dies at 84", *New York Times* (27 February): B7; online at:
  http://www.nytimes.com/2001/02/27/nyregion/27SHAN.html
  (accessed 12 November 2013).

- Johnson, Mark (2002), Review of Davis 2012, *MAA Reviews*
  http://mathdl.maa.org/mathDL/19/?pa=reviews&sa=viewBook&bookId=68938
  (accessed 5 November 2012).

- Keats, Jonathan (2009), "The Mechanical Loom", in "The Start of Everything", *Scientific American* (September): 88; online at:
  http://www.readcube.com/articles/10.1038/scientificamerican0909-88
  (accessed 30 October 2013).
  – The story of the Jacquard loom.

- Kernan, Michael (1990), "The Object at Hand", *Smithsonian* 21 (May): 22, 24, 26.
  – About the Nazi's code-making machine, the Enigma.

- Kim, Eugene Eric; & Toole, Betty Alexandra (1999), "Ada and the first Computer", *Scientific American* (May): 76–81; online at:
  http://people.cs.kuleuven.be/∼danny.deschreye/AdaTheFirst.pdf
  (accessed 12 November 2013).

- Kleene, Stephen C. (1981), "Origins of Recursive Function Theory", *Annals of the History of Computing* 3(1) (January): 52–67.

- Levy, Steven (2013), "The Brief History of the ENIAC Computer", 101 Objects that Made America, *Smithsonian* (November): 62–64; online at:
  http://www.smithsonianmag.com/history-archaeology/The-Brief-History-of-the-ENIAC-Computer-228879421.html
  (accessed 9 November 2013).

- Lohr, Steve (1996), "The Face of Computing 50 Years and 18,000 Tubes Ago",
  New York Times (19 February): D3; online at:
  http://www.nytimes.com/1996/02/19/business/the-face-of-computing-50-years-
  and-18000-tubes-ago.html
  (accessed 12 November 2013).

- Lohr, Steve (2001), "Frances E. Holberton, 84, Early Computer Programmer", *New
  York Times* (17 December): F5; online at:
  http://www.nytimes.com/2001/12/17/business/frances-e-holberton-84-early-
  computer-programmer.html (accessed 12 November 2013).

  – "Mrs. Holberton. . . was one of the six young women recruited by the Army to
    program. . . Eniac."

- Lohr, Steve (2002), "Scientist at Work: Frances Allen; Would-Be Math Teacher
  Ended Up Educating a Computer Revolution", *New York Times* (6 August): F3;
  http://tinyurl.com/mqhbpey[11] (accessed 12 November 2013).

- Lohr, Steve (2010), "Inventor Whose Pioneer PC Helped Inspire Microsoft Dies",
  New York Times (3 April): A1–A3.

  – Obituary of H. Edward Roberts, inventor of the Altair, the first personal
    computer.

- MacFarlane, Alistair (2013), "Ada Lovelace (1815–1852)", *Philosophy Now* 96;
  online at: http://philosophynow.org/issues/96/Ada_Lovelace_1815-1852
  (accessed 30 October 2013).

- Mahoney, Michael Sean (2011), *Histories of Computing*, ed. by Thomas Haigh
  (Cambridge, MA: Harvard University Press).

  – Not so much a history of computing or computers as a history of computer
    science. Chs. 10 and 11 are especially good on some of the recent
    mathematical history.

- Markoff, John (1992), "Rear Adm. Grace M. Hopper Dies; Innovator in Computers
  Was 85", *New York Times* (3 January); online at:
  http://tinyurl.com/lthvdy4[12] (accessed 11 November 2013).

- Markoff, John (2000), "A Tale of the Tape from the Days When It Was Still Micro
  Soft", *New York Times* (18 September): C1, C4; online at:
  http://tinyurl.com/ox9upbh[13] (accessed 12 November 2013).

  – About "the day someone stole Bill Gates's software".

- Markoff, John (2002), "Edsger Dijkstra, 72, PHysicist Who Shaped Computer Era",
  New York Times (10 August): A11; online at:
  http://www.nytimes.com/2002/08/10/obituaries/10DIJK.html
  (accessed 12 November 2013).

- Markoff, John (2005), *What the Dormouse Said: How the Sixties Counterculture
  Shaped the Personal Computer Industry* (New York: Viking).

---

[11]http://www.nytimes.com/2002/08/06/science/scientist-work-frances-allen-would-be-math-teacher-
ended-up-educating-computer.html

[12]http://www.nytimes.com/1992/01/03/us/rear-adm-grace-m-hopper-dies-innovator-in-computers-was-
85.html

[13]http://www.nytimes.com/2000/09/18/business/technology-a-tale-of-the-tape-from-the-days-when-it-
was-still-micro-soft.html

- – Reviewed in [Lanier, 2005].

- Markoff, John (2011), "It Started Digital Wheels Turning", *New York Times* Science Times (8 November): D1, D4; online at:
  http://tinyurl.com/dxqt6ca[14] (accessed 12 November 2013).

  - – "In the age of steam,...[Charles Babbage] brainstormed a contraption that inspired dreams of computers. Experts in London intend to spend a decade piecing it together." Article contains a diagram of the Analytical Engine, online at:
    http://www.nytimes.com/interactive/2011/11/07/science/before-its-time-machine.html
    (accessed 12 November 2013).

- Martin, C. Dianne (1993), "The Myth of the Awesome Thinking Machine", *Communications of the ACM* 36(4) (April): 120–133;
  http://dx.doi.org/10.1145/255950.153587

  - – "[T]he thesis of this article [is] that early public attitudes toward computers were shaped by the press."

- Martin, Douglas (2008), "David Caminer, 92, Dies; A Pioneer in Computers", *New York Times* (29 June): 28;
  http://www.nytimes.com/2008/06/29/technology/29caminer.html
  (accessed 9 December 2013).

  - – Caminer worked for the first company to computerize its operations—the Lyons tea company—in 1951(!).

- Martin, Douglas (2013), "Mavis Batey, 92, Allied Code Breaker in World War II", *New York Times* (24 November): 32;
  http://www.nytimes.com/2013/11/23/world/europe/mavis-batey-world-war-ii-code-breaker-dies-at-92.html
  (accessed 9 December 2013).

  - – Batey worked with Turing at Bletchley Park.

- Mathews, Walter M.; & Reifers, Kaye (1984), "The Computer in Cartoons: A Retrospective from *The Saturday Review*", *Communications of the ACM* 27(11) (November): 1114–1119.

- O'Regan, Gerard (2008), *A Brief History of Computing* (Springer).

- Papadimitriou, Christos H. (2001), "The Sheer Logic of IT", review of [Davis, 2012] (2000 original edition), *American Scientist* 89 (March-April): 168–171; online at:
  http://www.americanscientist.org/bookshelf/pub/the-sheer-logic-of-it
  (accessed 11 November 2013).

- Park, Edwards (1996), "The Object at Hand", *Smithsonian* 26(11) (February): 20–23.

  - – "The incredible world of computers was born some 150 years ago, with a clunky machine dreamed up by a calculating genius named Charles Babbage."

---

[14]http://www.nytimes.com/2011/11/08/science/computer-experts-building-1830s-babbage-analytical-engine.html

- Press, Larry (1993), "Before the Altair: The History of Personal Computing", *Communications of the ACM* 36(9) (September): 27–33; online at: http://bpastudio.csudh.edu/fac/lpress/articles/hist.htm (accessed 11 November 2013).

- Robinson, J. Alan (1994), "Logic, Computers, Turing, and von Neumann", in K. Furukawa, D. Michie, & S. Muggleton (eds.), *Machine Intelligence 13: Machine Intelligence and Inductive Learning* (Oxford: Clarendon Press): 1–35.
  - Historical comments, by the developer of the resolution method of automated theorem proving, on the development of computers and the related history of logic.

- Ryan, Bob (1991), "Dynabook Revisited with Alan Kay", *Byte* 16(2) (February): 203–204, 206–208.
  - "Is the reign of the desktop computer about to end?" A prediction of the future of what is now known as laptop computers.

- Sammet, Jean E. (1992), "Farewell to Grace Hopper—End of an Era!", *Communications of the ACM* 35(4) (April): 128–131; http://dx.doi.org/10.1145/129852.214846

- Samuel, Arthur L. (1953), "Computing Bit by Bit, or Digital Computers Made Easy", *Proceedings of the IRE* [Institute of Radio Engineers] 41(10) (October): 1223–1230.
  - An introduction to computers, aimed at (radio) engineers who might not be familiar with them; written by an IBM researcher who later became famous for his work on computer checkers-players.

- Shapiro, Stewart (1983), "Remarks on the Development of Computability", *History and Philosophy of Logic* 4: 203–220.
  - On the mathematical history of computing.

- Sieg, Wilfried (1994), "Mechanical Procedures and Mathematical Experience", in Alexander George (ed.), *Mathematics and Mind* (New York: Oxford University Press): 71–117; http://repository.cmu.edu/cgi/viewcontent.cgi?article=1248&context=philosophy (accessed 10 December 2013).
  - See §1 for remarks on the mathematical history of computation.

- Smith, Adam (1776), "Of the Division of Labor", Book I, Ch. I, of *The Wealth of Nations*, online at: http://www.econlib.org/library/Smith/smWNCover.html (from that page, link to "I.1. Of the Division of Labor") (accessed 5 November 2012).

- Stein, Dorothy (1985), *Ada: A Life and a Legacy* (Cambridge, MA: MIT Press).
  - See also:
    Kidder, Tracy (1985), "Less (and More) than Meets the Eye", review of Stein 1985, *New York Times Book Review* (29 December): 6–7; online at: http://www.nytimes.com/1985/12/29/books/less-and-more-than-meets-the-eye.html (accessed 11 November 2013).

- Stoll, Cliff (2006), "When Slide Rules Ruled", *Scientific American* 294(5) (May): 80–87.

- – "Before electronic calculators, the mechanical slide rule dominated scientific and engineering computation."
  - – Note: The slide rule is an *analog* calculator!

- Vardi, Moshe Y. (2011), "Solving the Unsolvable", *Communications of the ACM* 54(7) (July): 5; online at:
  http://www.inf.unibz.it/∼calvanese/teaching/tc/material/solving-the-unsolvable-CACM-2011-07.pdf
  (accessed 12 November 2013).

- Vardi, Moshe Y. (2013), "Who Begat Computing?", *Communications of the ACM* 56(1) (January): 5; online at:
  http://cacm.acm.org/magazines/2013/1/158780-who-begat-computing/fulltext
  (accessed 13 November 2013).

  - – For a reply, see:
    Copeland, Jack (2013), "What Apple and Microsoft Owe to Turing", *Huff[ington] Post Tech/The Blog* (12 August),
    http://www.huffingtonpost.com/jack-copeland/what-apple-and-microsoft-_b_3742114.html
    (accessed 13 November 2013).
  - – For further discussion of what a "stored-program computer" is, and whether it was Turing or von Neumann (or someone else) who invented it, see Ch. 8.

- von Neumann, John (1945), "First Draft of a Report on the EDVAC", ed. by Michael D. Godfrey, in *IEEE Annals of the History of Computing* 15(4) (1993): 27–75; online at:
  http://virtualtravelog.net.s115267.gridserver.com/wp/wp-content/media/2003-08-TheFirstDraft.pdf (accessed 30 October 2013).

- Waldrop, M. Mitchell (2001), "The Origins of Personal Computing", *Scientific American* (December): 84–91.

- Wheeler, David L. (1997), "An Ancient Feud: Who Invented the Computer?", *Chronicle of Higher Education* (3 October): B2.

  - – On the ENIAC-ABC controversy, with a discussion of an attempt to replicate the ABC.

- Wright, Alex (2008), "The Web Time Forgot", *New York Times* (17 June): F1, F4; online at:
  http://www.nytimes.com/2008/06/17/science/17mund.html
  (accessed 11 November 2013).

  - – A 1934(!) version of a World Wide Web.

2. Websites:

- Three pictures showing the evolution, from the ENIAC to the iPad, of computers and their users: http://www.cse.buffalo.edu/∼rapaport/111F04/summary.html (accessed 5 November 2012).

- On Charles Babbage:

  (a) Charles Babbage Institute, Center for the History of Information Technology, University of Minnesota, http://www.cbi.umn.edu/ (accessed 5 November 2012).

(b) Greenemeier, Larry (2008, April 24), "150-Year-Old Computer Brought to Life [Slideshow]",
http://www.scientificamerican.com/article.cfm?id=150-year-old-computer-babbage (accessed 5 November 2012).

(c) Johnstone, Adrian (2014), "Babbage's Language of Thought", *Plan 28 Blog* (28 November),
http://blog.plan28.org/2014/11/babbages-language-of-thought.html
(accessed 5 January 2015).

   i. From a website devoted to "Charles Babbage's many inventions".
   ii. "Plan 28 is working towards building Charles Babbage's Analytical Engine"
   (http://plan28.org/, accessed 5 January 2015)

(d) Lee, J.A.N. (1994), "Charles Babbage"
http://ei.cs.vt.edu/∼history/Babbage.html (accessed 5 November 2012).

(e) Sydell, Laura (2009, December 10), "A 19th-Century Mathematician Finally Proves Himself",
http://www.npr.org/templates/story/story.php?storyId=121206408&ft=1&f=1001
(accessed 5 November 2012).

(f) Also see [Swade, 1993], Green 2005, and Campbell-Kelly 2010, above.

- "Computer Programming Used to Be Women's Work", *Smart News Blog*,
http://blogs.smithsonianmag.com/smartnews/2013/10/computer-programming-used-to-be-womens-work/
(accessed 9 November 2013).

- "Generations of Computers" (blog),
http://generationsofcomputers.blogspot.com/2007/12/generation-of-computers.html (accessed 5 November 2012).

   – Contains some good images of early calculating machines, including Pascal's and Leibniz's.

- IBM "Antique Attic" (three-"volume", illustrated exhibit of computing artifacts); links located at: http://www-03.ibm.com/ibm/history/exhibits/index.html
(accessed 5 November 2012).

- Joyce, David E. (1997), "The *Mathematical Problems* of David Hilbert"
http://aleph0.clarku.edu/∼djoyce/hilbert/ (accessed 5 November 2012).

- O'Connor, J.J.; & Robertson, E.F. (2012), The MacTutor History of Mathematics Archive, http://www-gap.dcs.st-and.ac.uk/∼history/ (accessed 6 November 2012), including:

   – O'Connor, J.J.; & Robertson, E.F. (1997), "Gaspard Clair Fran cois Marie Riche de Prony",
   http://www-gap.dcs.st-and.ac.uk/∼history/Mathematicians/De_Prony.html
   (accessed 6 November 2012).

and O'Connor & Robinson 1998, above.

- Sale, Tony, "The Colossus Rebuild Project",
http://www.codesandciphers.org.uk/lorenz/rebuild.htm
(accessed 30 October 2013)

- Seabrook, John (2007), "Pieces of History" (slideshow)
http://www.newyorker.com/online/2007/05/14/slideshow_070514_antikythera
(accessed 6 November 2012).

- Standage, Tom (1998), *The Victorian Internet: The Remarkable Story of the Telegraph and the Nineteenth Century's On-Line Pioneers* (New York: Walker Publishing).

  – See review by John R. Alden in *Smithsonian* 29(10) (January 1999): 126–127; online at:
    http://www.smithsonianmag.com/history-archaeology/bookreview_jan99_a.html (accessed 12 November 2013).

- Tout, Nigel (2000–2012), "Vintage Calculators Web Museum" http://www.vintagecalculators.com/ (accessed 6 November 2012).

- Wells, Benjamin (2003), "The Architecture of Colossus, the First PC" (abstract) http://www.stanford.edu/class/ee380/Abstracts/040204.html (accessed 6 November 2012).

  – See also Copeland et al. 2010, above.

Figure 6.2: ©2011, Hilary B. Price

Chapter 7

What Is an Algorithm?

Thou must learne the Alphabet, to wit, the order of the Letters as they stand.... Nowe if the word, which thou art desirous to finde, begin with (a) then looke in the beginning of this Table, but if with (v) looke towards the end. Againe, if thy word beginne with (ca) looke in the beginning of the letter (c) but if with (cu) then looke toward the end of that letter. And so of all the rest. &c. —Robert Cawdrey, *A Table Alphabeticall, conteyning and teaching the true writing, and understanding of hard usuall English wordes* (1604), cited in [Gleick, 2008, 78].

Editor:
We are making this communication intentionally short to leave as much room as possible for the answers. 1. Please define "Algorithm." 2. Please define "Formula." 3. Please state the difference. T. WANGSNESS, J. FRANKLIN
TRW Systems, Redondo Beach, California
[Wangsness and Franklin, 1966].

This nation is built on the notion that the rules restrain our behavior.... [Editorial, 2006]

Figure 7.1: ©2004 by ??

7.1   Recommended Readings:

1. Required:

   - Henkin, Leon (1962), "Are Logic and Mathematics Identical?", *Science* 138(3542) (November 16): 788–794.

     – Read pp. 788–791; skim the rest

     – An excellent, brief overview of the history of logic and the foundations of mathematics that led up to Turing's analysis.

   - Davis, Martin (1987), "Mathematical Logic and the Origin of Modern Computers", *Studies in the History of Mathematics*; reprinted in Rolf Herken (ed.), *Universal Turing Machine: A Half-Century Survey; Second Edition* (Vienna: Springer-Verlag, 1995): 135–158.

     – Overlaps and extends Henkin's history, and provides a useful summary of [Turing, 1936].

   - Herman, Gabor T. (1993), "Algorithms, Theory of", in Anthony S. Ralston & Edwin D. Riley (eds.), *Encyclopedia of Computer Science, 3rd edition* (New York: Van Nostrand Reinhold): 37–39.

     – Discussion
       of the informal notions of "algorithm" and "effective computability"; good background for Turing 1936.

2. Very Strongly Recommended:

   - *Either*:

     Soare, Robert I. (1996), "Computability and Recursion", *Bulletin of Symbolic Logic* 2(3) (September): 284–321.

     or:

     Soare, Robert I. (1999), "The History and Concept of Computability", in E.R. Griffor (ed.), *Handbook of Computability Theory* (Amsterdam: Elsevier): 3–36.

     – "a revised and shortened form" of Soare 1996.

     – Read §§1–3, 4.5–4.6; skim the rest

3. Strongly Recommended:

   - Browse through the "Examples of Algorithms" at:
     http://www.cse.buffalo.edu/~rapaport/584/whatisanalg.html
     (accessed 6 November 2012).

   - Haugeland, John (1981), "Semantic Engines: An Introduction to Mind Design", in John Haugeland (ed.), *Mind Design: Philosophy, Psychology, Artificial Intelligence* (Cambridge, MA: MIT Press): 95–128.

     – A good description of the syntax and semantics of formal systems and their relationship to Turing Machines.

   - Boehm, C.; & Jacopini, G. (1966), "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules", *Communications of the ACM* 9(5): 366–371.

– Uses flowcharts to prove that "go to" statements are eliminable from computer programs in favor of sequence, selection, and loops. An important paper, but not for the faint of heart!

7.2   Introduction

We are examining the questions of whether computer science is a science (or something else, such as a branch of engineering or some combination of both science and engineering) and what its subject matter is: Does it study comput*ers*: (physical) devices that compute—or comput*ing*: the algorithmic processes that computers do? (Or perhaps it studies something else, such as information, or information processing.) In the former case, we need to ask what *computers* are; in the previous chapter, we began that investigation by looking at the history of computers. In this chapter, we ask what *computing* is. Then we will be in a better position to return to our question of what a computer is, looking at it from a philosophical, rather than a historical, point of view. And after *that*, we will return to the question of what computing is, again looking at some philosophical issues.

7.3   What Is 'Computation'?

> Many now view computation as a fundamental part of nature, like atoms
> or the integers. [Fortnow, 2010, 2]

Although I have been talking about "computing", other terms that are used to describe more or less the same territory are 'computation' and 'algorithms'. It may be worth a brief detour into the etymology of these and some related terms.

7.3.1   'compute'

According to the *Oxford English Dictionary* (*OED*)[1], the verb 'compute' comes from the Latin verb '*computare*', meaning "to calculate, reckon, to count up". But when people talk about "computing" today, they mean a great deal more than mere counting; computing has come to include everything we can do with computers, from text processing to watching videos and playing games, so, clearly, the meaning has been extended to include non-numerical "reckoning".

The Latin word '*computare*' itself comes from the Latin morpheme '*com*', meaning "together with", and the Latin word '*putare*', meaning "to cleanse, to prune, to reckon, to consider, think" (and 'putare' came from a word meaning "clean, pure"). So, in ancient Rome at least, to "compute" seems to have meant, more or less, something like: "to consider or think about things together", "to clear things up together", or maybe "to reckon with (something)".

7.3.2   'reckon'

The verb 'reckon' originally meant "to give an account of, recount; to tell; to describe", and later came to mean "to count, to calculate". 'Reckon' is from an Indo-European[2] root '*rek*', possibly meaning "to reach" or "to tell, to narrate, to say" (as in "to recite"

---

[1]http://www.oed.com/view/Entry/37974

[2]http://en.wikipedia.org/wiki/Indo-European_languages

or "to recount"). These meanings, in turn, may derive from an earlier meaning "to arrange", "to put right", "to move in a straight line".[3]

7.3.3 'count', 'calculate', 'figure'

The origins of 'count', 'calculate', and 'figure' are also interesting.

'Count' also came from '*computare*' and originally meant "to enumerate", "to recite a list" (and, as we just saw, 'recite' is probably related to 'reckon'). Note that when you "count", you "recite" a list of number words.

'Calculate' came from Latin '*calculus*'. This certainly did not mean the contents of a certain high school or college course that studies the branch of mathematics concerned with differentiation and integration, and invented by Newton and Leibniz in the 17th century! Rather, it meant "pebble" or "small stone"(!), since counting was done with stones originally! (Even today, a "calculus" in medicine is an accumulation of minerals in the body, forming a small, stone-like object. The root '*calc*' came from 'calx', meaning "chalk, limestone", and is related to 'calcium'.

The *verb* 'to figure' means "to use figures to reckon", where the *noun* 'figure' seems originally to have meant "numerical symbol". The earliest citation for 'figure' in the *OED* is from 1225, when it meant "numerical symbol". A citation from 1250 has the meaning "embodied (human) form". And a citation from 1300 has the more general meaning of "shape". (This conversion of the noun 'figure' to a verb is an example of what the computer scientist Alan Perlis meant when he joked, "In English, every word can be verbed".)

7.3.4 'computation'

The bottom line seems to be this: 'Computation' originally meant something very closely related to our modern notion of "symbol (i.e., shape) manipulation", which is another way of describing syntax. (See Ch. 22, as yet unwritten.)

Now that we know how we got the *word* 'computation', we'll turn to what the *object* computation is.

7.4 What Is Computation?

To understand what computation is, we first need to understand what a (mathematical) function is.

7.4.1 What Is a Function?

7.4.1.1 Two Meanings

The English word 'function' has at least two, very different meanings. The ordinary, everyday meaning is, roughly, "purpose". Instead of asking, "What is the purpose of this button?", we might say, "What is the function of this button? What does it do?".

---

[3]http://etymonline.com/?term=reckon. Cf. http://www.utexas.edu/cola/centers/lrc/ielex/R/P1613.html

But we will be interested in its other, mathematical meaning, as when we say that some "dependent variable" *is a function of*—i.e., depends on—some "independent variable". (As it turns out, this technical sense of the word was, first of all, initiated by our old friend Leibniz and, second, was an extension of its other meaning; for the history of this, see the *OED* entry on the noun 'function', in its mathematical sense (sense 6; http://www.oed.com/view/Entry/75476. On the history of the concept of "function", see [O'Connor and Robertson, 2005].)

7.4.1.2 Functions Described Extensionally

Many introductory textbooks define a (mathematical) function as an "assignment" of values to inputs, but they never define what an "assignment" is.[4] A much more rigorous way of defining a function is to give a definition based on set theory.

A **function** (described "extensionally")[5] is a set of input-output pairs such that no two of them have the same input (or first element). In other words, a function is what is called a 'binary relation': a set of ordered pairs of elements from two sets. (The "two" sets can be the same one; you can have a relation among the members of a single set.) But it is a special kind of binary relation in which no *two*, distinct members of the relation have the same first element (but different second elements). That is, if you have what you think are two different members of a function, and if they have the same first element, then they also have the same second element, and so it only seemed as if they were two members; they were really one and the same member, not two different ones. As a rule of thumb, a binary relation is a function if "same input implies same output".

The logically correct way to say this, in mathematical English, is:

Let A, B be sets.
Then f **is a function from** A to B
=def (i.e., means by definition)

1. f is a binary relation from A to B,

<div align="center">and</div>

2. *for all* members a of A,
    and *for all* members b of B
    and *for all* members b' of B[6]
    if: (a, b) is a member of f
    and *if*: (a, b') is also a member of f
    then: $b = b'$

Because we are considering a binary relation as a set of ordered pairs, let's write the members of a binary relation from A to B as $\langle a, b \rangle$, where $a \in A$ and $b \in B$. Here are some examples of functions in this extensional sense:

---

[4]It is something like a "mapping". Such an "assignment" is not quite the same thing as an assignment of a value to a variable in a programming language or in a system of logic.

[5]http://www.cse.buffalo.edu/∼rapaport/intensional.html

[6]Keep in mind that b is allowed to be the same as b'!

1. $f = \{\langle 0,1 \rangle, \langle 1,2 \rangle, \langle 2,3 \rangle, \dots\}$

   This is sometimes written: $f(0) = 1, f(1) = 2, \dots$

2. $g = \{\langle 0,0 \rangle, \langle 1,2 \rangle, \langle 2,4 \rangle, \langle 3,6 \rangle, \dots\}$

   This is sometimes written: $g(0) = 0, g(1) = 2, \dots$

3. $E(y) = \langle m, d \rangle$

   $= \langle$

   $(((((19 * (y \bmod 19)) + (y/100) - ((y/100)/4) - (((8 * (y/100)) + 13)/25) + 15) \bmod 30) - (((y \bmod 19) + (11 * (((19 * (y \bmod 19)) + (y/100) - ((y/100)/4) - (((8 * (y/100)) + 13)/25) + 15) \bmod 30)))/319) + (((2 * ((y/100) \bmod 4)) + (2 * ((y \bmod 100)/4)) - ((y \bmod 100) \bmod 4) - (((19 * (y \bmod 19)) + (y/100) - ((y/100)/4) - (((8 * (y/100)) + 13)/25) + 15) \bmod 30) - (((y \bmod 19) + (11 * (((19 * (y \bmod 19)) + (y/100) - ((y/100)/4) - (((8 * (y/100)) + 13)/25) + 15) \bmod 30)))/319) + 32) \bmod 7) + 90)/25$,

   $(((((19 * (y \bmod 19)) + (y/100) - (y/100)/4 - (((8 * (y/100)) + 13)/25) + 15) \bmod 30) - (((y \bmod 19) + (11 * (((19 * (y \bmod 19)) + (y/100) - (y/100)/4 - (((8 * (y/100)) + 13)/25) + 15) \bmod 30)))/319) + (((2 * ((y/100) \bmod 4)) + (2 * ((y \bmod 100)/4)) - ((y \bmod 100) \bmod 4) - (((19 * (y \bmod 19)) + (y/100) - (y/100)/4 - (((8 * (y/100)) + 13)/25) + 15) \bmod 30) - (((y \bmod 19) + (11 * (((19 * (y \bmod 19)) + (y/100) - (y/100)/4 - (((8 * (y/100)) + 13)/25) + 15) \bmod 30)))/319) + 32) \bmod 7) + (((((19 * (y \bmod 19)) + (y/100) - (y/100)/4 - (((8 * (y/100)) + 13)/25) + 15) \bmod 30) - (((y \bmod 19) + (11 * (((19 * (y \bmod 19)) + (y/100) - (y/100)/4 - (((8 * (y/100)) + 13)/25) + 15) \bmod 30)))/319) + (((2 * ((y/100) \bmod 4)) + (2 * ((y \bmod 100)/4)) - ((y \bmod 100) \bmod 4) - (((19 * (y \bmod 19)) + (y/100) - (y/100)/4 - (((8 * (y/100)) + 13)/25) + 15) \bmod 30) - (((y \bmod 19) + (11 * (((19 * (y \bmod 19)) + (y/100) - (y/100)/4 - (((8 * (y/100)) + 13)/25) + 15) \bmod 30)))/319) + 32) \bmod 7) + 90)/25 + 19) \bmod 32$

   $\rangle$

   This function takes as input a year y and outputs an ordered pair consisting of the month m and day d that Easter falls on in year y.[Stewart, 2001][7]

4. Here is a *finite* function (i.e., a function with a finite number of members— remember: a function is a *set*, so it has members):

   $h = \{\langle$ 'yes', print 'hello' $\rangle$, $\langle$ 'no', print 'bye' $\rangle$, $\langle$ input $\neq$ 'yes' & input $\neq$ 'no', print 'sorry' $\rangle\}$

   The idea behind this function is that it prints 'hello', if the input is 'yes';
   it prints 'bye', if the input is 'no';
   and it prints 'sorry', if the input is neither 'yes' nor 'no'.

   ---

   [7]See also http://techsupt.winbatch.com/TS/T000001048F35.html (accessed 11 December 2013).

5. A *partial* function (i.e., a function that has no outputs for some possible inputs):

$$k = \{\ldots, \langle -2, \frac{1}{-2} \rangle, \langle -1, \frac{1}{-1} \rangle, \langle 1, \frac{1}{1} \rangle, \langle 2, \frac{1}{2} \rangle, \ldots\}$$

Here, $k(0)$ is undefined.

7.4.1.3 Functions Described as Machines

Sometimes, functions are characterized as "machines" that take input into a "black box" with a "crank" that mysteriously converts the input into output:
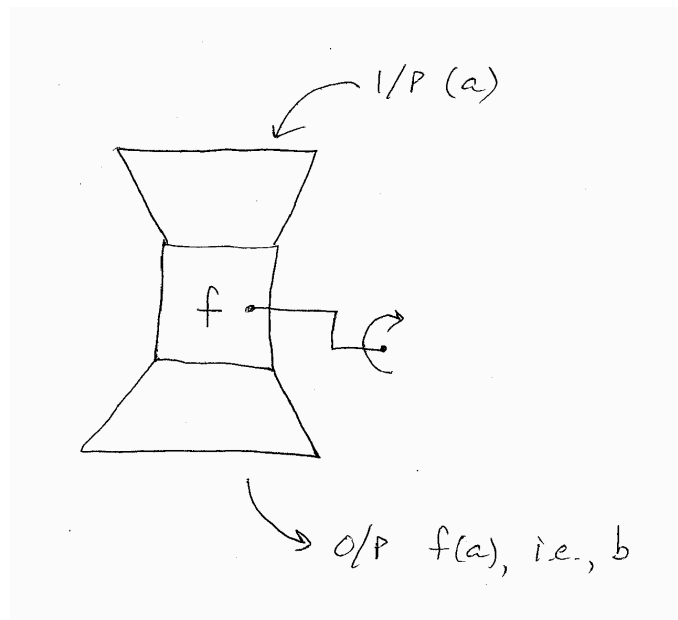
In this picture, f is a machine into which you put a; you then turn a crank (clockwise, let's suppose); f then grinds away at the input by means of some mysterious mechanism; and finally the machine outputs b (i.e., $f(a)$). But this view of a function as being something "active" or "dynamic" that changes something is incorrect.[8]

---

[8] See http://www.askphilosophers.org/question/1877.

Despite what you may have been told elsewhere (I was told this in high school), this is **NOT** what a function is! A function, as we have seen, is merely the *set* of input-output pairs. So, what is the machine? *It is a **computer**!* (And the mysterious "gears" hidden inside the black box implement an *algorithm* that computes the function.) In fact, one problem with this machine metaphor for a function, as we will see, is that not all functions can be computed by algorithms; that is, there are functions for which there are no such "function machines".[9]

7.4.1.4 Functions Described Intensionally

Sometimes, functions are described ("intensionally")[10]—but not *uniquely* described—by **formulas**. Here are some examples:

1. $f(i) = 2i$

2. $g(i) = i + 1$

3. $g'(i) = 2i - i + 7/(3 + 4)$

   Note that both g and g' describe the same function; that is, $g = g'$, even though their formulas are different.

4. $h(i) = \begin{cases} \text{'hello'} & \text{if } i = \text{'yes'} \\ \text{'bye'} & \text{if } i = \text{'no'} \\ \text{'sorry'}, & \text{otherwise} \end{cases}$

5. if $i \neq 0$, then $k(i) = \frac{1}{i}$.

Although formulas may look a lot like algorithms, they are not the same thing. Consider, for example, the formula $2 + 4 * 5$: Without an explicit statement of a rule telling you whether to multiply first or to add first, there is no way of knowing whether the number expressed by that formula is 30 or 22. Such a rule, however, would constitute an algorithm telling you how to calculate the value of the formula. Or consider the formula $2x + 1$: Should you first calculate $2x$ and then add 1? Or should you store 1 somewhere (say, by writing it on a piece of paper), then calculate $2x$, and finally add $2x$ to 1? Of course, the commutative law of addition tells us that, in this case, it doesn't matter in which order you compute the value of the formula, but here we have a clear case of only one formula but (at least) two distinct algorithms.

Perhaps an even clearer example is function E, above—the one that tells you when Easter occurs. I dare you to try to use this formula to find out when Easter will occur

---

[9]Interestingly, Gödel made this observation in the 1930s in an unpublished paper!

> [Turing] has shown that the computable functions defined in this way [that is, in terms of Turing machines] are exactly those for which you can construct a machine with a finite number of parts which will do the following thing. If you write down any number $n_1, \ldots, n_r$ on a slip of paper and put the slip into the machine and turn the crank, then after a finite number of turns the machine will stop and the value of the function for the argument $n_1, \ldots, n_r$ will be printed on the paper. [Gödel, 930s, 168], cited in [Shagrir, 2006a, §3] and in [Soare, 2009, §3.2, p. 374].

So, the machine pictured in Fig. 7.4.1.3 is a Turing machine!

[10]See http://www.cse.buffalo.edu/∼rapaport/intensional.html

next year! Where would you even begin? (A related matter is knowing whether the formula is even correct! We'll explore this issue in Ch. 16.)[11]

(For a good discussion of the difference between formulae and algorithms, see the question asked in [Wangsness and Franklin, 1966] (quoted in full at the beginning of this chapter) and the answers in [Huber, 1966] and [Knuth, 1966]; Knuth's answer is a commentary on Huber's. Huber's answer, roughly, is that an algorithm is a set of instructions for computing the value of a function by "executing" (or carrying out, or following) the instructions, whereas a formula is an expression describing the value of a function; it can be "evaluated" (that is, the value of the function can be determined from the formula) but not executed (because a formula does not come equipped with an algorithm for telling you *how* to evaluate it). In a sense, a formula is "static", whereas an algorithm is (potentially) "dynamic".)

Functions describable by formulas are not the only kind of functions. There are functions without formulas for computing them; that is, there are functions such that the "gears" of their "machines" work by magic! For example, there are "table look-up" functions, where the only way to identify the correct output for a given input is to look it up in a table (rather than to compute it); usually, this is the case when there is no lawlike pattern relating the inputs and the outputs. Of course, there are non-computable functions, such as the Halting Problem (we'll have more to say on what this in §7.8). And there are random functions.

In fact, one of the central purposes—perhaps *the* central quesion—of computer science (or, at least, of the non-engineering portion of the field) is to figure out which functions *do* have algorithms for computing them! This includes "non-mathematical" functions, such as the (human) cognitive "functions" that take as input sensory information from the environment and produce as output (human, cognitive) behavior. To express this another way, the subfield of computer science known as AI can be considered as having as its purpose figuring out which such cognitive functions are computable.

7.4.1.5 Computable Functions

So we have two concepts: function and algorithm. We have given a careful definition of the mathematical notion of function. We have not yet give a careful definition of the mathematical notion of algorithm, but we have given some informal characterizations (and we will look at others in §7.5, below). We can combine them as follows:

> A function f **is computable** will mean, roughly, that there is an "algorithm" that *computes f*.

This is only a rough definition or characterization because, for one thing, we haven't yet defined 'algorithm'. But, assuming that an algorithm is, roughly, a set of instructions for computing the output of the function, then it makes sense to define a function as being computable if we can...well...compute it!

---

[11]I created this formula by working backwards from the algorithm given in [Stewart, 2001], so it's quite possible that I introduced a typographical error! Even if I didn't, I am assuming that [Stewart, 2001]'s algorithm is correct. And that could be a big assumption.

So, a function f is computable if there is an algorithm A_f such that, for all inputs i, $A_f(i) = f(i)$. That is, a function f is computable by an algorithm A_f if both f and A_f have the same input-output "behavior" (that is, if both define the same binary relation, or set of input-output pairs). Moreover, A_f must specify *how* f's inputs and outputs are related. So, whereas a function only *shows* its input-output pairs but is silent about *how* they are related, an algorithm for that function must say more. It must be a procedure, or a mechanism, or a set of intermediate steps or instructions that either transforms the input into the output or else shows you explicitly how to find the output by starting with the input, or how to get from the input to the output. Algorithms shouldn't be magic or merely arbitrary.

It seems easy enough to give examples of algorithms for some of the functions listed earlier:

1. $A_f(i) =$

    input i;
    add 1 to i;
    output result.

2. $A_g(i) =$

    input i;
    multiply i by 2;
    output result.

3. $A'_f(i) =$

    input i;
    mulitply i by 2;
    call the result x;
    subtract i from x;
    add 3 to 4;
    call the result y;
    divide 7 by y;
    add x to y;
    output result.

4. For $E(m,d)$, see the English algorithm in [Stewart, 2001] or the computer program online at
   http://techsupt.winbatch.com/TS/T000001048F35.html.
   Note that, even though that algorithm may not be easy to follow, it is certainly much easier than trying to compute the output of E from the formula. (For one thing, the algorithm tells you where to begin!)

Can this notion of algorithm be made more precise? *How?*

7.5   Algorithms Made Precise

> The meaning of the word *algorithm*, like the meaning of most other words
> commonly used in the English language, is somewhat vague. In order to
> have a *theory of algorithms*, we need a mathematically precise definition
> of an algorithm. However, in giving such a precise definition, we run
> the risk of not reflecting exactly the intuitive notion behind the word.
> [Herman, 1983, 57]

Before anyone attempted to define 'algorithm', many algorithms were in use
both by mathematicians (e.g., ancient Babylonian procedures for finding lengths
and for computing compound interest—see [Knuth, 972a], Euclid's procedures for
construction of geometric objects by compass and straightedge [Toussaint, 1993],
Euclid's algorithm for computing the greatest common divisor of two integers) as
well as by ordinary people (e.g., the algorithms for simple arithmetic with Hindu-
Arabic numerals [Robertson, 1979]).[12] In fact, the original, eponymous use of the
word referred to those arithmetic rules as devised by Abu Abdallah Muhammad ibn
Musa Al-Khwarizmi, a Persian mathematician who lived ca. 780–850 AD, i.e., about
1200 years ago.[13]

When David Hilbert investigated the foundations of mathematics, his followers
began to try to make the notion of algorithm precise, beginning with discussions of
"effectively calculable", a phrase first used by Jacques Herbrand in 1931 [Gandy, 1988,
68] and later taken up by Alonzo Church [Church, 1936b] and Stephen Kleene
[Kleene, 1952], but left largely undefined, at least in print.[14]

J. Barkley Rosser [Rosser, 1939, 55 (my italics and enumeration)] made an effort
to clarify the contribution of the modifier 'effective':

> "Effective method" is here used in the rather special sense of a method
> each step of which is [1] *precisely predetermined* and which is [2] *certain
> to produce the answer* [3] in a *finite number of steps*.

But what, exactly, does 'precisely predetermined' mean? And does 'finite number
of steps' mean that the written statement of the algorithm has a finite number of
instructions, or that, when executing them, only a finite number of tasks must be
performed? (In other words, what gets counted: written steps or executed instructions?
One written step—"for $i := 1$ to 100 do $x := x + 1$"—can result in 100 executed

---

[12]This section is adapted from [Rapaport, 2012b, Appendix].

[13]What sounds like his last name, 'Al-Khwarizmi', really just means something like "the
person who comes from Khwarizm; "the Aral Sea...was once known as Lake Khwarizm"
[Knuth, 1985, 171]. See http://www-groups.dcs.st-and.ac.uk/~history/Mathematicians/Al-Khwarizmi.html
and [Crossley and Henry, 1990].

[14][Church, 1956] calls it an "informal notion": See pp. 50 (and our §10.5.1 below), 52 (including notes
118 ["an effective method of calculating, especially if it consists of a sequence of steps with later steps
depending on results of earlier ones, is called an *algorithm*"] and 119 ["an effective method of computation,
or algorithm, is one for which it woud be possible to build a computing machine", by which he means a
Turing machine]), 83, 99n.183 ["a procedure...should not be called effective unless there is a predictable
upper bound of the number of steps that will be required"], and 326 (note 535); cf. [Manzano, 1997, 219–
220]. See also Sieg 1997; another explication of 'effective' is in [Gandy, 1980, 124], which we'll return to
in Ch. 10.

instructions. And one written step—"while true do $x := x + 1$"—can result in an infinite number of executed instructions!)

Much later, *after* Turing's, Church's, Gödel's, and Post's CITE? precise formulations and *during* the age of computers and computer programming, A.A. Markov, Stephen Kleene, and Donald Knuth also gave slightly less vague, though still informal, characterizations.

According to Markov [Markov, 1954, 1], an algorithm is a "computational process" satisfying three (informal) properties:

1. being "determined" ("carried out according to a precise prescription...leaving no possibility of arbitrary choice, and in the known sense generally understood"),

2. having "applicability" ("The possibility of starting from original given objects which can vary within known limits"), and

3. having "effectiveness" ("The tendency of the algorithm to obtain a certain result, finally obtained for appropriate original given objects").

These are a bit obscure: Being "determined" may be akin to Rosser's "precisely predetermined". But what about being "applicable"? Perhaps this simply means that an algorithm must not be limited to converting one, specific input to an output, but must be more general. And Markov's notion of "effectiveness" seems restricted to only the second part of Rosser's notion, namely, that of "producing the answer". There is no mention of finiteness, unless that is implied by being computational.

In his undergraduate-level, logic textbook, Kleene [Kleene, 1967] elaborates on the notions of "effective" and "algorithm" that he left unspecified in his earlier, classic, graduate-level treatise on metamathematics. He continues to identify "effective procedure" with "algorithm" [Kleene, 1967, 231], but now he offers a characterization of an algorithm as

1. a "procedure" (i.e., a 'finite' "set of rules or instructions") that

2. "in a finite number of steps" answers a question, where

3. each instruction can be "followed" "mechanically, like robots; no insight or ingenuity or invention is required",

4. each instruction "tell[s] us what to do next", and

5. the algorithm "enable[s] us to recognize when the steps come to an end" [Kleene, 1967, 223].

Knuth [Knuth, 1973, "Basic Concepts: §1.1: Algorithms", pp. xiv–9, esp. pp. 1–9] goes into considerably more detail, albeit still informally. He says that an algorithm is "a finite set of rules which gives a sequence of operations for solving a specific type of problem", with "five important features" [Knuth, 1973, 4]:

1. "*Finiteness*. An algorithm must always terminate after a finite number of steps" [Knuth, 1973, 4].

Note the double finiteness: A finite number of rules in the text of the algorithm *and* a finite number of steps to be carried out. Moreover, algorithms must halt. (Halting is not guaranteed by finiteness; see point 5, below.) Interestingly, Knuth also says that an algorithm is a finite "computational method", where a "computational method" is a "procedure" that only has the next four features [Knuth, 1973, 4].

2. "*Definiteness*. Each step. . . must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified. . ." [Knuth, 1973, 5].

   This seems to be Knuth's analogue of the "precision" that Rosser and Markov mention.

3. "*Input.* An algorithm has zero or more inputs" [Knuth, 1973, 5].

   Curiously, only Knuth and Markov seem to mention this explicitly, with Markov's "applicability" property suggesting that there must be at least *one* input. Why does Knuth say *zero* or more? Presumably, he wants to allow for the possibility of a program that simply outputs some information. On the other hand, if algorithms are procedures for computing functions, and if functions are "regular" sets of input-output pairs (regular in the sense that the same input is always associated with the same output), then algorithms would always have to have input. Perhaps Knuth has in mind the possibility of the input being internally stored in the computer rather than having to be obtained from the external environment. Or perhaps this is how constant functions (functions whose output is constant, no matter what their input is) are handled. It is worth noting, however, that [Hartmanis and Stearns, 1965, 288]—the founding document of the field of computational complexity—allows their multi-tape Turing machines to have at most one tape, which is an output-only tape; there need not be any input tapes.

4. "*Output.* An algorithm has one or more outputs" [Knuth, 1973, 5].

   That there must be at least one output echoes Rosser's property (2) ("certain to produce the answer") and Markov's notion (3) of "effectiveness" ("a certain result"). But Knuth characterizes outputs as "quantities which have a specified relation to the inputs" [Knuth, 1973, 5]; the "relation" would no doubt be the functional relation between inputs and outputs, but, if there is no input, what kind of a relation would the output be in? Very curious![15]

5. "*Effectiveness*. This means that all of the operations to be performed in the algorithm must be sufficiently basic that they can in principle be done exactly and in a finite length of time by a man [sic] using pencil and paper" [Knuth, 1973, 6].

---

[15]For an example of an algorithm that has an input but no output, see §17; see also [Copeland and Shagrir, 2011, 230–231].

Note, first, how the term 'effective' has many different meanings among all these characterizations of "algorithm", ranging from it being an unexplained term, through being synonymous with 'algorithm', to naming very particular—and very different— properties of algorithms.

Second, it is not clear how Knuth's notion of effectiveness differs from his notion of definiteness; both seem to have to do with the preciseness of the operations.

Third, Knuth brings in another notion of finiteness: finiteness in time. Note that an instruction to carry out an infinite sequence of steps in a finite time could be accomplished by doing each step twice as fast as the previous step; or each step might only take a finite amount of time, but the number of steps required might take longer than the expected life of the universe (as in computing a perfect, non-losing strategy in chess; cf. [Zobrist, 2000, 367]). These may have interesting theoretical implications (which we will explore in Ch. 11) but do not seem very practical. Knuth [Knuth, 1973, 7] observes that "we want *good* algorithms in some loosely-defined aesthetic sense. One criterion of goodness is the length of time taken to perform the algorithm...."

Finally, the "gold standard" of "a [hu]man using pencil and paper" seems clearly to be an allusion to Turing's analysis [Turing, 1936], which we will examine in great detail in the next chapter.

We can summarize these informal observations as follows:

An algorithm (for executor E to accomplish goal G) is:

1. a procedure, i.e., a finite set (or sequence) of statements (or rules, or instructions), such that each statement is:
    (a) composed of a finite number of symbols (or marks) from a finite alphabet
    (b) and unambiguous for E—i.e.,
        i. E knows how to do it
        ii. E can do it
        iii. it can be done in a finite amount of time
        iv. and, after doing it, E knows what to do next—
2. which procedure takes a finite amount of time, i.e., halts,
3. and that ends with G accomplished

But the important thing to note is that the more one tries to make precise these *informal* requirements for something to be an algorithm, the more one recapitulates Turing's motivation for the formulation of a Turing machine. We will look at exactly what Turing did in Ch. 8. But first we are going to look a bit more deeply into the current view of computation.

7.6   Four Great Insights of Computer Science

In this section, we will look at four great insights of computer science: The first three help make precise the vague notion of algorithm that we have been looking at. The fourth links the vague notion with a precise one. Together, they define the smallest possible language in which you can write any procedure for any computer. (And by 'computer' here, I mean anything—machine or human—that can execute an algorithm.)

7.6.1   Bacon's, Leibniz's, Morse's, Boole's, Ramsey's, Turing's, and Shannon's Insight

The first great insight is this:

> **All the information about any computable problem can be represented using only *two nouns*: 0 and 1**

Rather than enter into a complicated and controversial historical investigation of who is responsible for this insight, I have chosen instead to simply list the names of some of the people who contributed to it:

- Sir Francis Bacon, around 1605, developed an encoding of the alphabet by any objects "capable of a twofold difference".[16] And, of course, once you've represented the alphabet in a binary coding, then anything capable of being represented in text can be similarly encoded.

- Our old friend, Leibniz gave an "Explanation of Binary Arithemetic" in 1703.[17]

- Famously, Samuel F.B. Morse not only invented the telegraph but also (following in Bacon's footsteps) developed his eponymous, binary code in the mid-1800s.[18]

- The philosopher Frank P. Ramsey, in a 1929 essay on "a language for discussing...facts"—perhaps something like Leibniz's *characteristica universalis* (which we discussed in §6.5 [and possibly also in Ch. 3])— suggested that "all [of the terms of the language] may be best symbolized by numbers. For instance, colours have a structure, i which any given colour maybe assigned a place by three numbers.... Even smells may be so treated..." [Ramsey, 1929, 101–102]. PERHAPS ADD MORE EXAMPLES? SEE http://www.cse.buffalo.edu/~rapaport/111F04/greatidea1.html

- In 1936, as we will see in Ch. 8, Turing made essential use of 0 and 1 in the development of Turing machines.

---

[16][Cerf, 2015, 7] and http://home.hiwaay.net/~paul/bacon/advancement/book6ch1.html

[17]http://www.leibniz-translations.com/binary.htm

[18]http://en.wikipedia.org/wiki/Morse_code. Arguably, however, Morse code (traditionally conceived as having only two symbols, "dot" and "dash") is not strictly binary, because there are "blanks", or time-lapses, between those symbols; CITE GLEICK.

- Finally, the next year, Claude Shannon (in his development of the mathematical theory of information) used "The symbol 0. . . to represent. . . a closed circuit, and the symbol 1. . . to represent. . . an open circuit" [Shannon, 1937, 4] and then showed how propositional logic could be used to represent such circuits.

There is nothing special about 0 or 1. As Bacon emphasized, any other "bistable"[19] pair suffices, as long as they can flip-flop between two easily distinguishable states, such as "on/off", "magnetized/de-magnetized", "high voltage/low voltage", etc. Strictly speaking, these can be used to represent *discrete* things; *continuous* things can be approximated to any desired degree, however. This limitation to two nouns is not necessary: Turing's original theory had no restriction on how many symbols there were. There were only restrictions on the nature of the symbols (they couldn't be too "close" to each other; that is, they had to be distinguishable) and that there be only finitely many.

But, if we want to have a minimal language for computation, having only two symbols suffices, and making them 0 and 1 (rather than, say, 'A' and 'B') is mathematically convenient.

7.6.2 Turing's Insight

So we have two nouns for our minimal language. Turing is also responsible for providing the *verbs* of our minimal language. Our second great insight is this:

Every algorithm can be expressed in a language for a computer (namely, a Turing machine) **consisting of:**

- **an arbitrarily long, paper tape divided into squares** (like a roll of toilet paper, except you never run out [Weizenbaum, 1976]),

- **with a read/write head,**

- **whose only nouns are '0' and '1',**

- **and whose only five verbs (or basic instructions) are:**

  1. move-left-1-square
  2. move-right-1-square
     - (These could be combined into a single verb (that is, a function) that takes a direct object (that is, a single input argument): move(location).)
  3. **print-0-at-current-square**
  4. **print-1-at-current-square**
  5. **erase-current-square**
     - (These could be combined into another, single, transitive verb: print(symbol), where symbol could be either '0', '1',

---

[19]I.e., something that can be in precisely one of two states; http://en.wikipedia.org/wiki/Bistability.

or 'blank' (here, erasing would be modeled by printing a blank).)[20]

So, we can get by with either only two (slightly complex) verbs or five (slightly simpler) verbs. But either version is pretty small.

7.6.3   Boehm and Jacopini's Insight

We have two nouns and perhaps only two verbs. Now we need some grammatical rules to enable us to put them together. The software-engineering concept of "structured programming" does the trick. This is a style of programming that avoids the use of the 'go to' command. In early programming languages, programmers found it useful to "jump" or "go to" another location in the program, sometimes with the ability to return to where the program jumped *from*, but sometimes not. This resulted in what was sometimes called "spaghetti code", because, if you looked at a flowchart of the program, it consisted of long, intertwined strands of code that were hard to read and harder to ensure that they were correct. Edsger W. Dijkstra wrote a letter to the editor of the *Communications of the ACM*, that was headlined "Go To Statement Considered Harmful" [Dijkstra, 1968], arguing against the use of such statements. This resulted in an attempt to better "structure" computer programs so that the use of 'go to' could be minimized; Corrado Böhm and Giuseppe Jacopini proved that it could be completely eliminated [Böhm and Jacopini, 1966]. This gives rise to the third insight (and the third item needed to form our language):

> **Only three rules of grammar are needed to combine any set of basic instructions (verbs) into more complex ones:**
>
> 1. **sequence**:
>   - first do this; then do that
>
> 2. **selection** (or choice):
>   - IF such-&-such is the case,
>   THEN do this
>   ELSE do that
>
> 3. **repetition** (or looping):
>   - WHILE such & such is the case
>   DO this
>
> . . . where "this" and "that" can be:

---

[20]Deciding how to count the number of actions is an interesting question. [Gurevich, 1999, 99–100] points out that "at one step, a Turing machine may change its control state, print a symbol at the current tape cell[,] and move its head. . . . One may claim that every Turing machine performs only one action a time, but that action [can] have several parts. The number of parts is in the eye of the beholder. You counted three parts. I can count four parts by requiring that the old symbol is erased before the new symbol is printed. Also, the new symbol may be composed, e.g. '12'. Printing a composed symbol can be a composed action all by itself."

- any of the basic instructions, or
- any complex instruction created by application of any grammatical rule.

Digression:

For readers who are familiar with the structure of recursive definitions, what we have here is a recursive definition of the notion of an "instruction": The base cases of the recursion are the primitive verbs of the Turing machine ('move(location)' and 'print(symbol)'), and the recursive cases are given by sequence, selection, and repetition.

More generally, a recursive definition of some concept *C* consists of two parts. The first part, called the "base case", gives you some explicit examples of *C*. These are not just any old examples, but are considered to be the simplest, or most basic or "atomic", instances of *C*— the building blocks from which all other, more complex instances of *C* can be constructed.

The second part of a recursive definition of *C* consists of rules (algorithms, if you will!) that tell you how to construct those more complex instances of *C*. But these rules don't simply tell you how to construct the complex instances from the base cases. Rather, they tell you how to construct the complex instances *from any instances that have already been constructed*. The first complex instances, of course, will be constructed directly from the base cases. But others, even more complex, will be constructed from the ones that were constructed directly from the base cases, and so on. What makes such a definition "recursive" is that simpler instances of *C* "recur" in the definitions of more complex instances.

So, the base case of a recursive definition tells you how to begin. And the recursive case tells you how to continue. But recursive definitions sometimes seem to be circular: After all, we seem to be defining *C* in terms of *C* itself! But really we are defining "new" (more complex) instances of *C* in terms of *other*, "older" (that is, already constructed), or simpler instances of *C*, which is not circular at all. (It would only be circular if the *base cases* were somehow defined in terms of themselves. But they are not "defined"; they are given, by fiat.)[21] **End digression**

There are optional, additional instructions and grammatical rules:

1. An "exit" instruction, allowing the program to exit from a loop under certain conditions, before the body of the loop is completed; this is provided for simplicity or elegance.

---

[21]Not only is recursion considered to be the core of the concept of computation, but it has been argued that it is also the core of the concept of language [Hauser et al., 2002]: The "faculty of language…in the narrow sense…only includes recursion and is the only uniquely human component of the faculty of language". However, this is a highly controversial claim; to follow the debate, see [Pinker and Jackendoff, 2005, Fitch et al., 2005, Jackendoff and Pinker, 2005]. Recursion is discussed more fully in those three follow-up papers than in the original one.

2. named procedures (or procedural abstraction): Define new (typically, complex) actions by giving a single name to a (complex) action. (This is even more optional than "exit", but it is very powerful in terms of human readability and comprehension, and even in terms of machine efficiency; see [Pylyshyn, 1992].)

3. recursion (an elegant replacement for repetition): A recursive instruction tells you how to compute the output value of a function in terms of previously computed output values instead of in terms of its input value; of course, the base case (that is, the output value of the function for its initial input) has to be given to you in a kind of table-lookup.

We now have our minimal language: Any algorithm for any computable problem can be expressed in this language (for a Turing machine) that consists of the two nouns '0' and '1', the two verbs 'move(location)' and 'print(symbol)', and the three grammatical rules of sequence, selection, and repetition.

Is that really all that is needed? Can your interaction with, say, a spreadsheet program or Facebook be expressed in this simple (if not "simple-minded"!) language? There's no doubt that a spreadsheet program, for example, that was written in this language would be very long and very hard to read. But that's not the point. The question is: Could it be done? And the answer is our final great insight.

7.6.4   The (Church-)Turing Thesis

Our final great insight is the answer to the question at the end of the previous section: in one word, 'yes':

> **Effective computability can be identified with (anything logically equivalent to) Turing-machine computability.**

That is, an algorithm is definable as a program expressible in (anything equivalent to) our minimal language.

This idea was almost simultaneously put forth both by Church [Church, 1936b] and by Turing [Turing, 1936]. Some people call it 'Church's Thesis'; others call it 'Turing's Thesis'; and, as you might expect, some call it 'the Church-Turing Thesis'. For this reason, Robert Soare [Soare, 2009] has advocated calling it, more simply and more perspicuously, the 'Computability Thesis'.

But it is only a proposed definition of 'computable' or 'algorithm': It proposes to identify any informal, intuitive notion of effective computability or algorithm with the formal, mathematically precise notion of a Turing machine. How do we know that Turing-machine computability captures (all of) the intuitive notion(s) of effective computability?

There are several lines of evidence:

1. There is Turing's analysis of computation, which we will examine in detail in Ch. 8. But, of course, there is also Church's analysis in terms of the "lambda calculus". Should one of these be preferred over the other? There are two reasons for preferring Turing's over Church's: First, Turing's is easier to understand, because it follows from his analysis of how humans compute. Second—and

this is "merely" an appeal to authority—Gödel preferred Turing's analysis, not only to Church's but also to his own! (For discussion, see [Soare, 2009].) PROBABLY NEED TO SAY A BIT MORE HERE, ESPECIALLY ABOUT GOEDEL AND WHY HE IS AN AUTHORITY

2. All of the formalisms that have been proposed as precise, mathematical analyses of computability are not only logically equivalent (that is, any function that is computable according to one analysis is also computable according to each of the others), but they are constructively equivalent (that is, they are inter-compilable, in the sense that you can write a computer program that will translate (or compile) a program in any of these languages into an equivalent program in any of the others): GIVE CITATIONS

   - Turing machines

   - Post machines (like TMs, but treating the tape as a queue)

   - Church's lambda calculus (which John McCarthy later used as the basis of the Lisp programming language)

   - Markov algorithms (later used as the basis of the Snobol programming language)

   - Post productions (later used as the basis of production systems) [Post, 1941, Post, 1943], [Soare, 2012, 3293].

   - Herbrand-Gödel recursion equations (later used as the basis of the Algol family of programming languages)

   - μ-recursive functions

   - register machines [Shepherdson and Sturgis, 1963], etc.

3. Finally, there is empirical evidence: All algorithms so far translate to Turing-machines; that is, there are no intuitively effective-computable algorithms that are not Turing-machine computable.

But this has not stopped some philosophers and computer scientists from challenging the Computability Thesis. Some have advocated forms of computation that "exceed" Turing-machine computability. We will explore some of these options in Ch. 11.

What we will now turn our attention to is a somewhat more mathematical outline of structured programming and recursive functions, after which we will ask whether there are functions that are non-computable.

7.7   Structured Programming
and Recursive Functions[22]

7.7.1   Structured Programming

In §7.6.3, I mentioned "structured programming", a style of programming that avoids the use of "jump" commands, or the 'go to' command, and that Boehm & Jacopini proved that it could be completely eliminated. Let's see how this can be done:

We can begin with a (recursive) definition of 'structured program': As with all recursive definitions, we need to give a base case and a recursive case. The base case is the notion of a 'basic program'. Using the capital and lower-case Greek letters 'pi' (Π, π) to represent programs, we can define the 'basic programs' as follows:

1. Basic programs:

    (a) If $\pi = $ **begin end.**, then π is a basic (structured) program.

    This is called 'the empty program'.

    (b) Let F be a "primitive operation" that is (informally) computable—such as an assignment statement (which will have the form "$y \leftarrow c$", where y is a variable and c is a value that is assigned to y).

    Then, if $\pi = $ **begin** F **end.**, then π is a basic (structured) program.

    Such programs are called '1-operation programs'. Note that it is not specified which operations are primitive; they will vary with the programming language. That means that structured programming is a *style* of programming, not a particular programming language. It is a style that can be used with *any* programming language. As we will see when we examine Turing's paper in Ch. 8, he spends a lot of time justifying his choice of primitive operations.

    The recursive case for structured programs specifies how to construct more complex programs from simpler ones. The simplest ones, of course, are the basic programs: the empty program and the 1-operation programs. So, in good recursive fashion, we begin by constructing slightly more complex programs from these. Once we have both the basic programs and the slightly more complex programs constructed from them, we can combine them—using the recursive constructs below—to form more complex ones.

2. Program constructors:

    Let π, π' be (simple or complex) programs, each of which contains exactly 1 occurrence of **end**.

---

[22]The material in this section is based on lectures by John Case. WHICH IN TURN ARE BASED ON CLARK & COWELL; SO COMPARE MY PRESENTATION WITH THEIRS!!

Let P be a "Boolean test". (A Boolean test, such as "$x > 0$", is sometimes called a 'propositional function' or 'open sentence'. The essential feature of a Boolean test is that either its output value is true or else it is false.) P must also be (informally) computable (and, again, Turing spends a lot of time justifying his choices of tests).

And let y be an integer-valued variable.

Then the following are also (more complex) programs:

(a) $\Pi =$ **begin** π; π' **end.** is a (complex) structured program.

      Such a Π is the "linear concatenation" of π followed by π'.

(b) $\Pi =$ **begin**
         if P
           then π
           else π'
    end.
is a (complex) structured program.

      Such a Π is a "conditional branch": If P is true, then π is executed, but, if P is false, then π' is executed.

(c) $\Pi =$ **begin**
         while $y > 0$ **do**
           begin
             π;
             $y \leftarrow y - 1$
           end
    end.

      Such a Π is called a "count loop" (or "for-loop", or "bounded loop"): The simpler program π is repeatedly executed while (i.e., as long as) the Boolean test "$y > 0$" is true (i.e., until it becomes false). Eventually, it will become false, because each time the loop is executed, y is decremented by 1, so eventually it must become equal to 0. Thus, an infinite loop is avoided.

(d) $\Pi =$ **begin**
         while P **do** π

    end.
is a (complex) structured program.

      Such a Π is called a "while-loop" (or "free" loop, or "unbounded" loop): The simpler program π is repeatedly executed while (i.e., as long as) the Boolean test P is true (i.e., until P is false). Note that, unlike the case of a count loop, a while loop can be an infinite loop, because there is no built-in guarantee that P will eventually become false (because, in turn, there is no restriction on what P

can be, as long as it is a Boolean test; in particular, if the Boolean
test is the constantly-true test "true"—or the constantly-true test
"1=1"—then the loop will be guaranteed to be infinite).

We can classify structured programs based on the above recursive definition:

1. π *is a count-program* (or a "for-program", or a "bounded-loop program") =def

   (a) π is a basic program, or

   (b) π is constructed from count-programs by:

   - linear concatenation, or
   - conditional branching, or
   - count looping

   (c) Nothing else is a count-program.

2. π *is a while-program* (or a "free-loop program", or an "unbounded-loop program") =def

   (a) π is a basic program, or

   (b) π is constructed from while-programs by:

   - linear concatenation, or
   - conditional branching, or
   - count-looping, or
   - while-looping

   (c) Nothing else is a while-program.

The inclusion of count-loop programs in the construction-clause for while-programs is not strictly needed, because all count-loops are while-loops (just let the P of a while-loop be $y > 0$ and let the π of the while-loop be the linear concatenation of some other π' followed by $y \leftarrow y - 1$).

So count-programs are a proper subclass of while-programs: While-programs include all count-programs plus programs constructed from while-loops that are not also count-loops.

7.7.2   Recursive Functions

Now let's look at one of the classic analyses of computable functions: a recursive definition of non-negative integer functions that are intuitively computable—that is, functions whose inputs are non-negative integers, also known as "natural numbers".

First, what is a "natural number"? The set $\mathbb{N}$ of natural numbers = $\{0, 1, 2, \ldots\}$. They are the numbers defined (recursively!) by Peano's axioms [Peano, 1889], [Dedekind, 1890]:

P1 0 is a natural number. (This is the base case.)

P2 Let S be a one-to-one (or "injective") function that satisfies the following two properties:

(a) S takes as input a natural number and returns as output its "successor"

(b) For any natural number n, $S(n) \neq 0$
(so, S is not an "onto", or "surjective", function).

Then, if n is a natural number, then so is $S(n)$.
(This is the recursive case.)

P3 Nothing else is a natural number.

That is, consider a subset **M** of $\mathbb{N}$. Suppose that:

(a) $0 \in \mathbf{M}$

and that

(b) for all $n \in \mathbb{N}$, if $n \in \mathbf{M}$, then $S(n) \in \mathbf{M}$.

Then $S = \mathbb{N}$.

(This is the axiom that underlies the logical rule of inference known as "mathematical induction".)

To define the recursive functions, we once again begin with "basic" functions that are intuitively, clearly computable, and then we construct more complex functions from them.

(a) Basic functions:

i. *successor:* $S(x) = x + 1$

ii. *predecessor:* $P(x) = x \mathbin{\dot{-}} 1$ (sometimes pronounced x "monus" 1)
(where $a \mathbin{\dot{-}} b = \text{def} \begin{cases} a - b, & \text{if } a \geq b \\ 0, & \text{otherwise} \end{cases}$)

iii. *projection:*[24] $P_k^j(x_1, \ldots, x_j, \ldots, x_k) = x_j$

These can be seen to correspond in an intuitive way to the basic operations of a Turing machine: The successor function corresponds to move(right), the predecessor function corresponds to move(left) (where you cannot move any further to the left than the beginning of the Turing-machine tape), and the projection function corresponds to reading the current square of the tape.[25]

(b) Function constructors:
Let $g, h, h_1, \ldots, h_m$ be (basic or complex) recursive functions.
Then the following are also (complex) recursive functions:

---

[24]Sometimes called 'identity' [Kleene, 1952, 220], [Soare, 2012, 3280].

[25]For an analogous comparison, in the context of register machines, see [Shepherdson and Sturgis, 1963, 220].

i. *f is defined from $g, h_1, \ldots, h_m$ by generalized composition* =def
$$f(x_1, \ldots, x_k) = g(h_1(x_1, \ldots, x_k), \ldots, h_m(x_1, \ldots, x_k))$$

This can be made a bit easier to read by using the symbol $\bar{x}$ for the sequence $x_1, \ldots, x_k$. If we do this, then generalized composition can be written as follows:

$$f(\bar{x}) = g(h_1(\bar{x}), \ldots, h_m(\bar{x}))$$

and this itself can be further simplified to:

$$f(\bar{x}) = g(\overline{h(\bar{x})})$$

Note that $g(h(x))$, called "function composition", is sometimes written '$g \circ h$'. So, roughly, if g and h are recursive functions, then so is their (generalized) composition $g \circ h$.

This should be compared to structured programming's notion of linear concatenation (i.e., sequencing): First compute h; then compute g.

ii. *f is defined from g, h, i by conditional definition* =def

$$f(x_1, \ldots, x_k) = \begin{cases} g(x_1, \ldots, x_k), & \text{if } x_i = 0 \\ h(x_1, \ldots, x_k), & \text{if } x_i > 0 \end{cases}$$

Using our simplified notation, we can write this as:

$$f(\bar{x}) = \begin{cases} g(\bar{x}), & \text{if } x_i = 0 \\ h(\bar{x}), & \text{if } x_i > 0 \end{cases}$$

This should be compared to structured programming's notion of conditional branch (i.e., selection): If a Boolean test (in this case, "$x_i = 0$") is true, then compute g, else compute h.

iii. *f is defined from $g, h_1, \ldots, h_k, i$ by while-recursion* =def

$$f(x_1, \ldots, x_k) = \begin{cases} g(x_1, \ldots, x_k), & \text{if } x_i = 0 \\ f(h_1(x_1, \ldots, x_k), \ldots, h_k(x_1, \ldots, x_k)), & \text{if } x_i > 0 \end{cases}$$

Again, using our simplified notation, this can be written as:

$$f(\bar{x}) = \begin{cases} g(\bar{x}), & \text{if } x_i = 0 \\ f(\overline{h(\bar{x})}), & \text{if } x_i > 0 \end{cases}$$

And, again, this should be compared to structured programming's notion of while-loop (i.e., repetition): While a Boolean test (in this case, "$x_i > 0$") is true, compute h, but, when the test becomes false, then compute g).

An example of a while-recursive function might help to clarify the notion. The Fibonacci sequence is:

$$0, 1, 1, 2, 3, 5, 8, 13, \ldots$$

where each term after the first two terms is computed as the sum of the previous two terms. This can be defined recursively:

- First, write down 0, then 1.
- Each subsequent term in the sequence is the sum of the previous two terms.

This can be defined using a while-recursive formula as follows:

$$f(x) = \begin{cases} 0, & \text{if } x = 0 \\ 1, & \text{if } x = 1 \\ f(x-1) + f(x-2), & \text{if } x > 1 \end{cases}$$

We can make this look a bit more like the official definition of while-recursion by taking $h_1(x) = x \dot{-} 1$ and $h_2(x) = x \dot{-} 2$. In other words, the two base cases of f are projection functions, and the recursive case uses the predecessor function twice (the second time, it is the predecessor of the predecessor).

Given these definitions, we can now classify computable functions:

(a) *f is a while-recursive function* =def

    i. f is a basic function, or

    ii. f is defined from while-recursive functions by:

- generalized composition, or
- conditional definition, or
- while-recursion

    iii. Nothing else is while-recursive.

This is the essence of the Boehm-Jacopini Theorem: Any computer program (that is, any algorithm for any computable function) can be written using only the three rules of grammar: sequence (generalized composition), selection (conditional definition), and repetition (while-recursion).

(b)   i. *f is defined from g,h by primitive recursion* =def

$$f(x_1, \ldots, x_k, y) = \begin{cases} g(x_1, \ldots, x_k), & \text{if } y = 0 \\ h(x_1, \ldots, x_k, f(x_1, \ldots, x_k, y-1)), & \text{if } y > 0 \end{cases}$$

Using our simplified notation, this becomes:

$$f(\bar{x}, y) = \begin{cases} g(\bar{x}), & \text{if } y = 0 \\ h(\bar{x}, f(\bar{x}, y-1)), & \text{if } y > 0 \end{cases}$$

We can now more clearly see that this combines conditional definition with a computation of f based on f's value for its *previous* output.

This is the essence of recursive definitions of functions: Instead of computing the function's output based on its *current input*, the output is computed on the basis of the function's *previous output*.[27]

This can be usefully compared to structured programming's notion of a count-loop: while $y > 0$, decrement y and then compute f.

ii. *f is a primitive-recursive function* =def
- f is a basic function, or
- f is defined from primitive-recursive functions by:
  - generalized composition, or
  - primitive recursion
- Nothing else is primitive-recursive.

The primitive-recursive functions and the while-recursive functions overlap: Both include the basic functions and functions defined by generalized composition (sequencing). The primitive-recursive functions also include the functions defined by primitive recursion (a combination of selection and count-loops), but nothing else. The while-recursive functions include (along with the basic functions and generalized composition) functions defined by conditional definition (selection) and those defined by while-recursion (while-loops).

(c)   i. *f is defined from h by the μ-operator* [pronounced: "mu"-operator] =def
$$f(x_1,\ldots,x_k) = \mu z[h(x_1,\ldots,x_k,z) = 0]$$
where:
$\mu z[h(x_1,\ldots,x_k,z) = 0]$ =def

$$
\begin{cases}
min\{z: \begin{cases} h(x_1,\ldots,x_k,z) = 0 \\ \textbf{and} \\ (\forall y < z)[h(x_1,\ldots,x_k,y) \text{ has a value}] \end{cases} \}, & \textbf{if } \text{such } z \text{ exists} \\[2em]
\text{undefined}, & \textbf{if } \text{no such } z \text{ exists}
\end{cases}
$$

This is a complicated notion, but one well worth getting an intuitive understanding of. It may help to know that it is sometimes called "unbounded search" [Soare, 2012, 3284]. Let me first introduce a useful notation. If $f(x)$ has a value (that is, if it defined, or, in other words, if an algorithm that computes f halts), then we will write: $f(x)\downarrow$; and if $f(x)$ is undefined (that is, if it is only a "partial" function, or, in other words, if an algorithm for computing f goes into an infinite loop), then we will write: $f(x)\uparrow$.

Now, using our simplfied notation, consider the sequence

$$h(\bar{x},0),\ h(\bar{x},1),\ h(\bar{x},2),\ \ldots,\ h(\bar{x},n),\ h(\bar{x},z),\ \ldots,\ h(\bar{x},z')$$

---

[27] For a useful discussion of this, see [Allen, 2001].

Suppose each of the first *n* terms of this sequence halts with a non-zero value, but thereafter each term halts with value 0; that is:

$$h(\bar{x},0)\downarrow \;\neq\; 0$$

$$h(\bar{x},1)\downarrow \;\neq\; 0$$

$$h(\bar{x},2)\downarrow \;\neq\; 0$$

$$\ldots$$

$$h(\bar{x},n)\downarrow \;\neq\; 0$$

but:

$$h(\bar{x},z)\downarrow \;=\; 0$$

$$\ldots$$

$$h(\bar{x},z')\downarrow\,=0$$

What the μ-operator does is give us a description of that smallest z (i.e., the first z in the sequence) for which h halts with value 0. So the definition of μ says, roughly:

$\mu z[h(\bar{x},z) = 0]$ is the smallest z for which $h(\bar{x},y)$ has a non-0 value for each $y < z$
but $h(\bar{x},z) = 0$, **if** such a z exists;
otherwise, if no such z exists, then μz is undefined.

So, f is defined from h by the μ-operator if you can compute $f(\bar{x})$ by computing the smallest z for which $h(\bar{x},z) = 0$.

If h is intuitively computable, then, to compute z, we just have to compute $h(\bar{x},y)$, for each successive natural number y until we find z. So definition by μ-operator is also intuitively computable.

ii. *f is a partial-recursive function* =def
- f is a basic function, or
- f is defined from partial-recursive functions by:
  – generalized composition, or
  – primitive recursion, or
  – the μ-operator
- Nothing else is partial-recursive.

iii. *f is a recursive function* =def
- f is partial-recursive, and
- f is a total function
  (i.e., defined for all elements of its domain)

How are all of these notions related? As we saw, there is an overlap between the primitive-recursive functions and the while-recursive functions, with the basic functions and generalized composition in their intersection.

The partial-recursive functions are a superset of the primitive-recursive functions. The partial-recursive functions consist of the primitive-recursive functions together with the functions defined with the μ-operator.

The recursive functions are partial-recursive functions that are also total functions.

Finally, f is primitive-recursive if and only if f is count-program-computable. If f is primitive-recursive (i.e., if f is count-program-computable), then f is partial-recursive, which is logically equivalent to being while-program-computable. *And both of these are logically equivalent to being Turing-machine computable, lambda-definable, Markov-algorithmic, etc.*

7.8   The Halting Problem

Have we left anything out? That is, are there any other functions besides these? Yes! The "Halting Problem" provides an example of a *non-computable* function. Recall that a function *is* computable if and only if there is an algorithm (that is, a computer program) that computes it. So, is there a computer program (for example, a program for a Turing machine) that takes as input:

1. a computer program C, and

2. C's input (suppose that C's input is some integer, i)

and that outputs:

- "yes", if C halts on i

- "no", otherwise (that is, if C loops on i)?

Call this program "H". Because H's input is C and i, we can refer to H and its input as:

$$H(C, i)$$

And our question is: Does H exist? Can we write such a program?

In terms of the "function machine" illustration, above, we are asking whether there is a "function machine" (i.e., a computer) whose internal mechanism (i.e., whose program) is H. When you input the pair $\langle C, i \rangle$ to this "function machine" and turn its "crank", it should output "yes" if the function machine for C successfully outputs a value when you give it input i, and it should ouput "no" if the function machine for C goes into an infinite loop and never outputs anything.

(Here's another way to think about this: H is a kind of "super"-machine that takes as input, not only an integer i, but also another *machine C*. When you turn H's "crank", H first feeds i to C, and then H turns C's "crank". If H detects that C has successfully output a value, then H outputs "yes"; otherwise, H outputs "no".)

This would be very useful for introductory computer-programming teachers (or software engineers in general)! One thing you never want in a computer program is an

unintentional infinite loop. (Sometimes, you might want an *intentional* one: You *don't* want an ATM machine, for instance, to halt—you do want it to behave in an infinite loop so that it is always ready to accept new input from a new customer.) It would be very useful to have a *single*, handy program that could quickly check to see if *any* program that someone writes has an infinite loop in it. But no such program can be written!

It is important to be clear that it *can* be possible to write a program that will check if another program has an infinite loop. In other words, given a program C_1, there might be another program H_1 that will check whether C_1—*but not necessarily any other program*—has an infinite loop. What can't be done is this: To write a single program H that will take *any program C whatsoever* and tell you whether C will halt or not.[30]

Note that we can't answer the question whether C halts on i by just running C on i: If it halts, we know that it halts. But if it loops, how would we know that it loops? After all, it might just be taking a long time to halt.

There are two ways that we might try to write H.

1. You can imagine that $H(C, i)$ works as follows:

   - H gives C its input i, and then runs C on i.

   - If C halts on i, then H outputs "yes"; otherwise, H outputs "no".

   So, we might write H as follows:

   > **algorithm** $H_1(C, i)$:
   > **begin**
   >   **if** $C(i) \downarrow$
   >   **then** output 'yes'
   >   **else** output 'no'
   > **end.**

2. But here's another way to write H:

   > **algorithm** $H_2(C, i)$:
   > **begin**
   >   output 'no'; {i.e., make an initial guess that C loops}
   >   **if** $C(i) \downarrow$
   >   **then** output 'yes'; {i.e., revise your guess}
   > **end.**

---

[30]To be more precise, the difference between these two situations has to do with the order of the logical quantifiers: The Halting Problem asks this question: Does there exist a *single* program H, such that, for *any* program C, H outputs "yes" if C halts, else it outputs "no"? The answer to this question is "no"; no such program H exists. The other question, for which there can be a positive answer, is this: *Given* any program C, does there exist a program H (which will depend on what C is!) that outputs "yes" if C halts, else it outputs "no"? Note that different Cs might have different Hs. The answer to this question can be "yes".

Programs like this will prove useful in our later discussion of hypercomputation (that is, computation that, allegedly, cannot be modeled by Turing machines). But it won't work here, because we're going to need to convert H to another program called H^*, and H_2 can't be converted that way, as we'll see. More importantly, H_2 doesn't really do the required job: It doesn't give us a definitive answer to the question of whether C halts, because its initial answer is not really "no", but something like "not yet".

The answer to our question about whether such an H exists or can be written is negative: There is no such program as $H(C,i)$. In other words, $H(C,i)$ is a non-computable function. Note that it *is* a function: There exists a perfectly good set of input-output pairs that satisfies the definition of 'function' and that looks like this:

$$\{\langle C_1(i_1)\rangle, \text{"yes"}\rangle \,, \, \ldots \,, \, \langle C_j(i_k)\rangle, \text{"no"}\rangle \,, \, \ldots\}$$

Here is a sketch of a proof that H is not computable. The proof takes the form of a *reductio ad absurdum*: We will assume that H *is* computable, and derive a contradiction. If an assumption implies a contradiction, then—because no contradiction can be true—the assumption must have been wrong. So, our assumption that H is computable will be shown to be false.

So, assume that H is computable. Now consider another program, H^*, that works as follows: H^* is just like H, except that:

- if C halts on i, then H^* loops

    (remember: if C halts on i, H does *not* loop; it outputs "yes")

- and if C loops on i, then H^* outputs "no" (just like H does).

Here is how we might write H^*, corresponding to the version of H that we called 'H_1' above:

```
algorithm H₁ * (C, i):
begin
    if C(i) ↓
        then
            begin
                while true do begin end
            end
        else output 'no'
end.
```

Here, 'true' is a Boolean test that is always true. (As we noted earlier, you could replace it by something like '1=1', which is also always true.)

Note that we cannot write a version of H_2 that might look like this:

```
algorithm H₂ * (C, i):
begin
    output 'no'; {i.e., make an initial guess that C loops}
    if C(i) ↓
```

> **then** {i.e., if C halts, then loop}
>   **begin**
>   **while** true **do begin end**
>   **end**
> **end.**

Why not? Because if *C* halts, then the only output we will ever see is the message that says that *C* loops! That initial, incorrect guess is never revised. So, we'll stick with *H* (that is, with H_1) and with H^* (that is, with H_1*).

Note, by the way, that if *H* exists, so does H^*. That is, we can turn *H* into H^* as follows: If *H* were to output "yes", then let H^* go into an infinite loop. That is, replace *H*'s "output 'yes' " by H^*'s infinite loop.

Returning to our proof sketch, the next step is to code *C* as a number, so that it can be treated as input to itself.

What? Why do that? Because this is the way to simulate the idea of putting the *C* "machine" into the *H* machine and then having the *H* machine "turn" *C*'s "crank".

So, how do you "code" a program as a number? This is an insight due to Kurt Gödel. To code *any* text (including a computer program) as a number in such a way that you could also *de*code it, begin by coding each symbol in the text as a unique number (for example, using the ASCII code). Suppose that these numbers, in order, are $L_1, L_2, L_3, \ldots, L_n$, where L_1 codes the first symbol in the text, L_2 codes the second, $\ldots$, and L_n codes the last symbol.

Then compute the following number:

$$2^{L_1} \times 3^{L_2} \times 5^{L_3} \times 7^{L_4} \times \ldots \times p_n^{L_n}$$

and where p_n is the *n*th prime number, and where the *i*th factor in this product is the *i*th prime number raised to the L_ith power.

By the "Fundamental Theorem of Arithmetic",[31] the number that is the value of this product can be uniquely factored, those exponents can be recovered, and then they can be decoded to yield the original text.[32]

Now consider $H^*(C,C)$. (This step is called "diagonalization".) That is, suppose that you (1) code up program *C* as a Gödel number, (2) use it as input to the program *C* itself (after all, the Gödel number of *C* is an integer, and thus it is in the domain of the function that *C* computes, so it is a legal input for *C*), and (3) then let H^* do its job on *that* pair of inputs.

By the definition of H^*:

> if program *C* halts on input *C*, then $H^*(C,C)$ loops, and
> if program *C* loops on input *C*, then $H^*(C,C)$ outputs "no".

---

[31] http://mathworld.wolfram.com/FundamentalTheoremofArithmetic.html (accessed 19 November 2013).

[32] Gödel numbering is actually a bit more complicated than this. For more information, see "Gödel Number" [http://mathworld.wolfram.com/GoedelNumber.html] (accessed 19 November 2013) or "Gödel Numbering" [http://en.wikipedia.org/wiki/Godel_numbering] (accessed 13 November 2013). Turing has an even simpler way to code symbols; see [Turing, 1936, §5 ("Enumeration of Computable Sequences")]. For a comparison of the two methods, see [Kleene, 1987, 492].

Now code H^* by a Gödel number! And consider $H^*(H^*,H^*)$. (This is another instance of diagonalization. It may look like some kind of self-reference, but it really isn't, because the first occurrence of 'H^*' names a function, but the second and third occurrences are just numbers that happen to be the code for that function.)

In other words, (1) code up H^* by a Gödel number, (2) use it as input to the program H^* itself, and then let H^* do its job on that pair of inputs.

Again, by the definition of H^*:

if program H^* halts on input H^*, then $H^*(H^*,H^*)$ loops, and
if program H^* loops on input H^*, then $H^*(H^*,H^*)$ outputs "no".

But outputting "no" means that it halts!

So, if H^* halts (outputting "no"), then it loops, and, if H^* loops, then it halts. In other words, it loops if and only if it halts; that is, it *does* loop if and only if it does *not* loop!

But that's a contradiction. So, there is no such H^*. So, there is no such program as H. In other words, the Halting Problem is not computable.

7.9   Summary

Let's take stock of where we are. We asked whether computer science is the science of *computing* (rather than the science of *computers*. In order to answer that, we asked what computing, or computation, is. We have now seen one answer to that question: Computation is the process of executing an algorithm to determine the output value of a function, given an input value. We have seen how to make this informal notion precise, and we have also seen that it is an interesting notion in the sense that not all functions are computable.

But this was a temporary interruption of our study of the history of computers and computing. In the next chapter, we will return to the history by examining Alan Turing's formulation of computability.

7.10 NOTES FOR NEXT DRAFT

1. Need a section on "heuristic". Be sure to cite its first mention back in Ch. 3, and be sure to include a discussion of [Newell and Simon, 1976].

   Romanycia & Pelletier 1985 is probably the best survey of the varying meanings of 'heuristic'. Shapiro 1992 suggests that a heuristic is "any problem solving procedure that fails to be an algorithm, or that has not been shown to be an algorithm". It can fail to be an algorithm because "it doesn't terminate", or because "it has not been proved correct", and so on (depending, of course, on how you define 'algorithm').

   I prefer the following:

   > 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, 406].

   Being "good enough" is, of course, a subjective notion, but the idea behind it is related to Simon's notion of bounded rationality: We might not be able to solve P because of limitations in space, time, or knowledge. But we might be able to solve 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. Oommen & Rueda (in their abstract, p. 1) call the "good enough" solution "a *sub-optimal* solution that, hopefully, is arbitrarily close to the *optimal*."

2. For §7.6:
   Add as verb: "read" (?)
   Add as verb??: "P?" (i.e., Boolean test)

3. Question for discussion:
   Can scientific experiments be considered as (mathematical) functions? Harry Collins described an "experimenter's regress":

   > [Y]ou can say an experiment has truly been replicated only if the replication gets the same result as the original, a conclusion which makes replication pointless. Avoiding this, and agreeing that a replication counts as "the same procedure" even when it gets a different result, requires recognising the role of tacit knowledge and judgment in experiments. [*The Economist*, 2013]

   If you consider the experiment as a mathematical binary relation whose input is, let's say, the experimental set-up and whose output is the result of the experiment, then, if a replication of the experiment always gets the same result, then the relation is a function. In that case, does it make any sense to replicate an experiment in order to confirm it?

4. Cartoons from website for algorithms:

- willy n ethel, for recursion
- shoe, for named procedures
- hagar, for imprecision
- nancy and/or shoe and/or real-life one, for ambiguity
- nancy, for infinite loop
- zits, for detail

5. A function described extensionally is like a black box; we know the inputs and outputs, but not what (if anything) goes on inside it. A function described intensionally (via a formula) is less opaque and gives us more understanding of the relationship between the input and the outputs. A function described algorithmically gives us even more understanding, telling us not only what the relationship is, but giving explicit instructions on how to make the conversion from input to output. (For more on this notion of understanding in terms of the internal workings of a black box, see [Strevens, 2013]; this article also suggests an analogy between computation and causation—cf. Chs. 10, 12, 14, 16, and 21.)

6. On "algorithmic thinking":

> I think it is generally agreed that mathematicians have somewhat different thought processes from physicists, who have somewhat different thought processes from chemists, who have somewhat different thought processes from biologists. Similarly, the respective "mentalities" of lawyers, poets, playwrights, historians, linguists, farmers, and so on, seem to be unique. Each of these groups can probably recognize that other types of people have a different approach to knowledge; and it seems likely that a person gravitates to a particular kind of occupation according to the mode of thought that he or she grew up with, whenever a choice is possible. C.P. Snow wrote a famous book about "two cultures," scientific vs. humanistic, but in fact there seem to be many more than two. [Knuth, 1985, 171]

There is a saying that, to a hammer, everything looks like a nail (http://en.wikipedia.org/wiki/Law_of_the_instrument). This can be taken two ways: as a warning not to look at things from only one point of view, or as an observation to the effect that everyone has their own point of view. I take Knuth's remarks along the latter lines. And, of course, his eventual observation is going to be that computer scientists look at the world algorithmically. Given the wide range of different points of view that he mentions, one conclusion could be that, just as students are encouraged to study many of those subjects, so as to see the world from those points of view, so we should add algorithmic thinking— computer science—to the mix, because of its unique point of view.

Comparing mathematical thinking to algorithmic thinking, [Knuth, 1985, 181] finds several areas of overlap and two areas that differentiate the latter from the

former. The areas of overlap include manipulating formulas, representing reality, problem solving by reduction to simpler problems (which, by the way, is a form of recursion), abstract reasoning, dealing with information structures, and, of course, dealing with algorithms (presumably in a narrower sense). The two areas unique to algorithmic thinking are the "notion of 'complexity' or economy of operation..." (presumably, what is studied under the heading of "computational complexity" [Loui, 1996, Aaronson, 2013]) and—of most significance—"the dynamic notion of the *state* of a process: 'How did I get here? What is true now? What should happen next if I'm going to get to the end?' Changing states of affairs, or snapshots of a computation, seem to be intimately related to algorithms and algorithmic thinking".

7. On the difference between algorithm and program:
One way to consider the difference between and algorithm and a program (written in some programming language) is that the former is an abstract object and the latter is a concrete implementation of an algorithm. Here's a slightly different take on this:

> A program specifies in the exact syntax of some programming language the computation one expects a computer to perform. ...An algorithm may specify essentially the same computation as a specific program.... Yet, the purpose of an algorithm is to communicate a computation not to computers but to humans. [Dewdney, 1989, 1]

Dewdney seems to distinguish *three* things: a computation, an algorithm (for that computation), and a program (for that computation). What he calls a 'computation' seems to be more like what I would call an 'algorithm'. (There might also be a fourth thing: a computation being performed, or what is sometimes called a 'process'.) His notion of "algorithm" seems to be more concrete than mine, because it has to be communicated to a human, hence needs to be an expression in some humanly understandable language.

Further evidence that his notion differs from mine are the following remarks:

> A recipe, for example, is an algorithm for preparing food (assuming, for the moment, that we think of cooking as a form of computation). [Dewdney, 1989, 1]

Before continuing, here's something to think about that will help prepare you for the discussion in Ch. 10: *Is* it reasonable to think of cooking as a form of computation? Unlike other forms of computation that involve numbers or, more generally, symbols,[33] cooking involves physical objects, in particular, food. Arithmetic and word processing are typical kinds of things we do with computers. (So are drawing pictures and making music.) Do we expect computers to be able to prepare dinner for us?

But let's continue with the evidence I mentioned:

---

[33] A related question: Do computers deal with numbers? Or with numerals (that is, symbols for numbers)?

> A major feature shared by…[recipes] with all algorithms is the looseness of its description. …Such things do not bother experienced cooks. Their judgment and common sense fill the obvious gaps in the recipe. [Dewdney, 1989, 3]

But algorithms, as standardly conceived, are never "loose"! They never require judgment on the part of the agent that executes them.

Another consideration raised by the previous quotation is this: If cooking is a form of computation, what plays the role of the computer? Dewdney makes it sound like *the cook* is the computer.

8. On the intenTionality of an algorithm:
   Somewhere, need to discuss this problem, possibly in Ch. 17 if not here: Must algorithms be "intentional" in the sense that they are *for some purpose*, designed *to solve some particular problem*? Is an algorithm that is designed to allow a robot to solve a blocks-world problem but that has no feedback to enable it to determine if it has dropped a block (and has therefore failed to accomplish its task) therefore not an algorithm? Is a recipe (let's assume for the moment that it's an algorithm) for making Hollandaise sauce but that is executed on the moon and therefore fails for lack of oxygen (or whatever) not therefore an algorithm? Is an algorithm for computing GCD that is given base-2 numerals as input instead of base-10 numerals therefore not an algorithm? [Rescorla, 2013]. Is an algorithm for playing a particular chess game but that also simulates a particular Civil War battle one algorithm or two? [Fodor, 1978, 232]. Is an algorithm for "analyzing x-ray diffraction data that…also solves Sudoku puzzles" [Elser, 2012, 420] one algorithm or two? Is this any different from the situation in (applied) mathematics, where a mathematical structure designed for one purpose is found useful elsewhere? (E.g., the application of group theory in physics.) Perhaps all that means is that the two application situations share a common (mathematical strucure). So, perhaps, it is a common, i.e., abstract, *algorithmic* structure that is the "real" algorithm, and not some specific interpretation or use of it (which might require a different input-output encoding or interpretation). This is also related to the more general question of the nature and role of (mathematical or other scientific) models. (For other ideas along these lines—put in Further Sources of Info—see:
   Hayes, Brian (2004), "Small-Town Story", *American Scientist* (March-April): 115–119; https://www.americanscientist.org/issues/pub/small-town-story (accessed 14 March 2014)

   - For some ideas on the intentionality of algorithms, see Hayes's comments on human vs. ant behavior models.

   - Re: [Rescorla, 2013]:
     [Stewart, 2000, 98] suggests that '3 is not an integer power of 2' is true in base-10 arithmetic but false "in the set of integers 'modulo 5' ". More importantly, he says that "This doesn't mean that [the first statement] is wrong, because the context has been changed. It just means that I have to

be careful to define what I'm talking about." Similarly, we must distinguish between these two statements:

(a) The *number* represented by the *numeral* '3' is an integer power of the *number* represented by the *numeral* '2'.

(b) The *number* canonically represented as '{{{}}}' is an integer power of the *number* canonically represented as '{{}}'.

The first of these depends on the base that the symbols are in; the second doesn't. Or are neither about symbols, and both are about numbers?

9. In [Soare, 2012, 3289]'s characterization, the output of a Turing machine "is the total number of 1's on the tape." So, the key to determining what is computable (that is, what kinds of tasks are computable) is finding a coding scheme that allows a sequence of '1's—that is, an integer—to be *interpreted as* a symbol, a pixel, a sound, etc.

10. Not sure if this goes here, elsewhere, or nowhere, because it has more to do with causation than computation, but it may have some relationship to ethical issues in robotics, and it is certainly related to issues in (math) education: If I use a calculator (or a computer, or if a robot performs some action), who or what is "really" doing the calcuation (or the computation, or the action)—Is it the calculator (computer, robot)? Or me? Compare this real-life story: I was making waffles "from scratch" on a waffle iron. The 7-year-old son of friends who were visiting was watching me and said, "Actually, *you're* not making it; it's the thing [what he was trying to say was that it was *the waffle iron* that was making the waffles]. But you set it up, so you're the cook."

11. The Halting Problem is not the only non-computable function. There are many more; in fact, there are an infinite number of them. Two other famous ones are Hilbert's 10th Problem and the Busy Beaver function.

The first of these was the 10th problem in the famous list of math problems that Hilbert presented in his 1900 speech as goals for 20th century mathematicians to solve [Hilbert, 1900]. It concerns Diophantine equations, that is, equations of the form $p(x_1, \ldots, x_n) = 0$, where p is a polynomial with integer coefficients. Hilbert's 10th Problem says:

> Given a diophantine equation with any number of unknown quantities and with rational integral numerical coefficients: *To devise a process according to which it can be determined by a finite number of operations whether the equation is solvable in rational integers.* [Hilbert, 1900, 458]

In the early 1970s, Yuri Matiyasevich, along with the mathematicians Martin Davis and Julia Robinson, and the philosopher Hilary Putnam, proved that there was no such "process" (that is, no such algorithm). (For more information, see: http://en.wikipedia.org/wiki/Hilbert's_tenth_problem
and "Hilbert's Tenth Problem page" at:
http://logic.pdmi.ras.ru/Hilbert10/ (both accessed 16 December 2013).)

The Busy Beaver function has been described as follows:

> $\Sigma(n)$ is defined to be the largest number which can be computed by an *n*-state Turing machine.... [Chaitin, 1987, 2]

GIVE DEWDNEY DESCRIPTION, TOO. It was first described, and proved to be non-computable, by Tibor Radó [Radó, 1962].[34] (For more information, see [Suber, 1997b] and
http://en.wikipedia.org/wiki/Busy_beaver
(accessed 16 December 2013).)

12. On inputs and outputs (relevant to discussion above about whether the informal notion of algorithm requires these; also relevant to discussions in later chapters on relation of program to world, Cleland's views, etc.):

    > Do computations have to have inputs and outputs? The mathematical resources of computability theory can be used to define 'computations' that lack inputs, outputs, or both. But the computations that are generally relevant for applications are computations with both inputs and outputs. [Piccinini, 2011, 741, note 11]

    See comment in Ch. 17 at NOTES§6.

13.   While the "official" Turing thesis is that every computable function is Turing computable, Turing's informal argument in favor of his thesis justifies a stronger thesis: every algorithm can be simulated by a Turing machine. [Gurevich, 2000, 77]

    But note that Gurevich says 'simulated': The actual algorithm running on the TM will typically not be the same as the algorithm being simulated, because:

    > TMs deal with single bits. Typically there is a huge gap between the abstraction level of the program and that of the TMs; even if you succeed in the tedious task of translating the program to the language of TMs, you will not see the forest for the trees. Can one generalize Turing machines so that any algorithm, never mind how abstract, can be modeled by a generalized machine very closely and faithfully? The obvious answer seems to be no. The world of algorithms is too diverse even if one restricts attention to sequential algorithms.... [Gurevich, 2000, 78]

    Therefore, "there is more to an algorithm than the function it computes" [Gurevich, 2000, 80].

    Given a computable function f, there will be an algorithm that computes it. A Turing machine for f will be an example of such an algorithm (or, if you

---

[34]This paper, by the way, is a wonderfully readable introduction to the Busy Beaver "game". §II gives a very simple example of a Turing machine aimed at students, and §§I–III and, especially, §VIII are amusing and well worth reading!

prefer, a universal Turing machine with a program for computing f). But there will be many algorithms that compute f, and not all of them will be expressible in the language of Turing machines. So, there's a difference between an algorithm and a program (which can be thought of as an algorithm expressed in some programming language).

An algorithm is something abstract that can be implemented as different programs in different programming languages. Any program that implements a particular algorithm at best *simulates* any other program that implements that algorithm.

Moreover, Turing-machine algorithms (according to [Gurevich, 2011, 6]) are "symbolic sequential" algorithms. What about ruler-and-compass algorithms? [Toussaint, 1993]: "[T]he functions computed by ruler-and-compass algorithms are not Turing computable" [Gurevich, 2011, §3]. Why not? One possible way to "reduce" the latter to the former might be by way of an output plotter that can simulate ruler-and-compass constructions. Or, more simply (in one sense of 'simply'!), if all such (geometric) constructions can be expressed in terms of analytic geometry, then they would seem to be able to be expressed in terms of functions (over real numbers), and hence be (Turing-)computable. But let's suppose that Gurevich is right: Surely, there's something "algorithmic" about ruler-and-compass constructions. They would seem to satisfy the informal definition of 'algorithm' given in §7.5, differing only in what counts as a "statement", "rule", or "instruction". And, given the Boehm-Jacopini Theorem and the third insight of §7.6.3, the core difference is' what counts as a "primitive" or "basic" statement, rule, or instruction. For Turing machines, those are "print" and "move"; for recursive functions, they are "successor", "predecessor", and "projection"; for ruler-and-compass constructions, they would be something like drawing an arc with a certain radius, drawing a line of a certain length, etc. Under this view, are *all* algorithms expressible in the language of Turing machines? (see also [Goodman, 1987, §4])

Gurevich distinguishes pretty clearly "algorithms" from "Turing machines", pointing out that, although for any (intuitively) computable function there is an algorithm—in particular, a TM—that computes it, it doesn't follow that a TM can compute every *algorithm*; at best, TMs can *simulate* any algorithm. One example is Euclid's GCD algorithm, which comes in both a "subtraction" and a "division" form (division just being repeated subtraction). A TM probably can't implement the latter (in any "direct" way), because its only way of (indirectly) implementing division would be by implementing repeated subtraction.

He also says that ruler-and-compass constructions are not Turing computable, nor is Gaussian elimination. He gives no justification for either statement. On the face of it, the former might be TM-computable if you hook a plotter, or even a ruler and a compass, up to it and let the TM control them. Even easier, translate every ruler-and-compass algorithm into the language of analytic geometry, which, in turn, is expressible in terms of (computable) functions (on

the reals). Seems to me that that makes ruler-and-compass constructions TM-computable. (Although, consistent with the remarks in the previous paragraph, the actual ruler-and-compass *algorithms* are only being simulated.) As for Gaussian elimination (an algorithm for dealing with matrices), I can't find any source other than Gurevich who says that they aren't (TM-)computable.

> But ruler and compass was their computer. It was their true computer. They really computed that way. [Gurevich, 2013]

([Feferman, 1992, 318] makes a similar remark.)

Here's another statement of the issue:

> There may be a huge difference between an algorithm and the simulating Turing machine. The poor Turing machine may work long and hard, with its head creeping painfully slowly back and forth on that long narrow tape, in order to simulate just one step of the given algorithm. ...The problem arises whether there is a machine model such that any algorithm, however abstract, can be simulated *step-for-step* by a machine of that model. Let me call such a model universal with respect to algorithms. Turing's machine is universal with respect to computable functions, but not with respect to algorithms. [Gurevich, 1999, 95–96, my italics]

14. In a section titled "The flaw in Church's thesis", Robert I. Soare says:

    > If the basic steps are stepwise recursive, then it follows easily by the Kleene normal form theorem...that the entire process is recursive. *The fatal weakness in Church's argument was the core assumption that the atomic steps were stepwise recursive, something he did not justify.* [Soare, 2012, 3286, my italics]

    Church's theory, the λ-calculus, had as its basic, or atomic, steps formal operations on function formulas that some people—Gödel in particular—did not find to be intuitively computable.[35] The same could be said even for Gödel's own theory of recursive functions. But Turing's basic operations were, by design, simple things that any human could easily do: put a mark at specific location on a piece of paper, and shift attention to a different location.

15. For statements of equivalence of general recursive, μ-recursive, λ-definable, etc., see [Soare, 2012, 3284].

16. One of the essential features of an algorithm is that it is an explicit set of instructions (for doing something). (Exactly what counts as being an "instruction" and as being "explicit" is what's at the heart of the formal notion of "algorithm".) Consider this observation:

---

[35]For an elemetary introduction to the lambda-calculus, see [PolR, 2009, §"Alonzo Church's Lambda-Calculus"].

> Computer-driven machines can now carry out a huge range of
> very highly skilled tasks, from navigating and landing aircraft,
> to manufacturing and assembling a wide range of products.
> [MacFarlane, 2013, 18]

What makes this possible is that we have figured out how to express the detailed
instructions for how to do these things—we have figured out algorithms for them;
we not only know *how* to do them, we know *how to tell* someone or something
how to do them.

And one way of characterizing computer science is as the discipline that is
concerned with determining which tasks are such that we can express how to
accomplish them in the language of an algorithm; that is, computer science is
concerned with determining which tasks are computable.

17. On the difference between functions given by formulas and functions given by
algorithms:
[Pearl, 2000] "considers the difference between two representations of simple
algebraic relations, which he terms 'equations versus diagrams,' contrasting:

$$Y = 2X$$
$$Z = Y + 1$$

with

$$X \rightarrow \times 2[Y] \rightarrow +1[Z]$$

The former describes relations between variables; the latter specifies a simple
computer program, in the form of a flowchart, indicating the order in which
operations are carried out" [Chater and Oaksford, 2013, 1172].[36] (The former
could even be simplified to $Z = 2X + 1$.) The flowchart, or algorithmic, version
allows for "interventions"—changes that can be inserted in between steps of the
algorithm; the equation, or formula, version does not allow for that, being merely
a static description of the output of the function.

18.   In general[,] algorithms perform tasks, and computing functions is a
        rather special class of tasks. [Gurevich, 2012, 2]

Unless every task can be expressed as (that is, coded as) a (mathematical)
function. This is also relevant to Cleland's concerns; see Ch. 10. Even Gurevich
himself alludes to this, when he says:

> The validity relation on first-order formulas can be naturally
> represented as a real number, and the Entscheidungsproblem becomes

---

[36]The interpretation of the flowchart version is something like this:

(a) input X

(b) multiply X by 2; store in Y

(c) add 1 to Y; store in Z

Note that this algorithm does not appear to have an output; see §7.5.

> whether this particular real number is computable. [Gurevich, 2012, 2]

But, by our "first great insight" (§7.6.1), anything that we want to process computationally can be encoded as a real number. So, perhaps "computing functions" is not so "special" after all.

19. To my list of features of algorithms in §7.5, above, [Gurevich, 2012, 4] adds "isolation":

> Computation is self-contained. No oracle is consulted, and nobody interferes with the computation either during a computation step or in between steps. The whole computation of the algorithm is determined by the initial state.

Does anyone else cited above mention this? How does (the lack of) this relate to interactive computing/hypercomputation? (See Ch. 11.)

Does the use of such an oracle mean that the computation is no longer purely automatic or mechanical? (But why not?)

Also, while I'm thinking about it, what about Gurevich's first "constraint": "Computation is symbolic (or digital, symbol-pushing)" (p. 4). That is, the syntactic nature of computation. Is that part of my characterization above? (For more on this, see papers by Rescorla.)

20.   > [C]onflating algorithms with Turing machines is a misreading of Turing's 1936 paper.... Turing's aim was to define *computability*, not algorithms. His paper argued that every function on natural numbers that can be computed by a human computer... can also be computed by a Turing machine. There is no claim in the paper that Turing machines offer a general model for algorithms. [Vardi, 2012]

Vardi goes on to cite Gurevich's idea that algorithms are "abstract state machines", whose "key requirement is that one step of the machine can cause only a bounded local change on... [a] state" (which is an "arbitrary data structure"). Note that this corresponds to Turing's analysis and to our analysis in §7.5.

He also cites Moschovakis's idea that algorithms are "recursors": "recursive description[s] built on top of arbitrary operations taken as primitives." Note that this corresponds to recursive functions and the lambda-calculus, as we discussed in §7.7.

And he then observes, in line with the proofs that Turing machines, the lambda-calculus, recursive functions, etc., are all logically equivalent, that these *distinct* notions are analogous to the wave-particle duality in quantum mechanics: "An algorithm is *both* an abstract state machine and a recursor, and neither view by itself fully describes what an algorithm is. This *algorithmic duality* seems to be a fundamental principle of computer science."

21. Another informal notion of "algorithm":

> (a) [A] method for answering any one of a given infinite class of questions…is given by a set of rules or instructions, describing a procedure that works as follows. (b) *After* the procedure has been described, [then] if we select *any* question from the class, the procedure will then tell us how to perform successive steps, so that after a finite number of them we will have the answer to the question selected. (c) In particular, immediately after selecting the question from the class, the rules or instructions will tell us what step to perform first, unless the answer to the question selected is immediate. (d) After our performing any step to which the procedure has led us, the rules or instructions will *either* enable us to recognize that now we have the answer before us and to read it off, *or else* that we do not yet have the answer before us, in which case they will tell us what step to perform next. (e) In performing the steps, we simply follow the instructions like robots; no ingenuity or mathematical invention is required of us. [Kleene, 1995, 18]

So, an algorithm (informally) is:

a A *set* of *rules or instructions* that describes *a procedure*. The procedure is one thing; its description is another; the latter is a set of imperative sentences.

b Given a class of questions CQ and a procedure PCQ for answering any member: $\forall q \in CQ$[PCQ gives a *finite sequence* of steps (described by its rules) that answers q]

So, the finiteness occurs in the execution of PCQ (not necessarily in PCQ itself). And PCQ does not depend on q, only on CQ.

That does suggest that an algorithm is "intentional" in the sense of having a goal or purpose.

c The algorithm inputs question q, and either outputs q's answer (base case), or else outputs the first step to answer q (recursive case).

d If it is the recursive case, then, presumably, q has been reduced to a simpler question and the process repeats until the answer is produced as a base case.

Moreover, the answer is immediately recognizable. That does not necessarily require an intelligent mind to recognize it. It could be merely that the algorithm halts with a message that the output is, indeed, the answer. (The difference between this and a trial-and-error machine is that here the output is of the form: The answer to q is q_i, where q_i is the answer and the algorithm halts, or q_i is a one-step-simpler question and the algorithm tells us what the next step is. In a trial-and-error machine, the output is of the form: My current guess is that the answer to q is q_i and the algorithm tells us what the next step is.

e The algorithm is "complete" in the sense that it contains all information for executing the steps. We, the executor, do not (have to) supply anything else. In

particular, we do not (have to) accept any further, unknown/unforeseen input
from any other source (that is, no oracle or interaction)

22. On the relation of algorithm, decidability, and computability:
    Questions come in two kinds: true-false (yes-no) questions and "what"
    questions. For example, "Is 2+2=4?" is a true-false question, and "What is 2+2?"
    is a "what" question. A *relation* $R(a_1, \ldots, a_n)$—that is, a "Boolean" proposition
    that is either true or else false (a yes-no question)—is "decidable" means that
    there is an algorithm (a decision procedure) that "decides" its truth value (that
    answers the question). A *function* $f(a_1, \ldots, a_n)$—that is, a "what" question—is
    "computable" means that there is an algorithm (a computation procedure) that
    "computes" its output value (that answers the question). [Kleene, 1995, 17]

23. [Kleene, 1995] shows how to "compile" (or translate the language of) recursive
    functions into (the language of) Turing machines, that is, how a Turing machine
    can compute recursive functions.

24. Nice terminology (due to Bill Duncan? Or did he get it from me?):
    Functions *relate* inputs to outputs;
    Algorithms (describe how to) *transform* inputs into outputs.

25.   [A function is a set of ordered pairs...[satisfying the unique output
        condition]. ...A computable function...is a mapping [that is, a
        function] that can be specifed in terms of some *rule* or other,
        and is generally characterized in terms of what you have to do
        to the first element to get the second [where the rule must satisfy
        the constraints of being an algorithm]. ...[A] noncomputable
        function...is an infinite set of ordered pairs for which no rule
        can be provided, not only now, but in principle. Hence **its
        specification consists simply and exactly in the list of ordered
        pairs.** [Churchland and Sejnowski, 1992, 62, their italics, my
        boldface]

    Note that this is almost the same as Chaitin's definition of a random number.

26. On input and output:

        There remains, however, the necessity of getting the original
        definitory information from outside into the device, and also of
        getting the final information, the results, from the device to the
        outside. [von Neumann, 1945, §2.6, p. 3].

    In other words, there has to be input and output. The computer has to have
    something to work on ("definitory information", or input), and it has to let the
    human user know what it has computed ("the final information, the results", or
    output). It can't just sit there silently computing.

27. A function from a set *A* to a set *B* is a *regularity* between the two sets (I borrow
    the term from [Shagrir, 2006b, 402]. It is not random, because the output of a

given input must always be the same; a random relation between *A* and *B* would not guarantee this. And a non-random, but non-functional, relation would give different (if not random) outputs for some inputs (as in the square-root relation, which, for a given positive integer input gives both positive and negative square roots as output).

28. David Marr CITE analyzed information processing into three levels: computational (what a system does), algorithmic (how it does it), and physical (how it is implemented). But a different terminology, more consistent with what we are using in this book, would identify four levels: functional (a description of a system in input-output terms; this would seem to align with Marr's "computational" level), formula (or algebraic) (a description using an (algebraic) formula that relates the inputs and outputs), algorithmic (a description of how to compute the formula; this might also be called the 'computational' level, but it seems to align more with Marr's algorithmic level), and implementional (a description of a specific physical device).

29. The notion that algorithms and Turing machines are not exactly the same thing (as Gurevich argues) is also explored in [Ross, 1974]:

> Does it [Turing's analysis] tell us what an algorithm is? [Ross, 1974, 517]

He points out that this depends on what we would mean by 'equivalent' when we ask whether a given program (expressing an algorithm) is equivalent to a Turing-machine program. Does this mean mere input-output equivalence? Or something stronger (some kind of "procedural" equivalence)?

And, relevant to hypercomputation, he observes that:

> Ultimately, the truth or falsity fo Church's thesis seems to rest heavily upon what powers are attributable to the idealized agents who are to perform the computations specified by algorithms. For example, if we were to consider God as calculator it might not be improper to consider as algorithmic instructions involving the performance of infinitely many operations. [Ross, 1974, 519]

And, relevant to Turing's "strange inversion", he observes:

> Almost all intuitively written algorithms require the agent to know what he is doing. That is, most intuitive algorithms require the agent to understand the meanings of the symbols he is manipulating and usually a great deal more as well. ... Doubt about whether powers of comprehension used by human agents give no computational powers beyond those of a pigeon leads to skepticism concerning Church's thesis. [Ross, 1974, 520]

On pigeons, Ross says, in a footnote, that

> It has been shown by P. Suppes that a pigeon can be taught to perform all the tasks necessary to follow the program of a universal Turing machine, hence any Turing program. [Ross, 1974, 520n1].

Shades of Searle's Chinese Room Argument!

30. On whether an algorithm requires output:

> **Read** is the companion process to **write**, each being necessary to make the other useful. **Read** only obtains what was put into expressions by **write** at an earlier time; and a **write** operation whose result is never **read** subsequently might as well not have happened.

The last clause suggests that output is not necessary.

31. Nice quotes relevant to Turing's strange inversion:

> One may be a mathematician of the first rank without being able to compute. *It is possible to be a great computer without having the slightest idea of mathematics.*
> —Novalis (Georg Philipp Friedrich Freiherr von Hardenberg), cited in [Ralston, 1999, 173]

> Every cell in my body knows how to replicate DNA yet I'm not in on it so I have to spend hours studying it.
> —anonymous meme found on the Web, 2015.

32. Leibniz isolated some general features of algorithmic procedures....
    ...an algorithmic procedure must *determine* completely what actions have to be undertaken by the computing agent. ...the instructions of a calculation procedure can be viewed as prescribing *operations on symbolic expressions* in general, and not just on numerical expressions. ...only physical properties of symbols—such as their shape and arrangement—and not, e.g., their meaning, play a role in a calculation process. ...only elementary intellectual capabilities are required on the part of the executor of a calculation procedure....
    [Spruit and Tamburrini, 1991, 7–8].

33. Computability is a relative notion, not an absolute one. All computation, classical or otherwise, takes place relative to some set or other or primitive capabilities. The primitives specified by Turing in 1936 occupy no privileged position. One may ask whether a function is computable relative to these primitives or to some superset of them. [Copeland, 1997, 707]; see also [Copeland and Sylvan, 1999, 46–47].

Compare the situation with Euclidean geometry: An angle cannot be trisected *using only compass and straightedge*; but, of course, using a protractor to measure its size and then calculating one-third of that will do the trick.

34. [Copeland, 2000] identifies 'effective' with 'mechanical', and defines "a method, or procedure, M, for achieving some desired result" to be effective iff:

   1 M is set out in terms of a finite number of exact instructions (each instruction being expressed by means of a finite number of symbols);

   2 M will, if carried out without error, always produce the desired result in a finite number of steps;

   3 M can (in practice or in principle) be carried out by a human being unaided by any machinery save paper and pencil;

   4 M demands no insight or ingenuity on the part of the human being carrying it out.

35. Following email discussions with Peter Denning, I'm inclined to add a fifth "insight" to my previous list: Turing machines can be physically implemented. (See also Ch. 14, Notes for Next Draft No. 8.)

   This brings in one engineering aspect of computer science. In fact, there are two engineering aspects. One might be called the software-engineering aspect: Although only two nouns, two transitive verbs, and three rules of grammar are *needed*, other nouns (data structures), other verbs (named subroutines), and other rules of grammar (other control structures such as exits from loops, controlled gotos, etc.) can be added to allow for greater expressivity and control of complexity.

   The other aspect of engineering is the physical kind (the fifth insight). But it brings in its train limitations imposed by physical reality: limitations of space, time, memory, etc. Issues concerning what is feasibly or efficiently computable in practice (as opposed to what is theoretically computable in principle)—complexity theory, worst-case analyses, etc.—and issues concerning the use of heuristics come in here.

36. On formulas vs. algorithms:
   Some functions expressed as formulas might be seen as containing an implicit algorithm for how to compute them:

   > [A] term in the series for arctan 1/5 can be written either as $(1/5)^m/m$ or as $1/(m5^m)$. Mathematically these expressions are identical, but they imply different computations. In the first case you multiply and divide long decimal fractions; in the second you build a large integer and then take its reciprocal. [Hayes, 2014a, 344]

   But these formulas can only be interpreted as algorithms with additional information about the order of operations (roughly, do things in innermost parentheses first, then do exponentiations, then multiplication and division (from left to right), then addition and substraction (from left to right)).

37. On the intentionality of algorithms (again):
   First, recall Robin Hill's point about this: Algorithms need to be expressed in the

form "To do A, do the following. . .".

Second, consider these observations:

> The syntactic behaviour of a Turing machine is fully described by
> its machine table. But we need also an explanation of why the
> Turing machine computes the arithmetic function that it does. We
> have an explanation of that only when *we attribute* various semantic
> properties to collections of the quadruples. [Peacocke, 1999, 197, my
> italics]

This is related to my experience of being told how to use a spreadsheet by
following "meaningless" instructions that turned out to be instructions for
adding. On this, see Dennett's "Turing's Strange Inversion".

Why do we need an explanation of why the Turing machine is computing the
function that does? Is that different from merely asking what it is doing? I think
it does go beyond that. Given a Turing machine that does something, we can ask
what it is doing. We can answer that by describing its machine table (a purely
syntactic answer). We can also answer by saying that it is adding two numbers.
(Compare: When I add two numbers using a calculator, we can describe what I
am doing either by saying that I am pressing certain buttons on the calculator in
a certain sequence or by saying that I am adding two numbers.) But it is a further
question to ask *why* printing and moving in that way (or pushing those buttons)
results in adding two numbers.

Peacocke's emphasis on the semantic properties (the semantic interpretation of
what the Turing machine (or we) are doing) as being attributed by us is certainly
in line with Searle's view of things.

But, in line with Fodor's chess-Civil War example, the single thing that the
Turing machine is doing might be interpreted in two different (albeit isomorphic)
ways. Is one thing being done, or two?

One way in which the external semantic interpretation can be seen as being given
is in terms of identifying certain groups of operations as constituting subroutines
that can be named. Identifying certain groups as accomplishing a subtask seems
to me to be syntactic. Giving such a group a name borders on being semantic.
But not if the name is something like 'operation 7B'—only if the name is
something like 'adding two numbers' and then only if that name is associated
with other information (see my arguments about the ability of a computer to
understand what it is doing, in, for example, [Rapaport, 1988]).

> Egan   says   that   a   description   of   a   device   as   computing   a

mathematical function is an environment-independent description. [Peacocke, 1999, 198].

Egan's statement is certainly in line with objections to Cleland's views about the Church-Turing Computability Thesis (see §10.5.1). But Peacocke objects:

> The normal interpretation of a Turing machine assigns the number 0 to a single stroke '|', the number 1 to '||', the number 2 to '|||', and so on. But there will equally be an interpretation which assigns 0 to a single stroke '|', and then assigns the number 2 to '||', the number 4 to '|||', and generally assigns $2n$ to any symbol to which the previous interpretation assigns n. Under the second interpretation, the Turing machine will still be computing a function. ... What numerical value is computed, and equally which function is computed, by a given machine, is not determined by the purely formal characterization of the machine. There is no such thing as purely formal determination of a mathematical function. ... [W]e can say that a Turing machine is really computing one function rather than another only when it is suitably embedded in a wider system. [Peacocke, 1999, 198–199].

The description of this single Turing machine as either computing n or $2n$ is like the chess-Civil War situation. Again, we can say that "the" function that this Turing machine is computing is a function whose outputs are strokes, or a function whose outputs are n, or a function whose outputs are $2n$. There is no single, "correct" answer to the question "What is this Turing machine computing?". We can fix an answer by "embedding" the Turing machine in a "wider system"; this is what I advocate in my argument that a computer can understand what it is computing.

It is also reminiscent of a point made in [Frenkel, 2013]: He considers a simple equation, like $y^2 + y = x^3 + x^2$, and asks

> But what kind of numbers do we want x and y to be? There are several choices: one possibility is to say that x and y are natural numbers or intergers. Anotehr possibility is to say rational numbers. We can also look for solutions x, y that are real numbers, or even complex numbers... (p. 83).

He continues:

> [W]hen we talk about solutions of such an equation, it is important to specify to what numerical system they belong. There are many choices, and different choices give rise to different mathematical theories. (p. 99)

Here's a simpler example: What are the solutions to the equation $x^2 = 2$? In the rational numbers, there is no solution; in the positive real numbers, there is one solution; in the (positive and negative) real numbers, there are two solutions.

Similarly, $x^2 = -1$ has no solution in the real numbers, but two solutions in the complex numbers. Deciding which "wider" number system the equation should be "embedded" in gives different "interpretations" of it.

But does this mean that "content" or "interpretation" is necessarily part of computation? Surely, a Turing machine that outputs sequences of strokes does just that, and whether those strokes should be interpreted as n or $2n$ is a separate matter. Similarly, "the" solution to $x^2 = c$ is $\sqrt[2]{c}$; whether we assign a rational, real, or complex number to that solution is a separate matter.

[Sprevak, 2010, 260, col. 1, my italics] discusses " 'the received view...that ...a necessary condition on any *physical* process counting as a computation is that it possess representational content". A physical process is what it is, so to speak, independent of anything that it might "mean", "be about", or "represent". But, according to Sprevak's understanding of what he calls the received view, whatever it is by itself, so to speak (syntactically?), it isn't a *computation* unless it is about something. Hence Hill's insistence on the intentionality: 'Do X' might be a mere physical process, but 'To accomplish A, do X' is a computation.

One reason in support of this received view that Sprevak offers, and against Egan's interpretation of Marr, is that "mathematical computation theory does not, by itself, have the resources to explain how the visual system works. ... Marr needs some way of connecting the abtract mathematical descriptions to the nuts and bolts of physical reality" [Sprevak, 2010, 263, col. 1]. But the mathematical theory *does* have the resources; that's the point of [Wigner, 1960]. (It may still be a puzzle *how* it does, but there's no question *that* it does.)

[Sprevak, 2010, §3.2, p. 268] goes on to argue that

> one cannot make sense of I/O equivalence without requiring that computation involves representational content.
>
> Consider two computational systems that perform the same numerical calculation. Suppose that one system takes ink-marks shaped like Roman numerals (I, II, III, IV, ...) as input and yields ink-marks shaped like Roman numerals as output. Suppose that the other system takes ink-marks shaped like Arabic numerals (1, 2, 3, 4, ...) as input and yields ink-marks shaped like Arabic numerals as output. Suppose that the two systems compute the same function, say, the addition function. What could their I/O computational equivalence consist in? Again, there may be no physical or functional identity between their respective inputs and outputs. The only way in which the their inputs and outputs are relevantly similar seems to be that they represent the same thing.

But there is a difference between *what* a system is doing and whether *two* systems are doing the *same* thing: The two addition algorithms (Roman and Arabic)

arent' doing the (same) *identical* thing, but they *are* doing the same (*equivalent*) thing. It is the *sameness* that depends on the semantics.

He goes on to say:

> Two physical processes that are intrinsic physical duplicates may have different representational contents associated with them, and hence different computational identities. One physical process may calculate chess moves, while a physical duplicate of that process calculates stock market predictions. We seem inclined to say that, in a sense, the two processes compute different functions, yet in another sense they are I/O equivalent. Appeal to representational content can accommodate both judgements. [Sprevak, 2010, 268, col. 2]

But this is a different case: identical algorithm but different task. The other case is different algorithm but same (input-output–equivalent) task.

38.   ...[R]ealism about computation...is the view that whether or not a particular physical system is performing or implementing a particular computation is at least sometimes a fact that obtains independently of human beliefs, desires and intentions. Unless we have a precise account of implementation it will not be possible to decide whether or not realism about computation is correct just because it will not be clear what 'computation' means. [Ladyman, 2009, 377]

First, this is relevant to the Fodorian chess-Civil War example: Is it independent of human beliefs, desires, and intentions that a particular computer is playing chess rather than rehearsing a Civil War battle? It certainly seems to be the case that the internal computations are thus independent even if the input-output is not.

Second, the reason that the meaning of 'computation' is relevant has to do with this observation:

> The idea of computation bridges the abstract and the concrete worlds. For example, suppose somebody claims: 'The computation took ten minutes'. He or she is clearly referring to a concrete property of a particular process in space and time. On the other hand, suppose somebody claims: 'The computation is logically irreversible'. He or she is clearly referring to a general abstract notion of the computation and an abstract property of it, namely that knowledge of the computation and its output is not sufficient for knowledge of its input. [Ladyman, 2009, 376]

Here, the point is that 'computation' might refer to a concrete, physical process or to an abstract, logical-mathematical process. (Is 'process' even the correct term here? Perhaps something less "dynamic", such as 'structure', might be more appropriate. [Ladyman, 2009, 377] suggests the term 'transformation', which has a satisfactory feel of abstract dynamics to it.)

So, whether our computer is performing or implementing a chess game may depend on whether "the computation" that it is performing is a physical one (in which case, perhaps it is dependent on human intentions) or a theoretical one (in which case, perhaps it is independent, possibly because it is uninterpreted, hence non-intentional).

But those 'perhaps's in the previous paragraph need to be read with caution. Realism, according to [Ladyman, 2009, 377], is the view that what a physical computation is ought to be independent of human intentions:

> If whether or not the human nervous system implements particular computations is not a natural fact about the world that is independent of whether we represent it as doing so, then the computational theory of mind fails to naturalise the mind. Similarly, claims that the universe is a computer, that computation occurs in bacteria and cells, and so on must be understood merely as claims about how we think about these systems if realism about computation is false.

Searle's and Putnam's arguments are arguments *against* realism in computation:

> Searle... argues against realism about computation by reductio: if it were sufficient for some physical system to implement a particular computation that within it there be a pattern of activity with a particular structure, then since most macroscopic physical systems are hugely complex, there will be some pattern of activity within almost every macroscopic system of almost any particular structure, and so almost every macroscopic physical system would implement any given computation. Hence, it cannot be purely in virtue of the patterns of physical activity that systems implement computations, and since these are the only features of physical systems about which we should be realists, realism about computation fails. Putnam... argues that every ordinary open system realises every finite automaton because it is always possible to take a sequence of the successive temporal states of such a system as representing the successive states of the automaton. *Clearly, the basic idea of both arguments is that provided a physical system has sufficient complexity it can be said to implement any computation and therefore there is no fact of the matter about systems implementing one computation in particular.* [Ladyman, 2009, 378, my italics]

So it's the focus on the internal computation, so to speak, that provides the slippage allowing every physical system to implement every theoretical computation.

The analysis in [Ladyman, 2009, 378] supports this idea that it is the interpretation of the inputs and outputs that seems to matter:

> A   common   response   to   these   arguments   (for   example, [Chalmers, 1996] is to claim that for a computation to be

implemented depends not only on what a system does but also on what it would do were its initial state different in an appropriate way. For example, a handheld calculator can only be said to calculate the sum of 7 and 5 when the appropriate buttons are pressed and the screen outputs 12, because had different buttons been pressed it would have given a different output. Similarly, a logic gate that gives the correct output '1' when the inputs represent '1' and '1', is only computing AND if, had the inputs represented '1' and '0' it would have given the output '0'. In general, functions are maps from many different values in the domain of the function to values in its range, but in general implementations are only of a particular instance of the function, in others words, they are processes that evolve from a state that represents one value in the domain to the state that represents the corresponding value in the range.

Of course, the interpretation of the inputs and outputs *is* dependent on human intentions, as Ladyman later observes. But he also observes that even this is debateable, because there are theories of "teleosemantics" ("purposeful" semantics) that allow certain semantic interpretations to be independent of intentions. This, however, is beyond our scope. We explore the notion of implementation in Ch. 14.

39. On the "great insights", see also Ch. 14, Notes for Next Draft No. 8.

40. On the BJ insight (§7.6.3, No. 2), for more on the power of abstraction, see the first few paragraphs of [Antoy and Hanus, 2010].

41. [Rescorla, 2012a] argues "that computation is not sensitive to meaning or semantic properties" (§1, p. 703). More precisely, he argues that *if* a computational process were to be sensitive to semantic properties, then it would have to violate either a condition that he calls 'Syntactic Rules' or a condition that he calls 'Freedom', *and* that such a semantically sensitive computation would have to have an 'indigenous' semantics, not an 'inherited' semantics. He defines these terms as follows:

> SYNTACTIC RULES: Computation is manipulation of syntactic entities according to mechanical rules. We can specify those rules in syntactic terms, without mentioning semantic properties such as meaning, truth, or reference. (§3, p. 707).

> FREEDOM: We can offer a complete syntactic description of the system's states and the mechanical rules governing transitions between states (including any interaction with sensory and motor transducers), while leaving semantic interpretation unconstrained. More precisely, it is metaphysically possible for the system to satisfy our syntactic description while instantiating arbitrarily different semantic properties. (§3, p. 708).

> *Inherited meanings* arise when the system's semantic properties are
> assigned to it by other systems, through either explicit stipulation
> or tacit convention. Nothing about the system itself helps generate
> its own semantics. For instance, words in a book have inherited
> meanings. *Indigenous meanings* arise when a system helps generate
> its own semantics. Indigenous meanings do not arise merely from
> external assignment.   They arise partly from the system's own
> activity, perhaps with ample help from other factors, such as causal,
> evolutionary, or design history. Virtually all commentators agree that
> the mind has indigenous meanings. (§3, pp. 707–708)

(He notes that inherited meanings are similar to what others call "derived"
meanings or "derived" intentionality, and that indigenous meanings are similar to
what others call "original" meanings or intentionality. 'Extrinsic' and 'intrinsic'
might be other terms that are relevantly similar.)

42. On Peacocke and on the Fodorian chess-war example:
    [Rescorla, 2012a, §2.2, p. 707] agrees with Peacocke's analysis, but points out
    that "One can consistently hold the following combination of views: a physical
    system's semantic properties inform its computational nature, so they help type-
    identify its computational states; yet the transitions between computational states
    are not sensitive to semantic properties." Whether a computer is playing chess
    or simulating a Civil War battle depends on a (derived, extrinsic, or inherited)
    semantic interpretation assigned to it by an external agent. But the fact that
    a *single* computer can be doing *both* (playing chess and simulating a battle),
    simultaneously, depends on the fact that the computations are purely syntactic
    and that the two different semantic interpretations are interpretations of that *self-
    same* syntax): *What* the computer is doing is the realm of semantics; *how* it is
    doing it is in the realm of syntax.

43.   The halting theorem is often attributed to Turing in his
        1936 paper.   In fact, Turing did not discuss the halting
        problem, which was introduced by Martin Davis in about 1952.
        [Copeland and Proudfoot, 2010, 248, col. 2]

    The authors don't elaborate on this. Perhaps it has something to do with the
    difference between halting and circle-free?

44. This point might be equally appropriate in Ch. 19. The unsolvability of the
    Entscheidungsproblem due to the non-computability of the Halting Problem
    can be used as a response to the "Lady Lovelace objection" to the possibility
    of AI, that is, the objection that computers can only do what they have
    been programmed to do: According to [Abramson, 2014, his italics, my
    interpolation], Turing's 1936 paper shows

    > that the concepts of determinism and predictability fall apart.
    > Computers, which can be understood as the finite unfoldings of a
    > particular Turing machine, are completely deterministic. But there is

no definite procedure for figuring out, in every case, what they'll do [because of the Halting Problem]: if you could, then you would have a definite procedure for deciding whether any statement of arithmetic is true or not. But there is no such procedure for the one, so there is no such procedure for the other. *Computers are, in the general case, unpredictable, even by someone who knows exactly how they work.*

45. Were the ancient Babylonians really creating algorithms? Insofar as what they were doing fits our informal notion of algorithm, the answer looks to be: yes. But [Mahoney and Haigh (ed.), 2011, 39] cautions against applying 20th-century insights to ancient times:

> When scientists study history, they often use their modern tools to determine what past work was "really about"; for example, the Babyonian mathematicians were "really" writing algorithms. But that's precisely what was not "really" happening. What was really happening was what was possible, indeed imaginable, in the intellectual environment of the time; what was really happening was what the linguistic and conceptual framework then would allow. The framework of Babylonian mathematics had no place for a metamathematical notion such as algorithm.

He even cites the computer scientist Richard Hamming as making the same point in an essay on the history of computing, that "we would [like to] know they thought when they did it": What were Babylonian mathematicians thinking when they created what *we now* call "algorithms"?

But is that fair? Yes, it would be nice to know what they were really thinking, but isn't it also the case that, whatever they thought they were doing, we can describe it in terms of algorithms? (This is the difference between a *de dicto* and a *de re* approach to history!)

7.11   Further Sources of Information

A Books and Essays:

1 **Blass, Andreas; & Gurevich, Yuri (2003), "Algorithms: A Quest for Absolute Definitions",** *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, **No. 81 (October): 195–225.**

   - Pre-print accessed 22 April 2014 from:
     http://research.microsoft.com/en-us/um/people/gurevich/opera/164.pdf

2 Carey, Kevin (2010), "Decoding the Value of Computer Science", *Chronicle of Higher Education* (7 November);
   http://chronicle.com/article/Decoding-the-Value-of-Computer/125266/
   (accessed 10 January 2014).

   - An essay arguing for the value of algorithmic thinking in fields other than computer science (including finance and journalism.

3 Cerf, Vinton G. (2014), "Virtual Reality Redux", *Communications of the ACM* 57(1) (January): 7; accessed 21 January 2014 from
   http://cacm.acm.org/magazines/2014/1/170860-virtual-reality-redux/fulltext

   - Has some interesting comments that are relevant to the insight about binary representation of information.

4 **Chaitin, Gregory (2006), "How Real Are Real Numbers?",** *International Journal of Bifurcation and Chaos*; online at:
   2006 version: http://www.cs.auckland.ac.nz/∼chaitin/olympia.pdf
   2009 version: http://www.umcs.maine.edu/∼chaitin/wlu.html
   (both accessed 6 November 2012).

   - On the halting problem, the non-existence of real numbers(!), and the idea that "everything is software, God is a computer programmer,. . . and the world is. . . a giant computer"(!!).
   - On the non-"reality" of "real" numbers, see also [Knuth, 2001, 174–175].

5 Chaitin, Gregory (2006), "The Limits of Reason", *Scientific American* (March): 74–81; online at: http://www.umcs.maine.edu/∼chaitin/sciamer3.pdf
   (accessed 6 November 2012).

   - "Ideas on complexity and randomness originally suggested by Gottfried W. Leibniz in 1686, combined with modern information theory [in particular, implications of the Halting Problem], imply that there can never be a 'theory of everything' for all of mathematics".

6 Chisum, Donald S. (1985-1986), "The Patentability of Algorithms", *University of Pittsburgh Law Review* 47: 959–1022.

7 Cook, Byron; Podelski, Andreas; & Rybalchenko, Andrey (2011), "Proving Program Termination", *Communications of the ACM* 54(5) (May): 88–98.

   - "In contrast to popular belief, proving termination is not always impossible.. . . [M]any have drawn too stong of a conclusion [from Turing's proof that Halting is non-computable]. . . and falsely believe we are *always unable to* prove termination, rather than more benign consequence that we are *unable to always* prove termination.. . . In our new problem statement we will still require that a termination proving tool always return answers that are correct, but we will not necessarily require an answer" (my italics).

8 Copeland, B. Jack (1996), "What Is Computation?", *Synthese* 108: 335–359.

9 Copeland, B. Jack (1997), "The Broad Conception of Computation", *American Behavioral Scientist* 40(6) (May): 690–716.

- An essay on hypercomputation (or "non-classical" computation), but the introductory section (pp. 690–698) contains an enlightening discussion of the scope and limitations of Turing's accomplishments.

10 Copeland, B. Jack (2004), "Computation", in Luciano Floridi (ed.), *The Blackwell Guide to the Philosophy of Computing and Information* (Malden, MA: Blackwell).

11 Corballis, Michael C. (2007), "The Uniqueness of Human Recursive Thinking", *American Scientist* 95 (May-June): 240–248.

12 Crossley, John N.; & Henry, Alan S. (1990), "Thus Spake Al-Khwarizmi: A Translation of the Text of Cambridge University Library ms. Ii.vi.5", *Historia Mathematica* 17(2): 103–131.

13 Farkas, David K. (1999), "The Logical and Rhetorical Construction of Procedural Discourse", *Technical Communication* 46(1) (February): 42–54; online at:
http://www.hcde.washington.edu/sites/default/files/people/docs/proceduraldiscourse.pdf
(accessed 6 November 2012).

14 Feldman, Yishai A. (1993), "Finite-State Machines", in Allen Kent & James G. Williams (eds.), *Encyclopedia of Computer Science and Technology* (New York: Marcel Dekker) 28(13): 387–418.

15 Floridi, Luciano (1999), "The Digital Workshop", Ch. 2 of *Philosophy and Computing: An Introduction* (London: Routledge).

16 Folina, Janet (1998), "Church's Thesis: Prelude to a Proof", *Philosophia Mathematica* 6: 302–323.

17 Gurevich, Yuri (2011), "What Is an Algorithm?", *Technical Report MSR-TR-2011-116* (Redmond, WA: Microsoft Research); online at:
http://research.microsoft.com/pubs/155608/209-3.pdf (accessed 6 November 2012).

18 Harnad, Stevan (ed.) (1994), Special Issue on "What Is Computation?", *Minds and Machines* 4(4), containing:

- Harnad, Stevan, "Preface", pp. 377–378.
- **Harnad, Stevan, "Computation Is Just Interpretable Symbol Manipulation; Cognition Isn't", pp. 379–390**; online at:
  http://users.ecs.soton.ac.uk/harnad/Papers/Harnad/harnad94.computation.cognition.html
  (accessed 6 November 2012).
- **Chalmers, David J., "On Implementing a Computation", pp. 391–402.**
- Chrisley, Ronald L., "Why Everything Doesn't Realize Every Computation", pp. 403–420.
- MacLennan, Bruce J., "Words Lie in Our Way", pp. 421–437.
- Kentridge, Robert W., "Symbols, Neurons, Soap-Bubbles, and the Neural Computation Underlying Cognition", pp. 439–449.
- Boyle, C. Franklin, "Computation as an Intrinsic Property", pp. 451–467.
- **Bringsjord, Selmer, "Computation, among Other Things, Is beneath Us", pp. 489–490**; portions online at:
  http://homepages.rpi.edu/∼brings/SELPAP/beneath.ab.html
  (accessed 6 November 2012).

19 Hershenov, David B. (2002), "Scattered Artifacts", *Southern Journal of Philosophy* 60: 211–216.

- Accessed 1 August 2014 from:
  http://www.acsu.buffalo.edu/∼dh25/articles/scattered.pdf
- Is a "watch undergoing repair, its parts spread actross a craftsman's table" still a watch? This essay raises some interesting considerations about where in a computer the computation happens (that is, where is the input converted to output?).
- For more on the topic of scattered objects, see:
  http://theotodman.com/Notes/Notes_1/Notes_123.htm
  (accessed 1 August 2014).

20 Hill, Robin (2013), "What an Algorithm Is, and Is Not", *Communications of the ACM*, 56(6) (June): 8–9;
http://mags.acm.org/communications/june_2013/?pg=11#pg11
(accessed 23 December 2013).

21 Horgan, John (1990), "Profile: Claude E. Shannon; Unicyclist, Juggler and Father of Information Theory", *Scientific American* (January): 22, 22A–22B.

22 Knuth, Donald (1973), "Basic Concepts: §1.1: Algorithms", *The Art of Computer Programming, Second Edition* (Reading, MA: Addison-Wesley): xiv–9.

23 Knuth, Donald (1974), "Computer Science and Its Relation to Mathematics", *American Mathematical Monthly* 81(4) (April): 323–343.

24 Korfhage, Robert R. (1993), "Algorithm", in Anthony Ralston & Edwin D. Reilly (eds.), *Encyclopedia of Computer Science, 3rd Edition* (New York: Van Nostrand Reinhold): 27–29.

25 Kreisel, Georg (1987), "Church's Thesis and the Ideal of Informal Rigour", *Notre Dame Journal of Formal Logic* 28(4): 499–519.

- A paper by a well-known logician arguing that Church's Thesis can be proved.

26 Leiber, Justin (2006), "Turing's Golden: How Well Turing's Work Stands Today", *Philosophical Psychology* 19(1): 13–46.

27 Manzano, María (1997), "Alonzo Church: His Life, His Work and Some of His Miracles", *History and Philosophy of Logic* 18: 211–232.

28 Mendelson, Elliott (1990), "Second Thoughts about Church's Thesis and Mathematical Proofs", *Journal of Philosophy* 87(5) (May): 225–233.

- For a response, see:
  Bringsjord, Selmer (1993), "The Narrational Case against Church's Thesis", accessed 17 April 2014 from:
  http://homepages.rpi.edu/∼brings/SELPAP/CT/ct/

29 Moschovakis, Yiannis N. (1998), "On Founding the Theory of Algorithms", in H.G. Dales & G. Oliveri (eds.), *Truth in Mathematics* (Oxford: Clarendon Press): 71–104; online at:
http://www.math.ucla.edu/ ynm/papers/foundalg.ps
(accessed 6 November 2012).

30 Moschovakis, Yiannis N. (2001), "What Is an Algorithm?", in B. Engquist & W. Schmid (eds.), *Mathematics Unlimited: 2001 and Beyond* (Berlin: Springer): 918–936.

31 Piccinini, Gualtiero (2007), "Computationalism, the Church-Turing Thesis, and the Church-Turing Fallacy", *Synthese* 154: 97–120.

32 Rescorla, Michael (2007), "Church's Thesis and the Conceptual Analysis of Computability", *Notre Dame Journal of Formal Logic* 48(2): 253–280.

- Rescorla argues that Church's Thesis (that a number-theoretic function is intuitively computable iff it is recursive) is not the same as Turing's Thesis (that a number-theoretic function is intuitively computable iff a corresponding string-theoretic function [that represents the number-theoretic one] is computable by a Turing machine).

- See also:
  Shapiro, Stuart C. (1977), "Representing Numbers in Semantic Networks: Prolegomena", in *Proceedings of the 5th International Joint Conference on Artificial Intelligence* (Los Altos, CA: Morgan Kaufmann): 284.

33 Schulman, Ari N. (2009), "Why Minds Are Not Like Computers", *The New Atlantis* (Winter): 46-68.

34 Shagrir, Oron (1999), "What Is Computer Science About?", *The Monist* 82(1): 131–149.

- Despite its title, this paper is more about what computers are and what computation is.

35 Shapiro, Stewart (1983), "Remarks on the Development of Computability", *History and Philosophy of Logic* 4: 203–220.

- Discusses the notion of computability from a historical perspective; contains a discussion of Church's thesis.

36 Shapiro, Stewart (1993), "Understanding Church's Thesis, Again", *Acta Analytica* 11: S.59–77.

- "tries to show that Church's thesis can be proved mathematically..."

37 Sieg, Wilfried (1994), "Mechanical Procedures and Mathematical Experience", in George Alexander (ed.), *Mathematics and Mind* (New York: Oxford University Press): 71–117.

- Accessed 10 December 2013 from:
  http://repository.cmu.edu/cgi/viewcontent.cgi?article=1248&context=philosophy
- Contains a detailed history and analysis of the development of the formal notions of algorithm in the 1930s and after.

38 Sieg, Wilfried (1997), "Step by Recursive Step: Church's Analysis of Effective Calculability", *Bulletin of Symbolic Logic* 3(2) (June): 154–180.

39 Sieg, Wilfried (2008), "Church without Dogma: Axioms for Computability", in S.B. Cooper, Benedikt Löwe, & Andrea Sorbi (eds), *New Computational Paradigms: Changing Conceptions of What Is Computable* (Springer): 139–152; preprint (2004) online at:
http://www.hss.cmu.edu/philosophy/sieg/Church%20without%20dogma.pdf
(accessed 6 November 2012).

40 Sieg, Wilfried (2006), "Gödel on Computability", *Philosophia Mathematica* 14 (June): 189–207; online at:
http://www.hss.cmu.edu/philosophy/sieg/A%20G_del%20on%20Computability.pdf
(accessed 6 November 2012).

41 **Smith, Brian Cantwell (2002), "The Foundations of Computing"**, in Matthias Scheutz (ed.), *Computationalism: New Directions* (Cambridge, MA: MIT Press): 23–58.

42 **Soare, Robert I. (2009), "Turing Oracle Machines, Online Computing, and Three Displacements in Computability Theory",** *Annals of Pure and Applied Logic* **160: 368–399.**

- Preprint accessed 2 July 2014 from:
  http://www.people.cs.uchicago.edu/ soare/History/turing.pdf
  and published version accessed 2 July 2014 from:
  http://ac.els-cdn.com/S0168007209000128/1-s2.0-S0168007209000128-
  main.pdf?_tid=8258a7e2-01ef-11e4-9636-
  00000aab0f6b&acdnat=1404309072_f745d1632bb6fdd95f711397fda63ee2

- §2, "Origins of Computabilty and Incomputablity", contains a good summary of the history of both Turing's and Gödel's accomplishments.

43 Vardi, Moshe Y. (2011), "Solving the Unsolvable", *Communications of the ACM* 54(7) (July): 5; online at:
http://www.inf.unibz.it/∼calvanese/teaching/tc/material/solving-the-unsolvable-CACM-2011-07.pdf
(accessed 12 November 2013).

- See also:
  Ledgard, Henry; & Vardi, Moshe Y. (2011), "Solved, for All Practical Purposes" (letter to the editor, and reply), *Communications of the ACM* 54(9) (September): 7; online at: http://tinyurl.com/LedgardVardi2011
  (accessed 12 November 2013).

44 Wangsness, T.; & Franklin, J. (1966), " 'Algorithm' and 'Formula' ", *Communications of the ACM* 9(4) (April): 243.

- The published answers:
    i. Huber, Harmut G.M. (1966), "Algorithm and Formula", *Communications of the ACM* 9(9) (September): 653–654.
    ii. Knuth, Donald (1966), "Algorithm and Program: Information and Data", *Communications of the ACM* 9(9) (September): 654.

45 On heuristics vs. algorithms:

    i. Findler, Nicholas V. (1993), "Heuristic", in Anthony Ralston & Edwin D. Reilly (eds.), *Encyclopedia of Computer Science, 3rd Edition* (New York: Van Nostrand Reinhold): 611–612.
    ii. Korf, Richard E. (1992), "Heuristics", in Stuart C. Shapiro (ed.), *Encyclopedia of Artificial Intelligence, 2nd Edition* (New York: John Wiley & Sons): 611–615.
    iii. **Newell, Allen, & Simon, Herbert A. (1976), "Computer Science as Empirical Inquiry: Symbols and Search"**, *Communications of the ACM* 19(3) (March): 113–126.
    iv. Oommen, B. John; & Rueda, Luis G. (2005), "A Formal Analysis of Why Heuristic Functions Work", *Artificial Intelligence* 164(1-2) (May): 1–22.
    v. Rapaport, William J. (1998), "How Minds Can Be Computational Systems", *Journal of Experimental and Theoretical Artificial Intelligence* 10: 403–419.
    vi. **Romanycia, Marc H.J., & Pelletier, Francis Jeffry (1985), "What Is a Heuristic?"**, *Computational Intelligence* 1: 47–58.
    vii. Shapiro, Stuart C. (1992), "Artificial Intelligence", in Stuart C. Shapiro (ed.), *Encyclopedia of Artificial Intelligence, 2nd Edition* (New York: John Wiley & Sons): 54–57; revised version appears in Anthony Ralston & Edwin D. Reilly (eds.),

Encyclopedia of Computer Science, 3rd Edition (New York: Van Nostrand Reinhold, 1993): 87–90.

viii. Simon, Herbert; & Newell, Allen (1958), "Heuristic Problem Solving: The Next Advance in Operations Research", *Operations Research* 6(1) (January-February): 1–10.

- In addition to its discussion of the division of labor (see §6.4.3, above) and its infamous prediction that a computer would become the world chess champion by 1968, this classic paper also distinguishes algorithmic problem solving of "well-structured" problems from heuristic problem solving of "ill-structured" problems.

ix. Thagard, Paul (2007), "Coherence, Truth, and the Development of Scientific Knowledge", *Philosophy of Science* 74 (January): 28–47.

- Not about heuristics, but presents a theory of "approximate" truth that bears a close resemblance to the idea that a heuristic is an algorithm that computes an approximately correct answer.

B Websites:

1 Chazelle, Bernard (2006), "The Algorithm: Idiom of Modern Science"
http://www.cs.princeton.edu/∼chazelle/pubs/algorithm.html
(accessed 6 November 2012).

2 Etymology of 'Compute'
http://www.cse.buffalo.edu/∼rapaport/584/computetymology.html
(accessed 6 November 2012).

3 O'Connor, John J.; & Robertson, Edmund F. (1999), "Abu Ja'far Muhammad ibn Musa Al-Khwarizmi"
http://www-groups.dcs.st-and.ac.uk/∼history/Mathematicians/Al-Khwarizmi.html
(accessed 6 November 2012).

4 PolR (2009), "An Explanation of Computation Theory for Lawyers", *Groklaw* (11 November),
http://www.groklaw.net/article.php?story=20091111151305785,
http://www.groklaw.net/staticpages/index.php?page=20091110152507492,
and
http://www.groklaw.net/pdf2/ComputationalTheoryforLawyers.pdf

(accessed 16 December 2013).

C Rajagopalan, Bala (ca. 2011), "Halting Problem Is Solvable" (joke),
http://www.netfunny.com/rhf/jokes/89q4/halting.760.html
(accessed 10 January 2014).

Figure 7.2: source: https://xkcd.com/844/

Figure 7.3: source: http://abstrusegoose.com/432
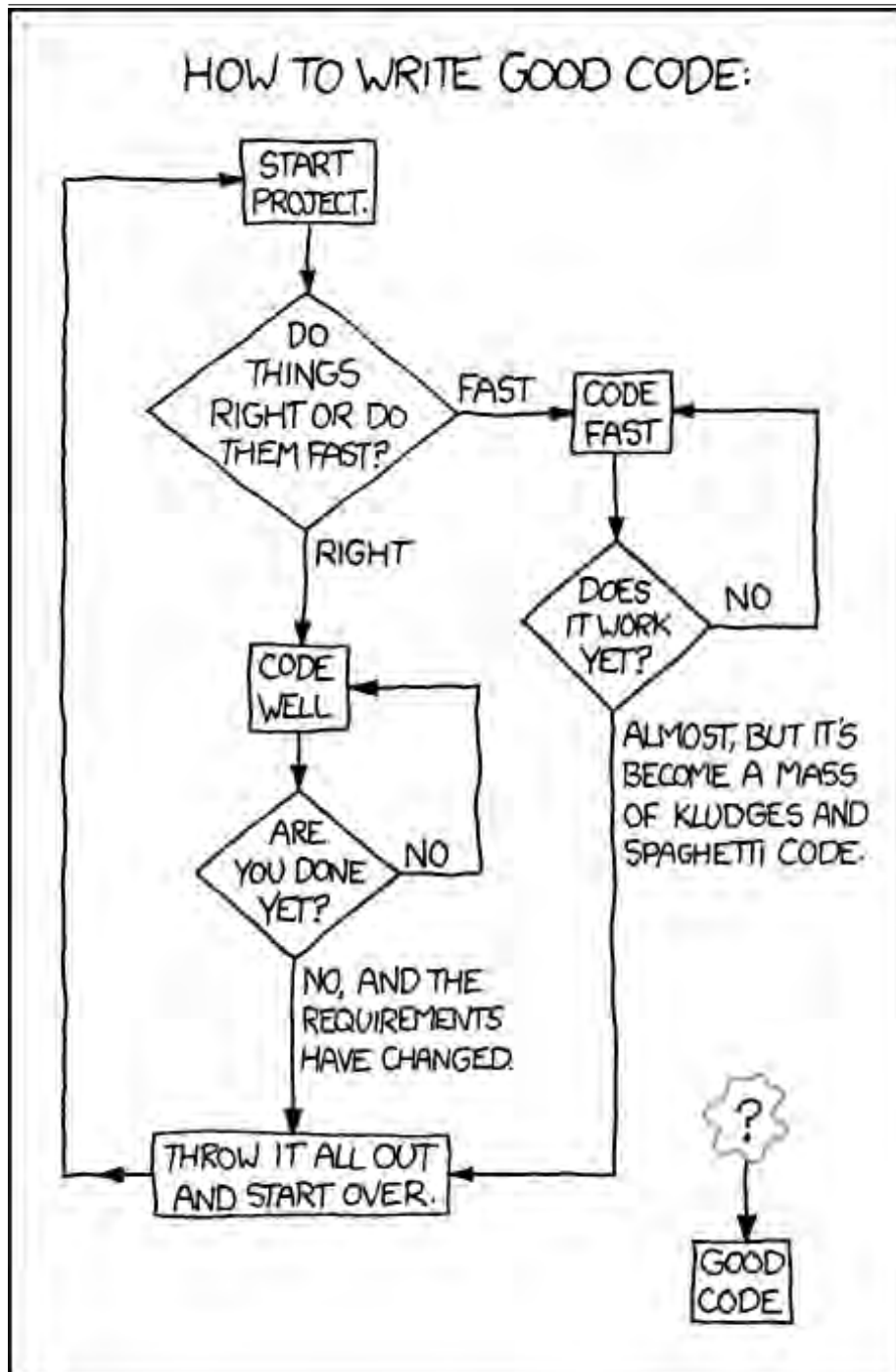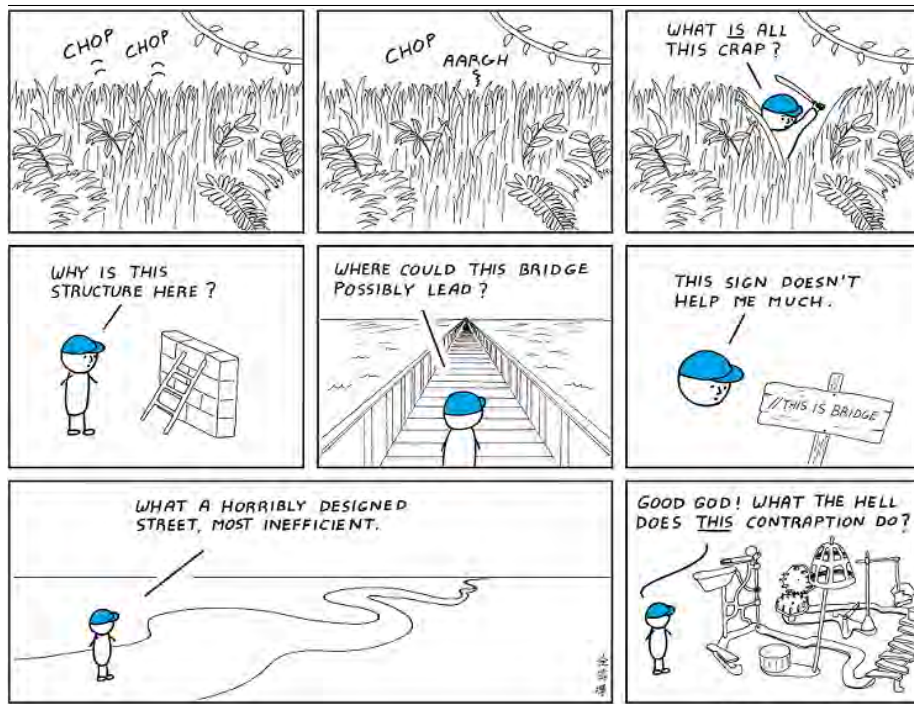
Chapter 8

Turing's Analysis of Computation

Turing's 'Machines'. These machines are *humans* who calculate. [Wittgenstein, 1980, 191e, §1096][1]

[A] human calculator, provided with pencil and paper and explicit instructions, can be regarded as a kind of Turing machine. [Church, 1937]

You can build an organ which can do anything that can be done, but you cannot build an organ which tells you whether it can be done. [von Neumann, 1966], cited in [Dyson, 2012a].

Figure 8.1: ©2009 by ??

---

[1] "The quotation, though insightful, is somewhat confusingly put. Better would have been: these machines are Turing's mechanical model of *humans* who calculate." [Kripke, 2013, 96fn12]
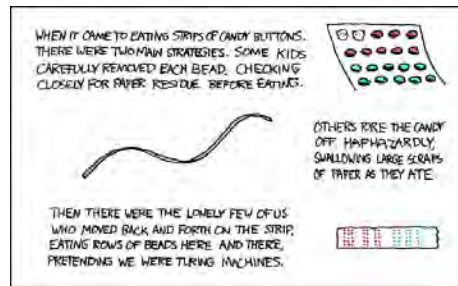
Figure 8.2: ©YEAR by ??

8.1   Required Reading:

- **Turing, Alan M. (1936), "On Computable Numbers, with an Application to the *Entscheidungsproblem*", *Proceedings of the London Mathematical Society*, Ser. 2, Vol. 42 (1937): 230–265**.

    - Reprinted, with corrections, in Martin Davis (ed.), *The Undecidable: Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions* (New York: Raven Press, 1965): 116–154.

    - Online at:
      http://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf (accessed 6 November 2012).

  1. Concentrate on the informal expository parts; the technical parts are, of course, of interest, but are rather difficult to follow, incorrect in many parts, and can be skimmed.

  2. In particular, concentrate on:
      - §§1–6
        (study the simple examples of Turing machines carefully;
        skim the complex ones)
      - and §9, part I
        (which elaborates on what it is that a *human* computer does).

  3. §7 describes the universal Turing machine;
     §8 describes the Halting Problem.

      - You can skim these sections (that's 'ski**m**', not 'ski**p**'!)

8.2 Introduction

If there is a single document that could be called the foundational document of computer science, it would be Alan Mathison Turing's 1936 article, "On Computable Numbers, with an Application to the *Entscheidungsproblem*", which appeared in the journal *Proceedings of the London Mathematical Society, Series 2*. In this paper, Turing (who was only about 24 years old at the time) accomplished 4 major goals:

1. He gave what is considered to be the clearest and most convincing mathematical analysis of computation (what is now called, in his honor, a "Turing machine").

2. He described a "universal" Turing machine, which is a mathematical version of a programmable computer.

3. He proved that there were some functions that were *not* computable, thus showing that computation was not a trivial property. After all, if all functions were computable—which no doubt would be a very nice feature–then computability would not really be a very interesting or special property. But, because some functions are *not* computable, computability is a property that only some (but not all) functions have, and so it becomes more interesting.

4. And (I will suggest) he wrote the first AI program (see §8.14, below).

Thus, arguably, in this paper, he created the modern discipline of computer science.

Because this paper was so important and so influential, it is well worth reading. Fortunately, although parts of it are tough going (and it contains some errors),[2] much of it is very clearly written. It is not so much that the "tough" parts are difficult or hard to understand, but they are full of nitty, gritty details that have to be slogged through. Fortunately, Turing has a subtle sense of humor, too.

In this chapter, I will provide a guide to reading *parts* of Turing's paper slowly and carefully, by actively thinking about it. (A wonderful guide to reading *all* of it slowly and carefully is [Petzold, 2008].)[3]

8.3 The *Entscheidungsproblem*

First, let's look at the title, in particular its last word: '*Entscheidungsproblem*'. This is a German noun that was well known to mathematicians in the 1930s; '*Entscheidung*' means "decision", '*-s*' represents the possessive,[4] and '*problem* means "problem". So, an *Entscheidungsproblem* is a decision problem, and *the* Decision Problem was the problem of finding an algorithm that would take two things as input: (1) a formal logic L and (2) a proposition φ_L in the language for that logic, and that would output either 'yes', if φ_L was a theorem of that logic, or else 'no', if $\neg\varphi_L$ was a theorem of that logic

---

[2]Some of which Turing himself corrected [Turing, 1938]. For a more complete "debugging", see [Davies, 1999].

[3]On the value of slow and active reading in general, see my website [Rapaport, 2014], and [Fletcher, 2007, Blessing, 2013].

[4]Just as in English, so '*Entscheidungs*' means "decision's".

(that is, if φ_L was not a theorem of *L*). In other words, the Decision Problem was the problem of finding a general algorithm for deciding whether any given proposition was a theorem.

Wouldn't that be nice? Mathematics could be completely automated: Given any mathematical proposition, one could apply this general algorithm to it, and you would be able to know if it were a theorem or not. Turing was fascinated by this problem, and he solved it. Along the way, he invented computer science! He solved the Decision Problem in the negative, by showing that no such algorithm existed—we've already seen how: He showed that there was at least one problem (the Halting Problem) for which there was no such algorithm.[5]

8.4   Paragraph 1

One of the best ways to read is to read slowly and actively. This is especially true when you are reading a technical paper, and even more especially when you are reading mathematics. Reading slowly and actively menas reading each sentence slowly, thinking about it actively, and making sure that you understand it before reading the next sentence. (One way to make sure that you understand it is to ask yourself *why* the author said it, or *why* it might be true.) If you don't understand it (after reading it slowly and actively), then you should re-read all of the previous sentences to make sure that you really understood them. Whenever you come to a sentence that you really don't understand, you should ask someone to help you understand it.

(Of course, it could also be the case that you don't understand it because it isn't true, or doesn't follow from what has been said, or is confused in some way—and not because it's somehow your fault that you don't understand it!)

8.4.1   Sentence 1

So, let's begin our slow and active reading of Turing's paper with the first sentence of the first paragraph:

> The "computable" numbers may be described briefly as the real numbers whose expressions as a decimal are *calculable by finite means.* [Turing, 1936, 230, my italics][6]

The word 'computable' occurs in quotes here, because Turing is using it in an informal, intuitive sense. It is the sense that he will make mathematically precise in the rest of the paper.

Real numbers are all of the numbers on the continuous number line. They consist of:

1. the rational numbers, which consist of:

   (a) the integers, which—in turn—consist of:

---

[5]For more on the decision problem, see §6.5.

[6]In the rest of this chapter, citations from [Turing, 1936] will just be cited by section or page number.

    i. the non-negative natural numbers (0, 1, 2, . . .), and

    ii. the negative natural numbers $(-1, -2, \ldots)$, and

  (b) all other numbers that can be expressed as a ratio of integers (or that can be expressed in decimal notation with repeating decimals), and

2. the irrational numbers (that is, those numbers that cannot be expressed as a ratio of integers, such as $\pi, \sqrt{2}$, etc.)

But the real numbers do not include the "complex" numbers, such as $\sqrt{-1}$.

    Every real number can be expressed "as a decimal", that is, in decimal notation. For instance:

$$1 = 1.0 = 1.00 = 1.000 \text{ (etc.)},$$

$$\tfrac{1}{2} = 0.5 = 0.50 = 0.500 = 0.5000 \text{ (etc.)},$$

$$\tfrac{1}{3} = 0.33333\ldots$$

These are all rational numbers and examples of repeating decimals. But the reals also include the irrational numbers, which have *non*-repeating decimals:[7]

$$\pi = 3.1415926535\ldots,$$

$$\sqrt{2} = 1.41421356237309\ldots$$

    Given a real number, is there an algorithm for computing its decimal representation? If so, then its "decimal [is] calculable by finite means" (because algorithms must be finite, as we saw in §7.5).[8]

8.4.2 Last Sentence

Now, if we were really going to do a slow (and active!) reading, we would next move on to sentence 2. But, in the interests of keeping this chapter shorter than a full book (and so as not to repeat everything in [Petzold, 2008]), we'll skip to the last sentence of the paragraph:

> According to my definition, a number is computable if its decimal can be written down *by a machine*. (p. 230, my italics.)

    This is probably best understood as an alternative way of expressing the first sentence: To be "calculable by finite means" is to be capable of being "written down by a machine". Perhaps this way of putting it extends the notion a bit, because it

---

[7]Decimal notation is also called 'base-10 notation'. It is merely one example (another being binary—or base-2—notation) of what is more generally known as 'radix notation' or 'positional notation'; see http://en.wikipedia.org/wiki/Radix and http://math.comsci.us/radix/radix.html (both accessed 22 November 2013).

[8]For another discussion of the computation of real numbers, see [Hartmanis and Stearns, 1967]. For a commentary on the "reality" of "real" numbers, see [Knuth, 2001, 174–175].

suggests that if a number is calculable by finite means, then that calculation can be done automatically, that is, by a machine—without human intervention. And that, after all, was the goal of all of those who tried to build calculators or computing machines, as we saw in Ch. 6. So, Turing's goal in this paper is to give a mathematical analysis of what can be accomplished by any such machine (and then to apply the results of this analysis to solving the Decision Problem).

8.5   Paragraph 2

8.5.1   Sentence 1

> In §§9, 10 I give some arguments with the intention of showing that the computable numbers include all numbers which could naturally be regarded as computable. (p. 230.)

We will look at some of those arguments later, but, right now, let's focus on the phrase 'naturally be regarded as computable". This refers to the same informal, intuitive, pre-theoretical notion of computation that his quoted use of 'computable' referred to in the first sentence. It is the sense in which Hilbert wondered about which mathematical problems were decidable, the sense in which people used the phrase "effective computation", the sense in which people used the word 'algorithm', and so on. It is the sense in which *people* (mathematicians, in particular) can compute. And one of its crucial features is that it be finite.

8.5.2   Last Sentence

Once again, we'll skip to the last sentence of the paragraph:

> The *computable* numbers do not, however, include all *definable* numbers, and an example is given of a definable number which is not computable. (p. 230, my italics)

As we noted before (§8.2), it is much more interesting if not all functions—or numbers—are computable. Any property that *everything* has is not especially interesting. But if there is a property that *only some* things have (and others lack), then we can begin to categorize those things and thus learn something more about them.

So Turing is promising to show us that computability is an interesting (because not a universal) property. And he's not going to do that by giving us some abstract (or "transcendental") argument; rather, he's actually going to show us a non-computable number (and, presumably, show us *why* it's not computable). We've already seen what this is: It's the (Gödel) number for an algorithm (a Turing machine) for the Halting Problem. So, in this chapter, we'll skip that part of Turing's paper. We'll also skip the rest of the introductory section of the paper, which simply lays out what Turing will cover in the rest of the paper.

8.6   Section 1, Paragraph 1

Let's move on to Turing's Section 1, "Computing Machines". We'll look at the first paragraph and then digress to Turing's Section 9 before returning to this section.

Here is the first paragraph of Section 1:

> We have said that the computable numbers are those whose decimals are calculable by finite means. This requires rather more explicit definition. No real attempt will be made to justify the definitions given until we reach §9. (p. 231.)

This is why we will digress to that section in a moment. But first let's continue with the present paragraph:

> For the present I shall only say that the justification [of the definitions] lies in the fact that the *human memory is necessarily limited*. (p. 231, my interpolation and italics.)

Turing's point—following Hilbert—is that we humans do not have infinite means at our disposal. We all eventually die, and we cannot work infinitely fast, so the number of calculations we can make in a single lifetime is finite.

But how big is "finite"? Let's suppose, for the sake of argument, that a typical human (named 'Pat') lives as long as 100 years. And let's suppose that from the time Pat is born until the time Pat dies, Pat does nothing but compute. Obviously, this is highly unrealistic, but I want to estimate the *maximum* number of computations that a typical human could perform. The *actual* number will, of course, be far fewer. How long does a computation performed by a human take? Let's suppose that the simplest possible computation (following our notion of a "basic function" in §7.7.2) is computing the successor of a natural number, and let's suppose that it takes as long as 1 second. In Pat's lifetime, approximately 3,153,600,000 successors can be computed (because that's approximately the number of seconds in 100 years). Are there any problems that would require more than that number of computations? Yes! It has been estimated that the number of possible moves in a chess game is 10^{125}, which is about 10^{116} times as large as the largest number of computations that a human could possibly perform. In other words, we humans are not only finite, we are *very* finite!

But computer scientists and mathematicians tend to ignore such human limitations and pay attention only to the mathematical notion of finiteness. Even the mathematical notion, which is quite a bit larger than the actual human notion (for more on this, see [Knuth, 2001]), is still smaller than infinity, and so the computable numbers, as Turing defines them, include quite a bit.

8.7   Section 9

8.7.1   Section 9, Paragraph 1

I want to skip now to Turing's §9, "The Extent of the Computable Numbers", because it is this section that contains the most fascinating part of Turing's analysis. We'll return to his §1 later. He begins as follows:

> No attempt has yet been made [in Turing's article] to show that the
> "computable" numbers include all numbers which would *naturally* be
> regarded as computable. (p. 249, my interpolation and italics.)

Here, Turing is comparing two notions of computability: the technical notion (signified
by the first occurrence of the word 'computable'—in "scare quotes") and the informal
or "natural" notion. He is going to argue that the first includes the second. Presumably,
it is more obvious that the second (the "natural" notion) includes the first (the technical
notion), that is, that if a number is technically computable, then it is "naturally"
computable. The less obvious inclusion is the one that is more in need of support,
that if a number is "naturally" computable, then it is technically computable. But what
kind of argument would help convince us of this? Turing says:

> All arguments which can be given are bound to be, fundamentally, appeals
> to intuition, and for this reason rather unsatisfactory mathematically.
> (p. 249.)

Why is this so? Because one of the two notions—the "natural" one—is informal, and
so no formal, logical argument can be based on it. This is why Turing's thesis (that is,
Church's thesis, the Church-Turing thesis, or the "computability thesis") is a *thesis* and
not a *theorem*—it is a hypothesis and not something formally provable. Nonetheless,
Turing will give us "appeals to intuition", that is, *in*formal arguments, in fact, three
kinds:

> The real question at issue is "What are the possible processes which can
> be carried out in computing a number?" (p. 249.)

If Turing can answer this question, even informally, then we may be able to come up
with a formal notion that captures the informal one. That is the first step (and the only
one that we shall consider in this book).

8.7.2   Section 9, Subsection I

The first step is "a direct appeal to intuition", and Turing notes that "this argument
is only an elaboration of the ideas of §1" (p. 249) (which is why we have made this
digression to Turing's §9 from his §1; when we return to his §1, we will see that it
summarizes his §9).

8.7.2.1   Section 9, Subsection I, Paragraph 1

The first part of the answer to the question, "What are the possible processes which
can be carried out in computing a number?"—that is, the first intuition about "natural"
computation—is this:

> Computing is normally done by writing certain symbols on paper. (p. 249.)

So, we need to be able to write symbols on paper. Is this true? What kind of symbols?
And what kind of paper?

8.7.2.1.1 Is it true? Is *computing* normally done by writing symbols on paper? We who live in the 21st century might think that this is obviously false: Computers don't *have to* write symbols on paper in order to their job. They do when we have the computer print a document, but they don't when we are watching a YouTube video. But remember that Turing is analyzing the "natural" notion of computing: the kind of computing that *humans* do. And his model includes arithmetic computations. Those typically *are* done by writing symbols on paper (or, perhaps, by imagining that we are writing symbols on paper, as when we do a computation "in our head").

8.7.2.1.2 What about the paper?

> We may suppose this paper is divided into squares like a child's arithmetic book. (p. 249.)

In other words, we can use graph paper! Presumably, we can put one symbol into each square of the graph paper. So, for example, if we're going to write down the symbols for computing the sum of 43 and 87, we could write it like this:

```
-------
| |1| |
-------
| |4|3|
-------
|+|8|7|
-------
|1|3|0|
-------
```

We write '43' in two squares, then we write '+87' in three squares beneath this, aligning the ones and tens columns. To perform the computation, we compute the sum of 7 and 3, and write it as follows: The ones place of the sum ('0') is written below '7' in the ones column and the tens place of the sum ('1') is "carried" to the square above the tens place of '43'. Then the sum of 1, 4, and 8 is computed and then written as follows: The ones place of that sum, namely, '3' (which is the tens place of the sum of 43 and 87) is written below '8' in the tens column, and the tens place of that sum—namely, '1' (which is the hundreds place of the sum of 43 and 8—namely, '1')—is written in the square to the left of that '3'.

Turing continues:

> In elementary arithmetic the two-dimensional character of the paper is sometimes used. (p. 249.)

As we have just seen.

> But such a use is always avoidable, and I think that it will be agreed that the two-dimensional character of paper is no essential of computation. (p. 249.)

In other words, we could have just as well (if not just as easily) written the computation thus:

```
_____
|1|  |4|3|+|8|7|=|1|3|0|
_____
```

Here, we begin by writing the problem '43+87' in five successive squares, followed, perhaps, by an equals-sign. And we can write the answer in the squares following the equals-sign, writing the carried '1' in an empty square somewhere else, clearly separated (here, by a blank square) from the problem. So, the use of two-dimensional graph paper has been avoided (at the cost of some extra bookkeeping). As a consequence, Turing can say:

> I assume then that **the computation is carried out** on one-dimensional paper, *i.e.* **on a tape divided into squares**. (p. 249, my boldface.)

Here is the famous tape of what will become a Turing machine! (Note, though, that Turing has not yet said anything about the *length* of the tape; at this point, it could be finite.) We now have our paper on which we can write our symbols.

8.7.2.1.3   What about the symbols?

> I shall also suppose that **the number of symbols which may be printed is finite**. (p. 249, my boldface)

This is the first item that Turing has put a limitation on: There are only a finite number of symbols.

(Actually, Turing is a bit ambiguous here: There might be an infinite number of different kinds of symbols, but we're only allowed to *print* a finite number of them. Or there might only be a finite number of different kinds of symbols—with a further vagueness about how many of them we can print: If the tape is finite, then we can only print a finite number of the finite amount of symbols, but, if the tape is infinite, we could print an infinite number of the finite amount of symbols. But it is clear from what he says next that he means that there are only a finite number of different kinds of symbols.)

Why finite? Because:

> If we were to allow an infinity of symbols, then there would be symbols differing to an arbitrarily small extent. (p. 249.)

There are two things to consider here: Why would this be the case? And why does it matter? The answer to both of these questions is easy: If the human who is doing the computation has to be able to identify and distinguish between an infinite number of symbols, surely some of them may get confused, especially if they look a lot alike! Would they have to look alike? A footnote at this point suggests why the answer is 'yes':

> If we regard a symbol as literally printed on a square we may suppose that the square is $0 \leq x \leq 1, 0 \leq y \leq 1$. The symbol is defined as a set of points in this square, viz. the set occupied by printer's ink. (p. 249, footnote.)

That is, we may suppose that the square is 1 unit by 1 unit (say, 1 cm by 1 cm). Any symbol has to be printed in this space. Imagine that each symbol consists of very tiny dots of ink. To be able to print an infinite number of different kinds of symbols in such a square, some of them are going to differ from others by just a single dot of ink, and any two such symbols are going to "differ to an arbitrarily small extent" and, thus, be impossible for a human to distinguish. So, "the number of symbols which may be printed" must be finite in order for the human to be able to easily read them.

Is this really a limitation?

> The effect of this restriction of the number of symbols is not very serious. (p. 249.)

Why not? Because:

> It is always possible to use sequences of symbols in the place of single symbols. Thus an Arabic numeral such as 17 or 999999999999999 is normally treated as a single symbol. Similarly in any European language words are treated as single symbols... (pp. 249–250.)

In other words, the familiar idea of treating a sequence of symbols (a "string" of symbols, as mathematicians sometimes say) as if it were a single symbol allows us to construct as many symbols as we want from a finite number of building blocks. That is, the rules of place-value notation (for Arabic numerals) and of spelling (for words in European languages) (that is, rules that tell us how to "concatenate" our symbols— to string them together) give us an arbitrarily large number (though still finite!) of symbols.

What about non-European languages? Turing makes a (possibly politically incorrect) joke:

> ...(Chinese, however, attempts to have an enumerable infinity of symbols). (p. 250.)

Chinese writing is pictographic and thus would seem to allow for symbols that run the risk of differing by an arbitrarily small extent, or, at least, that do not have to be constructed from a finite set of elementary symbols. As Turing also notes, using a finite number of basic symbols and rules for constructing complex symbols from them does not necessarily avoid the problem of not being able to identify or differentiate them:

> The differences from our point of view between the single and compound symbols is that the compound symbols, if they are too lengthy, cannot be observed at one glance. This is in accordance with experience. We cannot tell at a glance whether 9999999999999999 and 999999999999999 are the same. (p. 250.)

And probably you can't, either! So doesn't this mean that, even with a finite number of symbols, we're no better off than with an infinite number? Although Turing doesn't say so, we can solve this problem using the techniques he's given us: Don't try to write 15 or 16 occurrences of '9' inside one, tiny square: Write each '9' in a separate square! (And then count them to decide which sequence of them contains 15 and which contains 16, which is exactly how *you* "can tell...whether 9999999999999999 and 999999999999999 are the same.")

The other thing that Turing leaves unspecified here is the *minimum* number of elementary symbols we need. The answer, as we have seen, is: two (they could be a blank and '1', or '0' and '1', or any other two symbols). Turing himself will use a few more (just as we did in our addition example above, allowing for the 10 single-digit numerals together with '+' and '=').

8.7.2.1.4 States of mind.
So, let's assume that, to compute, we only need a 1-dimensional tape divided into squares and a finite number of symbols (minimally, two). What else?

> **(*)** The behaviour of *the computer* at any moment is determined by the symbols which *he* is observing, and *his* "state of mind" at that moment. (p. 250, my label and italics.)

I have always found this to be one of the most astounding and puzzling sentences! 'computer'? 'he'? 'his'? But it is only astounding or puzzling to those of us who live in the late 20th/early 21st century, when computers are machines, not humans! Recall the ad that we saw in §6.2 for a (human) computer from the 1892 *New York Times*. In 1936, when Turing was writing this article, *computers were still humans, not machines.* So, throughout this paper, whenever Turing uses the word 'computer', he means a *human* whose job it is to compute. I strongly recommend replacing (in your mind's ear, so to speak) each occurrence of the word 'computer' in this paper with the word 'clerk'.[9]

So, "the behavior of the *clerk* at any moment is determined by the symbols which he [or she!] is observing". In other words, the clerk decides what to do next by looking at the symbols, and *which* symbols the clerk looks at *partially* determines what the clerk will do. Why do I say 'partially'? Because the clerk also needs to know what to do with them: If the clerk is looking at two numerals, should they be added? Subtracted? Compared? The other information that the clerk needs is his or her "state of mind". What is that? Let's hold off on answering that question till we see what else Turing has to say.

> We may suppose that **there is a bound *B* to the number of symbols or squares which the computer [the clerk!] can observe at one moment.** If he[!] wishes to observe more, he must use successive observations. (p. 250, my interpolation and boldface.)

This is the second kind of finiteness (so, we have a finite number of different kinds of symbols and a finite number of them that can be observed at any given time.

---

[9]Some writers, such as [Sieg, 1994], use the nonce word 'computor' to mean a human who computes.

This upper bound B can be quite small; in fact, it can equal 1 (and $B = 1$ in most formal, mathematical presentations of Turing machines), but Turing is allowing for B to be large enough so that the clerk can read a single word without having to spell it out letter by letter, or a single numeral without having to count the number of its digits (presumably, the length of '9999999999999999' exceeds any reasonable B for humans). "Successive observations" will require the clerk to be able to move his or her eyes one square at a time to the left or right.[10]

> We will also suppose that **the number of states of mind which need to be taken into account is finite.** (p. 250, my boldface.)

Here, we have a third kind of finiteness. But we still don't know exactly what a "state of mind" is. Turing does tell us that:

> If we admitted an infinity of states of mind, some of them will be "arbitrarily close" and will be confused. (p. 250.)

—just as is the case with the number of symbols. And he also tells us that "the use of more complicated states of mind can be avoided by writing more symbols on the tape" (p. 250), but why that is the case is not at all obvious at this point. (Keep in mind, however, that we have jumped ahead from Turing's §1, so perhaps something that he said between then and now would have clarified this. Nevertheless, let's see what we can figure out.)

8.7.2.1.5 Operations.

So, a clerk who is going to compute needs only a (possibly finite) tape divided into squares and a finite number of different kinds of symbols; the clerk can look at only a bounded number of them at a time; and the clerk can be in only a finite number of "states of mind" at a time. Moreover, what the clerk can do (the clerk's "behavior") is determined by the observed symbols and his or her "state of mind". What kinds of behaviors can the clerk perform?

> Let us imagine the operations performed by the computer [the clerk] to be split up into "simple operations" which are so elementary that it is not easy to imagine them further divided. (p. 250.)

These are going to be the basic operations, the ones that all other operations will be constructed from. What could they be? This is an important question, because this is going to be the heart of computation.

> Every such operation consists of some change of the physical system consisting of the computer [the clerk] and his[!] tape. (p. 250.)

So, what "changes of the physical system" can the clerk make? The only things that can be changed are the clerk's state of mind (i.e., the clerk can change him- or herself, so to speak) and the tape, which would mean changing a symbol on the tape or changing which symbol is being observed. What else could there be? That's all we have to manipulate: the clerk, the tape, and the symbols. And all we've been told so far is that the clerk can write a symbol on the tape or observe one that's already written. Turing makes this clear in the next sentence:

---

[10]For more on this notion of bounds, see [Sieg, 2006, 16].

> We know the state of the system if we know the sequence of symbols
> on the tape, which of these are observed by the computer [by the clerk]
> (possibly with a special order), and the state of mind of the computer [of
> the clerk]. (p. 250.)

The "system" is the clerk, the tape, and the symbols. The only things we can know, or
need to know, are:

- which symbols are on the tape,

- where they are (their "sequence"),

- which are being observed (and in which order—the clerk might be looking from
  left to right, from right to left, and so on), and

- what the clerk's (still mysterious) "state of mind" is.

Here is the first "simple operation":

> We may suppose that in a simple operation not more than **one symbol is
> altered**. Any other changes can be split up into simple changes of this
> kind. (p. 250, my boldface.)

Altering a single symbol in a single square is a "simple" operation. (And alterations
of sequences of symbols can be accomplished by altering the single symbols in the
sequence.) How do you alter a symbol? You replace it with another one; that is, you
write down a (possibly different) symbol. (And perhaps you are allowed to erase a
symbol, but that can be thought of as writing a special "blank" symbol, 'ꞵ'.)

Which symbols can be altered? If the clerk is looking at the symbol in the first
square, can the clerk alter the symbol in the 15th square? Yes, but only by observing
the 15th square and *then* changing it:

> The situation in regard to the squares whose symbols may be altered in this
> way is the same as in regard to the observed squares. We may, therefore,
> without loss of generality, assume that the squares whose symbols are
> changed are always "observed" squares. (p. 250.)

But wait a minute! If the clerk has to be able to find the 15th square, isn't that a
kind of operation? Yes:

> Besides these changes of symbols, the simple operations must include
> changes of distribution of observed squares. The new observed squares
> must be immediately recognisable by the computer [by the clerk]. (p. 250.)

And how does the clerk do that? Is "finding the 15th square" a "simple" operation?
Maybe. How about "finding the 9999999999999999th square"? No:

> I think it is reasonable to suppose that they can only be squares whose
> distance from the closest of the immediately previously observed squares
> does not exceed a certain fixed amount. Let us say that each of the
> new observed squares is within L squares of an immediately previously
> observed square. (p. 250.)

So here we have a fourth kind of boundedness or finiteness: The clerk can only look a certain bounded distance away. How far can the distance be? Some plausible lengths are the length of a typical word or small numeral (so L could equal B). The minimum is, of course, 1 square (taking $L = B = 1$). So, another "simple" operation is looking one square to the left or to the right (and, of course, the ability to repeat that operation, so that the clerk can, eventually, find the 15th or the 9999999999999999th square).

What about a different kind of candidate for a "simple" operation: "find a square that contains the special symbol x":

> In connection with "immediate recognisability", it may be thought that there are other kinds of square which are immediately recognisable. In particular, squares marked by special symbols might be taken as immediately recognisable. Now if these squares are marked only by single symbols there can be only a finite number of them, and we should not upset our theory by adjoining these marked squares to the observed squares. (pp. 250–252.)

So, Turing allows such an operation as being "simple", because it doesn't violate the finiteness limitations. But he doesn't have to allow them. How would the clerk be able to find the only square that contains the special symbol x (assuming that there is one)? By first observing the current square. If x isn't on that square, then observe the next square to the left. If x isn't on that square, then observe the square to the right of the first one (by observing the square two squares to the right of the current one). And so on, moving back and forth, till a square with x is found. What if the clerk needs to find a sequence of squares marked with a sequence of special symbols?

> If, on the other hand, they [i.e., the squares marked by special symbols] are marked by a sequence of symbols, we cannot regard the process of recognition as a simple process. (p. 251.)

I won't follow Turing's illustration of how this can be done. Suffice it to say that it is similar to what I just sketched out as a way of avoiding having to include "finding a special square" as a "simple" operation, and Turing admits as much:

> If in spite of this it is still thought that there are other "immediately recognisable" squares, it does not upset my contention so long as these squares can be found by some process of which my type of machine is capable. (p. 251.)

In other words, other apparently "simple" operations that can be analyzed into some combination of the simplest operations of writing a symbol and observing are acceptable. It is worth noting that this can be interpreted as a claim that "subroutines" can be thought of as single operations.

Turing now summarizes his analysis of the minimum that a human computer (what I have been calling a "clerk") needs to be able to do in order to compute:

> The simple operations must therefore include:
>
> *a* Changes of the symbol on one of the observed squares.

b Changes of one of the squares observed to another square within *L* squares of one of the previously observed squares.

It may be that some of these changes necessarily involve a change of state of mind. The most general single operation must therefore be taken to be one of the following:

A A possible change (*a*) of symbol together with a possible change of state of mind.

B A possible change (*b*) of observed squares, together with a possible change of state of mind. (p. 251.)

In other words, the two basic operations are to write a symbol on the tape (and to change your "state of mind") and to look somewhere else on the tape (and to change your "state of mind"). That's it: writing and looking! Well, and "changing your state of mind", which we haven't clarified, but will next.

8.7.2.1.6 Conditions.   How does the clerk know which of these two things (writing or looking) to do? Turing's next remark tells us:

> The operation actually performed is determined, as has been suggested on p. 250, by the state of mind of the computer [that is, of the clerk] and the observed symbols. In particular, they determine the state of mind of the computer [that is, of the clerk] *after* the operation is carried out. (p. 251, my interpolation and italics.)

The passage on p. 250 that Turing is referring to is the one that I marked '(*)' and called 'astounding', above; it says roughly the same thing as the present passage. So, what Turing is saying here is that the clerk should

- *first* consider his or her state of mind *and* where he or she is currently looking on the paper,

- *then* decide what to do next (either write something there or look somewhere else), and, *finally*, change his or her state of mind.

Of course, after doing that, the clerk is in a (possibly) new state of mind and looking at a (possibly) new location on the paper, which means that the clerk is ready to do the next thing.

8.7.2.1.7 States of mind clarified.   Now, think about a typical computer program, especially an old-fashioned one, such as those written in (early versions of) Basic or Fortran, where each line of the program has a line number and a statement to be executed (a "command"). The computer (and here I mean the machine, not a clerk) starts at the first line number, executes the command, and then (typically) moves to the next line number; in "atypical" cases, the command might be a "jump" or "go to" command, which causes the computer to move to a different line number. At whatever line number the computer has moved to after executing the first command, it executes

the command at that new line number. And so on. But, if you compare this description with Turing's, you will see that *what corresponds to the line number of a program is Turing's notion of a "state of mind"!* And what is the currently observed symbol? It is the current input to the program!

So, let's paraphrase Turing's description of the basic operation that a clerk performs when computing. We'll write the paraphrase in terms of a computer program that the clerk is following:

> The operation performed is determined by the current line number of the program and the current input. The simple operations are: (a) print a symbol and (b) move 1 square left or right on the tape (which is tantamount to accepting new input), followed by changing to a new line number.[11]

We can also say this in a slightly different way:

> If the current line number is N and the current input is I,
> then print or move (or both) and go to line N'.

And a program for such a computer will consist of lines of "code" that look like this:

> Line N: **if** input = I
>   **then begin** print (or move); go to Line N' **end**
>   **else** go to Line N''

8.7.2.1.8 The Turing machine. I said above that passage (*) was "astounding"; here is its sequel:

> We may now construct a *machine* to do the work of this *computer.* (p. 251, my italics.)

Reading this sentence out of context can make it sound very confusing; after all, isn't a computer a machine? But, as we have seen, a computer (for Turing) is a human clerk who computes. And what Turing is now saying is that the *human* can be replaced by a *machine*, that is, by what we now call a computer (a mechanical device). This sentence marks the end of Turing's analysis of what a human computer does and the beginning of his mathematical construction of a mechanical computer that can do what the human does. His description of it here is very compact; it begins as follows:

---

[11]Here's another way to think about this: Suppose that we have *two* tapes. (Incidentally, it can be proved that any two-tape Turing machine is equivalent to a one-tape Turing machine; see [Dewdney, 1989, Ch. 28].) Tape 1 will be the one we have been discussing so far, containing input (the symbols being scanned) and output (the symbols being printed). Tape 2 will contain the computer's program, with each square representing a "state of mind". The computer can be thought of as starting in a square on Tape 2, executing the instruction in that square (by reading from, and writing to, a square on Tape 1 and then moving to a(nother) square on Tape 1), and then moving to a(nother) square on Tape 2, and repeating this "fetch-execute" loop. In reality, Turing machines only have one tape, and the instructions are not written anywaher; rather, they are "hardwired" into the Turing machine. Any written verion of them is (merely) a description of the Turing machine's behavior (or of its "wiring diagram"). But, if we encode Tape 2 on a portion of Tape 1, then we have a "stored-program"—or universal—computer. For a similar presentation of universal Turing machines, see [Dewdney, 1989, Ch. 48].

> To each state of mind of the computer [of the clerk!] corresponds an "*m*-configuration" of the machine.

So, an *m*-configuration is something in the machine that corresponds to a line number of a program.[12] But, considering modern computers and programs, programs are separate from the computers that run them, so what could Turing mean when he says that an *m*-configuration belongs to a machine? He means that the machine is "hardwired" (as we would now say) to execute exactly one program (exactly one algorithm). Being hardwired, no separate program needs to be written out; it is already "compiled" into the machine. A "Turing machine" can do one and only one thing; it can compute one and only one function, using an algorithm that is hardwired into it.

Turing continues:

> The *machine* scans *B* squares corresponding to the *B* squares observed by the *computer*. (p. 251, my italics.)

A modern "translation" of this sentence would say: "The *computer* scans *B* squares corresponding to the *B* squares observed by the *clerk*." The clerk is limited to observing a maximum of *B* squares on the tape, as we saw above (in an earlier quote from p. 250). The machine analogue of that is to move, or "scan", *B* squares to the left or right on the tape. In modern mathematical treatments of Turing machines, $B = 1$.

Turing continues:

> In any move the machine can change a symbol on a scanned square or can change any one of the scanned squares to another square distant not more than *L* squares from one of the other scanned squares. (pp. 251–252.)

In other words, the machine (the Turing machine, or modern hardwired computer) can pay attention to *B* squares at a time, and each line of its program allows it to print a new symbol on any of those squares or move to any other square that is no more than *L* squares away from any of the *B* squares. Again, modern treatments simplify this: The machine is scanning a single square, and each line of its program allows it to print a new symbol on that square or to move one square to its left or right (or both print and move).

Which "move" should the machine make?

> The move which is done, and the succeeding configuration, [that is, the next *m*-configuration; that is, the next step in the algorithm], are determined by the scanned symbol and the [current] *m*-configuration. (p. 252.)

That is, the move that the machine should make, as well as the *next m*-configuation (that is, the next step in the algorithm) are determined by the currently scanned symbol and the *current m*-configuration. Or, put in terms of computer programs, the instruction on the current line number together with the current input together determine what to do *now* (print, move, or both) and what to do *next* (which instruction to carry out next).

---

[12]Why '*m*'? It could stand for 'man', on the grounds that this is a machine analogue of a (hu)man's state of mind; or it could stand for 'mental', on the grounds that it is an analogue of a state of mind. But I think it most likely stands for 'machine', because it is a configuration, or state, of a machine. Of course, Turing might have intended it to be ambiguous among all these options.

8.7.2.1.9 Turing's thesis.

> The machines just described do not differ very essentially from computing
> machines as defined in §2, and corresponding to any machine of this type
> a computing machine can be constructed to compute the same sequence,
> that is to say the sequence computed by the computer. (p. 252.)

As for the first clause, please recall that we are in the middle of a very long digression
in which we have skipped ahead to Turing's §9 from Turing's §1; we have not yet read
Turing's §2. When we do, we will see a more detailed version of the machines that
Turing has just described for us here in §9.

The next clause is a bit ambiguous. When Turing says "any machine of *this* type",
is he referring to the machines of §9 or the machines of §2? It probably doesn't matter,
because he has said that the two kinds of machines "do not differ very essentially" from
each other. But I think that he is, in fact, referring to the machines of §9; "computing
machines" are the ones that are "defined in §2".

The last phrase is of more significance: These ("computing") *machines* (of Turing's
§2) "compute the same sequence...computed by the" *clerk*. In other words, whatever
a human clerk can do, these machines can also do. What a human clerk can do (i.e.,
which sequences, or functions, a human clerk can compute) is captured by the *informal*
notion of algorithm or computation. "These machines" are a formal counterpart, a
formal "explication" of that informal notion. So this last phrase is a statement of
Turing's thesis (that is, Church's thesis, or the Computation Thesis).

What about the other direction? Can a human clerk do everything that one of these
machines can do? Or are these machines in some way more powerful than humans? I
think the answer should be fairly obvious: Given the way the machines are constructed
on the basis of what it is that humans can do, surely a human could follow one of the
programs for these machines. So humans can do everything that one of the machines
can do, and—by Turing's thesis—these machines can do everything that humans can do
(well, everything that is computable in the informal sense). But these are contentious
matters, and we will return to them when we consider the controversies surrounding
hypercomputation (Ch. 11) and AI (Ch. 19).

For now, we have come to the end of our digression, and we now return to Turing's
§1, "Computing Machines".

8.8   Section 1, Paragraph 2

This is the paragraph in which Turing gives a more detailed presentation of his abstract
computing machine, the outcome of his detailed analysis from §9 of *human* computing.
He begins as follows:

> We may compare a man in the process of computing a real number
> to a machine which is only capable of a finite number of conditions
> $q_1, q_2, \ldots, q_R$ which will be called "*m*-configurations". (p. 231.)

Why "may" we do this? Turing will give his justification in his §9, which we have just
finished studying. By a "man", Turing of course means a human, not merely a male

human. To compute a real number is to compute the output of a real-valued function. And, as we have already seen, an *m*-configuration is a line of a computer program, that is, a step in an algorithm. Here, Turing is saying that each such algorithm has a finite number (namely, *R*) of steps, each labeled q_i. Put otherwise, (human, or informal) computation can be "compared with" (and, by Turing's thesis, identified with) a finite algorithm.

What else is needed?

> The machine is supplied with a "tape" (the analogue of paper) running through it, and divided into sections (called "squares") each capable of bearing a "symbol". (p. 231.)

There are a couple of things to note here. First, from our study of Turing's §9, we know why this is the case and what, exactly, the tape, squares, and symbols are supposed to be and why they are the way they are.

But, second, why does he put those three words in "scare quotes"? There are two possible answers. I suspect that the real answer is that Turing hasn't, at this point in his paper, explained in detail what they are; that comes later, in his §9.

But there is another possible reason, a mathematical reason: In Turing's formal, mathematical notion of a computing machine, the concepts of "tape", "squares" of a tape, and "symbols" are really *undefined (or primitive) terms* in exactly the same way that 'point', 'line', and 'plane' are undefined (or primitive) terms in Euclidean plane geometry. As Hilbert famously observed, "One must be able to say at all times—instead of points, lines, and planes—tables, chairs, and beer mugs".[13] So, here, too, one must be able to say at all times—instead of tapes, squares, and symbols—tables, chairs (but I'll use place settings; it will make more sense, as you will see), and beer mugs. A Turing machine, we might say, must have a table. Each table must have a sequence of place settings associated with it (so we must be able to talk about the *n*th place setting at a table). And each place setting can have a beer mug on it; there might be different kinds of beer mugs, but they have to be able to be distinguished from each other, so that we don't confuse them. In other words, *it is the logical or mathematical **structure** of a computing machine that matters, not what it is made of.* So, a "tape" doesn't have to be made of paper, a "square" doesn't have to be a regular quadrilateral that is physically part of the "tape" (chairs—or place settings—are presumably physically separate from their associated table), and "symbols" only have to be capable of being "borne" by a "square" (a numeral can be written on a square of the tape, a beer mug can be placed at a place setting belonging to a table).

Turing continues:

> At any moment there is just one square, say the *r*-th, bearing the symbol $S(r)$ which is "in the machine". We may call this square the "scanned square". (p. 231.)

First, 'S' is just the capital letter '*S*' in a font called "German Fraktur", or "black letter". It's a bit hard to read, so I will replace it with '*S*' in what follows (even when quoting Turing).

---

[13]Cf. Hilbert's *Gesammelte Abhandlungen* ("Complete Works"), vol. 3, p. 403, as cited in [Coffa, 1991, 135]; see also [Shapiro, 2009, 176].

Note, second, that this seems to be a slight simplification of his §9 analysis, with $B = 1$. Second, being "in the machine" might be another undefined (or primitive) term merely indicating a relationship between the machine and something else. But what else? Turing's punctuation allows for some ambiguity.[14] It might be the *symbol* (whatever it is, '0', '1', or a beer mug) that is in the machine, or it might be the *scanned square*. I think that it is the latter, from remarks that he makes next.

> The symbol on the scanned square may be called the "scanned symbol".
> The "scanned symbol" is the only one of which the machine is, so to speak,
> "directly aware". (p. 231.)

Here, Turing's scare quotes around 'directly aware', together with the hedge 'so to speak', clearly indicate that he is not intending to anthropomorphize his machine. His machines are not really "aware" of anything; only humans can be really "aware" of things. But the machine analogue of human awareness is: being a scanned symbol. There is nothing anthropomorphic about that: Either a square is being scanned (perhaps a light is shining on a particular place setting at the table) or it isn't, and either there is a symbol on the scanned square (there is a beer mug at the lighted place setting), or there isn't.

> However, by altering its *m*-configuration the machine can effectively
> remember some of the symbols which it has "seen" (scanned) previously.
> (p. 231.)

What does this mean? Let's try to paraphrase it: "By altering the line number of its program, the computing machine can effectively..."—can effectively do what? It can "remember previously scanned symbols". This is to be contrasted with the *currently* scanned symbol. How does the machine "remember" "by altering a line number"? Well, how would it "remember" what symbol was on, say, the 3rd square if it's now on the 4th square? It would have to move left one square and scan the symbol that's there. To do that, it would have to have an instruction to move left. And to do *that*, it would need to go to that instruction, which is just another way of saying that it would have to "alter its *m*-configuration". (For another slow-reading analysis of this sentence, see [Dresner, 2003, Dresner, 2012].)

> The *possible* behaviour of the machine at any moment is determined by
> the *m*-configuration q_n and the scanned symbol $S(r)$. (p. 231, my italics.)

It is only a *possible* behavior, because a given line of a program is only executed when control has passed to that line. If it is not being executed at a given moment, then it is only *possible* behavior, not *actual* behavior. The machine's *m*-configuration is the

---

[14]More precisely, his *lack* of punctuation: If 'which' is preceded by a comma, then "which is 'in the machine' " is a "non-restrictive relative clause" that refers to the square. With no comma, the "which" clause is a "restrictive" relative clause modifying 'symbol $S(r)$'. For more on relative clauses, see " 'Which' vs. 'that' ", online at http://www.cse.buffalo.edu/~rapaport/howtowrite.html#whichVthat
(accessed 17 January 2014).

analogue of a line number of a program, and the scanned symbol is the analogue of the external input to the machine.[15]

> This pair $q_n, S(r)$ will be called the "configuration": thus the configuration determines the possible behaviour of the machine. (p. 231.)

Giving a single name ('configuration') to the *combination* of the *m*-configuration *and* the currently scanned symbol reinforces the idea that the *m*-configuration alone is an analogue of a line number and that this combination is the condition (or antecedent) of a condition-action (or a conditional) statement: Line q_n begins, "**if** the currently scanned symbol is $S(r)$, **then**...", or "**if** the current instruction is the one on line q_n **and if** the currently scanned symbol is $S(r)$, **then**...".

What follows the '**then**'? That is, what should the machine do if the condition is satisfied?

> In some of the configurations in which the scanned square is blank (*i.e.* bears no symbol) the machine **writes** down a new symbol on the scanned square: in other configurations it **erases** the scanned symbol. The machine may also **change the square** which is being scanned, but only by shifting it one place to right or left. In addition to any of these operations **the *m*-configuration may be changed**. (p. 231, my boldface.)

So, we have 5 operations:

1. write a new symbol

2. erase the scanned symbol

3. shift 1 square left

4. shift 1 square right

5. change *m*-configuration.

The first 4 of these can be simplified to only 2 operations, each of which is slightly more complex:

1′ write a new symbol *including* ♭

2′ shift (which is now an operation that takes an argument: left or right).

There are four things to note:

- First, the symbols are left unspecified (which is why we can feel free to add a "blank" symbol), though, as we have seen, they can be limited to just '0' and '1' (and maybe also '♭').

---

[15]It is also possible, given what the reader knows at this stage of Turing's paper, not yet having read his §9, that an *m*-configuration is the entire internal state of the machine, perhaps encoding what could be called the machine's "prior" or "background" knowledge—in contrast to external information from the outside world, encoded in the scanned symbol.

- Second, Turing has, again, simplified his §9 analysis, letting $L = 1$.

- Third, "change *m*-configuration" is essentially a "jump" or "go to" instruction. The whole point of structured programming, as we have seen, is that this can be eliminated—so we really only need the first two of our slightly more complex operations, as long as we require our programs to be structured.

- Fourth, there is no "halt" command. (In §8.9.3, below, we will see why this is not needed.)

Turing next clarifies what symbols are needed. Recall that the kind of computation that Turing is interested in is the computation of the decimal of a real number.

> Some of the symbols written down will form the sequence of figures which is the decimal of the real number which is being computed. The others are just rough notes to "assist the memory". It will only be these rough notes which will be liable to erasure. (pp. 231–232.)

So, either we need symbols for the 10 Arabic numerals (if we write the real number in decimal notation) or we only need symbols for the 2 binary numerals (if we write the real number in binary notation). Any other symbols are merely used for bookkeeping, and they (and only they) can be erased afterwards, leaving a "clean" tape with only the answer on it.

There is one more thing to keep in mind: Every real number (in decimal notation)[16] has an infinite sequence of digits to the right of the decimal point, even if it is an integer or (a non-integer) rational number, which are typically written with either no digits or a finite number of digits in the decimal expansion (1, 1.0, 2.5, etc.). If the number is an integer, this is an infinite sequence of '0's; for example, $1 = 1.000000000000\ldots$ (which I will abbreviate as $1.\overline{0}$). If the number is rational, this is an infinite sequence of some repeating subsequence; for example:

$$\frac{1}{2} = 0.500000000000\ldots = 0.5\overline{0}$$

$$\frac{1}{3} = 0.333333333333\ldots = 0.\overline{3}$$

$$\frac{1}{7} = 0.142857142857\ldots = 0.\overline{142857}$$

And if the number is irrational, this is an infinite, *non*-repeating sequence; for example:

$$\sqrt{2} = 1.41421356237309\ldots$$

$$\pi = 3.1415926535\ldots$$

What this means is that one of Turing's computing machines should *never halt* when computing (i.e., writing out) the decimal of a real number. It should only halt if it is writing down a *finite* sequence, and it can do this in two ways: It could write down the

---

[16]Similar remarks can be made for binary notation.

finite sequence and then halt. Or it could write down the finite sequence and then go into an infinite loop (either rewriting the last digit over and over in the same square, or just looping in a do-nothing operation such as the empty program).

Finally,

> It is my contention that these operations include all those which are used in the computation of a number. (p. 232.)

This is another statement of Turing's thesis: To compute, all you have to do is arrange the operations of writing and shifting in a certain way. The way they are arranged—what is now called "the control structure of a computer program"—is controlled by the "configuration" and the change in *m*-configuration (or, in modern structured programming, by Boehm and Jacopini's three control structures of sequence, selection, and while-repetition). For Turing, it goes unsaid that all computation can be reduced to the computation of a number; this is the insight we discussed in §7.6.1 that all the information about any computable problem can be represented using only '0' and '1'; hence, any information—including pictures and sounds—can be represented as a number. (But it is also important to realize that this kind of universal binary representation of information doesn't have to be thought of as a number, because the two symbols don't have to be '0' and '1'!)

8.9   Section 2: "Definitions"

We are now ready to look at the section in which Turing's "computing machines" are defined, as we read in his §9 (see our §8.7.2.1.9, above).

8.9.1   "Automatic Machines"

Turing begins by giving us a sequence of definitions. The first is the most famous:

> If at each stage the motion of a machine (in the sense of §1) is *completely* determined by the configuration, we shall call the machine an "automatic machine" (or *a*-machine). (p. 232.)

Clearly, such a machine's "motion" (or behavior) is at least *partly* determined by its configuration (that is, by its *m*-configuration, or line number, together with its currently scanned symbol). Might it be determined by anything else? For all that Turing has said so far, maybe such a machine's human operator could "help" it along by moving the tape for it, or by writing something on the tape. This definition rules that out by limiting our consideration to such machines whose "motion" "is *completely* determined by the configuration". So, a human operator is not allowed to "help" it in any way: No cheating allowed! *Turing* may have called such a machine an '*a*-machine'; *we* now call them—in his honor—'Turing machines'.[17] seems to have first been used

What about machines that get outside help?

---

[17] Alonzo Church seems to have been the first person to use this term, in his review of Turing's paper [Church, 1937].

> For some purposes we might use machines (choice machines or *c*-machines) whose motion is only partially determined by the configuration (hence the use of the word "possible" in §1). When such a machine reaches one of these ambiguous configurations, it cannot go on until some arbitrary choice has been made by an external operator. (p. 232.)

First, note that Turing's explanation of the use of 'possible' may be slightly different from mine. But I think that they are consistent explanations. In the previous statements, Turing used 'possible' to *limit* the kind of operations that a Turing machine could perform. Here, he is introducing a kind of machine that has another kind of possible operation: writing, moving, or changing *m*-configuration *not* as the result of an explicit instruction *but* as the result of a "choice... made by an external operator". Note, by the way, that this external operator *doesn't have to be a human*; it *could* be another Turing machine! In any case, such *c*-machines were the topic of Turing's doctoral dissertation under Church, in which he called them 'oracle machines', or *o*-machines. Such machines are now used to model *interactive* computing, and we will return to them in Ch. 11 on hypercomputation.

8.9.2 "Computing Machines"

Turing gives us some more definitions:

> If an *a*-machine prints two kinds of symbols, of which the first kind (called figures) consists entirely of 0 and 1 (the others being called symbols of the second kind), then the machine will be called a computing machine. (p. 232.)

The principal definition here is that of 'computing machine', a special case of an *a*- (or Turing) machine that outputs its results as a binary numeral (in accordance with the first insight we discussed, in §7.6.1). Once again, here Turing is simplifying his §9 analysis of human computation, restricting the symbols to '0' and '1'. Well, not quite, because he also allows "symbols of the second kind", used for bookkeeping purposes or intermediate computations. Note, however, that any symbol of the second kind could be replaced—at the computational cost of more processing—by sequences of '0's and '1's.

Turing continues:

> If the machine is supplied with a blank tape and set in motion, starting from the correct initial *m*-configuration, the subsequence of the symbols printed by it which are of the first kind will be called the *sequence computed by the machine*. (p. 232.)

Here, he seems to be allowing for some of the symbols of the *second* kind to remain on the tape, so that only a *sub*sequence of the printed output constitutes the result of the computation. In other words, these secondary symbols need not be erased. One way to think of this is to compare it to the way we write decimal numerals greater than 999, namely, with the punctuation aid of the non-numerical symbols known as a 'comma' and a 'decimal point': 1,234,567.89

I almost wrote, "to remain on the tape *after the computation halts*". But does it halt? It can't—because every real number has an *infinite* decimal part! The secondary symbols could still be erased, during the computation; that's not of great significance (obviously, it's easier to not erase them and to just ignore them). The important point to remember is that computations of decimal representations of real numbers never halt. We'll return to this in a moment.

One more small point that simplifies matters:

> The real number whose expression as a binary decimal is obtained by prefacing this sequence by a decimal point is called the *number computed by the machine.* (p. 232.)

What about the part of the expression that is to the *left* of the decimal point? It looks as if the only numbers that Turing is interested in computing are the reals between 0 and 1 (presumably including 0, but excluding 1). Does this matter? Not really; first, all reals can be mapped to this interval, and, second, any other real can be computed simply by computing its "non-decimal" part in the same way. Restricting our attention to this subset of the reals simplifies the discussion without loss of generality.

A couple more definitions:

> At any stage of the motion of the machine, the number of the scanned square, the complete sequence of all symbols on the tape, and the *m*-configuration will be said to describe the *complete configuration* at that stage. The changes of the machine and tape between successive complete configurations will be called the *moves* of the machine. (p. 232.)

Three points to note: First, at any stage of the motion of the machine, only a *finite* number of symbols will have been printed, so it is perfectly legitimate to speak of "the complete sequence of all symbols on the tape" even though every real number has an *infinite* number of numerals after the decimal point. Second, the sequence of all symbols on the tape probably includes all occurrences of ♭ that do not occur after the last non-blank square (that is, that *do* occur *before* the last non-blank square); otherwise, there would be no way to distinguish the sequence $\langle 0, 0, 1, ♭, 0\rangle$ from the sequence $\langle ♭, 0, ♭, 0, ♭, 1, 0\rangle$.

Third, we now have three notions called 'configurations'; let's summarize them for convenience:

1. *m*-configuration = line number, q_n, of a program for a Turing machine.

2. configuration = the pair: $\langle q_n, S(r)\rangle$,
   where $S(r)$ is the symbol on the currently scanned square, r.

3. complete configuration = the triple:
   $\langle r$, the sequence of all symbols on the tape,[18] $q_n\rangle$.

---

[18]As described in the last paragraph.

8.9.3 "Circular and Circle-Free Machines"

We now come to what I have found to be one of the most puzzling sections of Turing's paper. It begins with the following definitions:

> If a computing machine never writes down more than a finite number of symbols of the first kind, it will be called *circular*. Otherwise it is said to be *circle-free*. (p. 233.)

Let's take this slowly: A computing machine is a Turing machine that only prints a binary representation of a real number together with a few symbols of the second kind. If such a machine *never* writes down more than a finite number of '0's and '1's, then, trivially, it has *only* written down a *finite* number of such symbols. *That means that is has halted!* And, in that case, Turing wants to call it 'circular'! But, to my ears, at least, 'circular' sounds like 'looping', which, in turn, sounds like it means "not halting".

And, if it *does* write down more than a finite number of '0's and '1's, then, trivially, it writes down an *infinite* number of them. *That means that it does not halt!* In that case, Turing wants to call it 'circle-free'! But that sounds like 'loop-free', which, in turn, sounds like it means that it *does* halt.

What's going on? Before looking ahead to see if, or how, Turing clarifies this, here's one guess: The only way that a Turing machine can print a finite number of "figures" (Turing's name for '0' and '1') and still "be circular" (which I am interpreting to mean "loop") is for it to keep repeating printing—that is, to "overprint"—some or all of them, that is, for it to "circle back" and print some of them over and over again. (In this case, no "halt" instruction is needed!)

And the only way that a Turing machine can print an infinite number of "figures" and also be "circle-free" is for it to continually print new figures to the right of the previous one that it printed (and, thus, not "circle back" to a previous square, overprinting it with the same symbol that's on it).

Is that what Turing has in mind? Let's see. The next paragraph says:

> A machine will be circular if it reaches a configuration from which there is no possible move or if it goes on moving, and possibly printing symbols of the second kind, but cannot print any more symbols of the first kind. The significance of the term "circular" will be explained in §8. (p. 233.)

The first sentence is rather long; let's take it phrase by phrase: "A machine will be circular"—that is, will print out only a finite number of figures—if [Case 1] it reaches a configuration from which there is no possible move." That is, it will be circular if it reaches a line number q_n and a currently scanned symbol $S(r)$ from which there is no possible move. How could that be? Easy: if there's no line of the program of the form: "Line q_n: **If** currently scanned symbol $= S(r)$ **then**.... In that case, the machine stops,[19] because there's no instruction telling it to do anything.[20]

---

[19] At this point, I cannot resist recommending that you read E.M. Forster's wonderfully prescient, 1909(!) short story, "The Machine Stops", [Forster, 1909].

[20] Another possibility is that line q_n says: **If** currently scanned symbol $= S(r)$, **then** go to line q_n. In that case, the machine never stops, becuase it forever loops (circles?) back to the same line.

That's even more paradoxical than my interpretation above; here, he is clearly saying that a machine is circular if it halts! Of course, if you are the operator of a Turing machine and you are only looking at the tape (and not at the machinery), would you be able to tell the difference between a machine that was printing the same symbol over and over again on the same square and a machine that was doing nothing?[21] Probably not. So, from an external, behavioral point of view, these would seem to amount to the same thing.

But Turing goes on: A machine will also be circular "if [Case 2] it goes on moving, and possibly printing [only] symbols of the second kind" but not printing any more "figures". Here, the crucial point is that the machine does not halt but goes on moving. It might or might not print *anything*, but, if it does, it only prints secondary symbols. So we have the following possibilities: a machine that keeps on moving, spewing out square after square of blank tape; or a machine that keeps on moving, occasionally printing a secondary symbol. In either case, it has only printed a *finite* number of *figures*. Because it has, therefore, *not* printed an *infinite* decimal representation of a real number, it has, for all practical purposes, halted—at least in the sense that it has finished its task, though it has not succeeded in computing a real number.

Once again, a machine is circular if it halts (for all practical purposes; it's still working, but just not doing anything significant). This isn't what I had in mind in my interpretation above. But it does seem to be quite clear, no matter how you interpret what Turing says, that he means that *a circular machine is one that does not compute a real number*, either by halting or by continuing on but doing nothing userful (not computing a real number). Machines that *do* compute real numbers are "circle-free", but they must also never halt; they must loop forever, in modern terms, but continually doing useful work (computing digits of the decimal expansion of a real number).[22]

In other words, a "good" Turing machine is a "circle-free" one that does *not* halt and that continually computes a real number. This seems to be contrary to modern terminology and the standard analysis of "algorithms" that we saw in §7.5. And how does this fit in with Turing's claim at the beginning of his paper that "the 'computable' numbers may be described briefly as the real numbers whose expressions as a decimal are calculable *by finite means*" (my italics)? The only way that I can see to make these two claims consistent is to interpret "by finite means" to refer to the number of steps in an algorithm, or the amount of time needed to carry one step out, or the number of operations needed to carry one step out (in case any of the steps are not just basic operations). It cannot mean, as we have just seen, that the entire task can be completed in a finite amount of time or that it would necessarily halt.

Finally, what about the allusion to Turing's §8? That section, which we will not investigate and which is uninformatively titled "Application of the Diagonal Process", is the section in which he proves that the Halting Problem is not computable (more precisely, that a Gödel-like number of a program for the Halting Problem is not a computable number). And, pretty obviously, his proof is going to be a little bit different from the one that we sketched in §7.8 because of the difference between

---

[21]The machine described in the text and the machine described in the previous footnote have this property.

[22]"A machine that computes a real number in this sense was called *circle-free*; one that does not (because it never prints more than a finite number of 0s and 1s) was called *circular*" [Davis, 1995b, 141].

our modern idea that only Turing machines that halt are "good" and Turing's idea that only Turing machines that are circle-free are "good". The interested reader is referred to [Petzold, 2008, Ch. 10], who notes, by the way, that the concept of "halting" was introduced into the modern literature by Martin Davis [Petzold, 2008, 179], "despite the fact that Turing's original machines never halt!" [Petzold, 2008, 329]. [van Leeuwen and Wiedermann, 2013] analyzes these notions in modern terms.

8.9.4   Section 2, Subsection "Computable Sequences and Numbers"

Here are Turing's final definitions from this section:

> A *sequence* is said to be computable if it can be computed by a circle-free machine. (p. 233, my italics.)

Although this is presented as a definition of 'computable' (actually, of 'computable sequence'), it can, in fact, be understood as another statement of Turing's thesis. Being "computable by a circle-free machine" is a very precise, mathematical concept. In this definition, I think that Turing is best understood as suggesting that this precise concept should replace the informal notion of being "computable". Alternatively, Turing is saying here that he will use the word 'computable' in this very precise way.

> A *number* is computable if it differs by an integer from the number computed by a circle-free machine. (p. 233, my italics.)

Circle-free machines compute (by printing out) a sequence of figures (a sequence of '0's and '1's). Such a sequence can be considered to be a decimal (actually, a binary) representation of a number between 0 and 1 (including 0, but not including 1). Here, Turing is saying that *any* real number can be said to be computable if it has the same decimal part (that is, the same part after the decimal point) of a number representable as a computable sequence of figures. (So, for instance, $\pi = 3.1415926535\ldots$ differs by the integer 3 from the number $0.1415926535\ldots$ computable by a circle-free Turing machine.)

8.10   Section 3: "Examples of Computing Machines"

We are now ready to look at some "real" Turing machines, more precisely, "computing machines", which, recall, are "automatic" *a*-machines that print only figures ('0', '1') and maybe symbols of the second kind. Hence, they compute real numbers. Turing gives us two examples, which we will look at in detail.

8.10.1   Example I

> A machine can be constructed to compute the sequence $010101\ldots$. (p. 233.)

Actually, as we will see, it prints

$$0\flat1\flat0\flat1\flat0\flat1\flat\ldots$$

What real number is this? First, note that it is a *rational* number of the form $0.\overline{01}$.
Treated as being written in binary notation, it $=\frac{1}{3}$; treated as being written in decimal
notation, it $=\frac{1}{99}$.

> The machine is to have the four *m*-configurations "$\mathfrak{b}$", "$\mathfrak{c}$", "$\mathfrak{f}$", "$\mathfrak{e}$" and is
> capable of printing "0" and "1". (p. 233.)

The four line numbers are (in more legible italic font): *b*, *c*, *f*, *e*.

> The behaviour of the machine is described in the following table in which
> "*R*" means "the machine moves so that it scans the square immediately
> on the right of the one it was scanning previously". Similarly for "*L*".
> "*E*" means "the scanned symbol is erased" and "*P*" stands for "prints".
> (p. 233.)

This is clear enough. It is, however, interesting to note that it is the Turing machine
that moves, not the tape!

Before going on with this paragraph, let's look at the "table". In later writings by
others, such tables are sometimes called 'machine tables'; *they are computer programs
for Turing machines*, written in a "Turing-machine programming language" that Turing
is here giving us the syntax and semantics for.

However, it is important to keep in mind that the Turing machine does not "consult"
this table to decide what to do. We humans would consult it in order to simulate
the Turing machine's behavior. But the Turing machine itself simply behaves *in
accordance with* that table, not by *following* it. The table should be thought of as a
mathematical-English description of the way that the Turing machine is "hardwired"
to behave. (We'll revisit this idea in §10.5.1.)

Here's the table, written a bit more legibly than in Turing's paper:

Configuration		*Behaviour*	
m-config.	*symbol*	*operations*	*final m-config.*
b	None	*P*0,*R*	*c*
c	None	*R*	*e*
e	None	*P*1,*R*	*f*
f	None	*R*	*b*

This program consists of 4 lines. It is important to note that it is a *set* of lines, not a
sequence: The order in which the lines are written down in the table (or "program") is
irrelevant; there will never be any ambiguity as to which line is to be executed. Perhaps
a better way of saying this is: There will never be any ambiguity as to which line is
causing the Turing machine to move.

Each line consists of two principal parts: a "configuration" and a "behavior". Each
configuration, as you may recall, consists of two parts: an *m*-configuration (or line

number) and a symbol (namely, the currently scanned symbol). Each behavior consists also of two parts: an "operation" (one or more of *E*, *L*, *R*, or *P*) and a "final *m*-configuration" (that is, the next line number to be executed).

> This table (and all succeeding tables of the same kind) is to be understood to mean that **for a configuration described in the first two columns the operations in the third column are carried out successively, and the machine then goes over into the *m*-configuration described in the last column.** (p. 233, my boldface.)

That is, each line of the program should be understood as follows: "Under the circumstances described by the configuration, do the operation and then go to the instruction at the final *m*-configuration". Or, to use Turing's other terminology: "If your current state of mind is the one listed in the current *m*-configuration, and if the symbol on the current square being scanned is the one in the *symbol* column, then do the *operation* and change your state of mind to the one in the *final m-configuration* column."

A modern way of thinking about this is to consider it to be a "production system". Production systems are an architecture introduced by Emil Post in his analysis of computation and used by many researchers in AI [Post, 1941]. A production system consists of a set of "condition-action" pairs; if the condition is satisfied, then the action is carried out. That's exactly what we have in this Turing-machine table: The configuration is the condition, and the behavior is the action.[23]

A further qualification:

> When the second column [that is, the symbol column] is left blank, it is understood that the behaviour of the third and fourth columns applies for any symbol and for no symbol. (p. 233.)

That is the situation we have in this first example, where 'None' is the entry in each row of the symbol column. So the only condition determining the behavior of this Turing machine is its current "state of mind", that is, its current line number.

Finally, we need to know what the initial situation that this "production system" is intended to apply to:

> The machine starts in the *m*-configuration *b* with a blank tape. (p. 233.)

Perhaps '*b*' stands for "begin", with subsequent "states of mind" (in alphabetical as well as sequential order) being *c*, *e*, and *f* ('*f*' for "final"? What happened to '*d*'?).

Let's trace this program. We start with a blank tape, which I will show as follows:

ƀƀƀƀƀƀƀƀƀƀ ...

We are in state *b*.

Looking at the table, we see that if we are in state *b*, then (because any symbol that might be on the tape is irrelevant), we should do the sequence of operations *P*0, *R*.

---

[23][Sieg, 2000, 7] notes that even Turing considered Turing machines as production systems.

Turing hasn't told us what '*P*0' means, but, because '*P*' means "print", it's pretty obvious that this means "print 0 on the currently scanned square".

Note, too, that he hasn't told us which square is currently being scanned! It probably doesn't matter,[24] but it's worth thinking about some of the options. One possibility is that we are scanning the first, or leftmost, square; this is the most likely option and the one that I will assume in what follows. But another possibility is that we are scanning some other square somewhere in the "middle" of the tape. That probably doesn't matter, because Turing only told us that it would compute the sequence '010101...'; he didn't say *where* it would be computed! There is one further possbility, not very different from the prvious one: The tape might not have a "first" square—it might be infinite in both directions! And now we need to consider something that Turing hasn't mentioned: How long is the tape? For all that he has told us, it could be infinitely long. But many mathematicians and philosophers (not to mention engineers!) are not overly fond of actual infinities. So, we can say, instead, that, at any moment, the tape only has a finite number of squares, *but* there is no limit to the number of extra squares that we are allowed to add on at one (or maybe both) ends. (As my former teacher and colleague John Case used to put it, if we run out of squares, we can always go to an office-supply store, buy some extra squares, and staple them onto our tape!)

So, let's now show our intial tape as follows, where the currently scanned square is underlined:

$$\underline{♭} \, ♭ \, ♭ \, ♭ \, ♭ \, ♭ \, ♭ \, ♭ \, ♭ \, ♭ \ldots$$

Performing the two operations on line *b* converts our initial tape to this one:

$$0 \, \underline{♭} \, ♭ \, ♭ \, ♭ \, ♭ \, ♭ \, ♭ \, ♭ \, ♭ \ldots$$

and puts us in state *c*. That is, we next execute the instruction on line *c*.

Looking at line *c*, we see that, no matter what symbol is on the current square (it is, in fact, blank), we should simply move right one more square and change our mind to *e*. So now our tape will look like this:

$$0 \, ♭ \, \underline{♭} \, ♭ \, ♭ \, ♭ \, ♭ \, ♭ \, ♭ \, ♭ \ldots$$

Because we are now in state *e*, we look at line *e* of the program, which tells us that, no matter what, if anything, is on the current square, print '1' there, move right again, and go into state *f*. So, our tape becomes:

$$0 \, ♭ \, 1 \, \underline{♭} \, ♭ \, ♭ \, ♭ \, ♭ \, ♭ \, ♭ \ldots$$

Now we are in state *f*, so, looking at line *f*, we see that we merely move right once again, yielding:

$$0 \, ♭ \, 1 \, ♭ \, \underline{♭} \, ♭ \, ♭ \, ♭ \, ♭ \, ♭ \ldots$$

---

[24]It probably doesn't matter, because all squares on the tape are blank. If the tape is infinite (or endless) in both directions, then each square is indistinguishable from any other square, at least until something is printed on one square.

And we go back into state *b*. But that starts this cycle all over again; we are indeed in an infinite loop! One more cycle through this turns our tape into:

$$0 \flat 1 \flat 0 \flat 1 \underline{\flat} \flat \flat \flat \flat \ldots$$

Clearly, repeated cycles through this infinitely looping program will yield a tape consisting entirely of the infinite sequence 010101... with blank squares separating each square with a symbol printed on it:

$$0 \flat 1 \flat 0 \flat 1 \flat 0 \flat 1 \flat \ldots$$

Can this program be written differently?

> If (contrary to the description in §1) we allow the letters *L, R* to appear more than once in the operations column we can simplify the table considerably. (p. 234.)

In "the description in §1" (p. 231), Turing allowed the machine to "change the square which is being scanned, but only by shifting it *one* place to right or left" (my italics). Now, he is allowing the machine to move *more* than one place to the right or left; this is accomplished by allowing a *sequence* of moves. Here is the modified program:

m-config.	*symbol*	*operations*	*final m-config.*
	None	$P0$	b
b	0	$R, R, P1$	b
	1	$R, R, P0$	b

Note that there is only one *m*-configuration, that is, only one line number; another way to think about this is that this program has only one instruction. Turing would say that this machine never changes its state of mind. But that one instruction is, of course, more complex than the previous ones. This one is what would now be called a 'case' statement: In case there is no current symbol, print 0; in case the current symbol = 0, move right 2 squares and print 1; and in case the current symbol = 1, move right 2 squares and print 0.

I urge you to try to follow this version of the program, both for practice in reading such programs and to convince yourself that it has the same behavior as the first one. (Another interesting exercise is to write a program for a Turing machine that will print the sequence 010101... *without* intervening blank squares.)

So, our machine has "compute[d] the sequence 010101...". Or has it? It has certainly written that sequence down. Is that the same thing as "computing" it?

And here is another question: Earlier, I said that 010101... was the binary representation of $\frac{1}{3}$ and the decimal representation of $\frac{1}{99}$. Have we just computed $\frac{1}{3}$ in base 2? Or $\frac{1}{99}$ in base 10?[25] Even if you are inclined to answer 'yes' to the question in the previous paragraph, you might be more inclined to answer 'no' to the question in this one. Although Turing may have a convincing reason (in his §9) to say that computing consists of nothing more than writing down symbols, surely there has to be

---

[25]For an interesting discussion of this, see [Rescorla, 2013]. We'll return to some of the issues discussed in Rescorla's paper in Ch. 10.

more to it than that; surely, just writing down symbols is only *part* of computing. The other parts have to do with *which* symbols get written down, *in what order*, and *for what reason*. If I asked you to compute the decimal representation of $\frac{1}{99}$, how would you know that you were supposed to write down 010101...? Surely, *that* is the heart of computation. Or is it?

At this point, however, we should give Turing the benefit of the doubt. After all, *he* did not say that we were going to compute $\frac{1}{99}$, only that we were going to "compute" 010101..., and, after all, "computing" that sequence really just *is* writing it down; it's a trivial, or basic, or elementary, or primitive computation (choose your favorite adjective). Moreover, arguably, Turing showed us this trivial example so that we could clearly see the *format* of his Turing-machine programs before getting a more complex example.

Before turning to such a more complex program, let's consider the syntax of these programs a bit more. NEED TO EXPLAIN 'SYNTAX' SOMEWHERE. Each line of the program has the following, general form:

$$q_B \; S_I \; O \; M \; q_E$$

where:

1. q_B is an initial (or **b**eginning) *m*-configuration (a line number)

2. S_I is either an input that the machine reads (that is, the symbol on the currently scanned square), or else it is empty

3. O is an operation (or a sequence of operations) to be performed (where the operations are Px, E, L, R, and where x is any legally allowed symbol),[26] and

4. q_E is a final (or **e**nding) *m*-configuration.

 And the semantics (or interpretation) of this program line is:

if the Turing machine is in *m*-configuration q_B
and if either the current input = S_I **or** no input is specified
then begin
    do the sequence of operations O;

     {where each operation is either:
     print x on the current square,
     where x is a legally allowed symbol,
     or erase the symbol that is on the current square,
     where printing x overwrites whatever is currently printed
     on the square,
     and where erasing results in a blank square
     (even if the square is already blank)}

---

[26]In our first program, the only symbols were '0' and '1'; we will see others in subsequent examples.

> do the sequence of moves M;
>
> > {where each move is either:
> > move left one square,
> > or move right one square}
>
> go to m-configuration q_E
>
> **end**
>
> **else**
>
> > {if the machine is not in m-configuration q_I
> > or the current symbol $\neq$ S_I and is not blank, then:}
>
> do nothing (that is, halt)

8.10.2   Example II

We now come to "a slightly more difficult example":

> As a slightly more difficult example we can construct a machine to
> compute the sequence 001011011101111011111.... (p. 234.)

First, note that the sequence to be computed consists of the subsequences

$$0, \ 1, \ 11, \ 111, \ 1111, \ 11111, \ \ldots$$

That is, it is a sequence beginning with '0', followed by the numbers 1, 2, 3, 4, 5,...
written in base 1 (that is, as "tally strokes")—with each term separated by a '0'.

But this seems very disappointing! It seems that this "more difficult" computation
is still just writing down some symbols without "computing" anything. Perhaps. But
note that what is being written down (or "computed") here are the natural numbers.
This program will begin counting, starting with 0, then the successor of 0, the successor
of that, and so on. But, as we saw in §7.7.2, the successor function is one of the basic
recursive functions, that is, one of the basic computable functions.

Being able to (merely!) write down the *successor* of any number, being able to
(merely!) write down the *predecessor* of any non-0 number, and being able to *find*
a given term in a sequence of numbers are the only basic recursive (or computable)
functions. Turing's "slightly more difficult example" will show us how to compute the
first of these. Devising a Turing machine program for computing the predecessor of the
natural number n should simply require us to take a numeral represented as a sequence
of n occurrences of '1' and erase the last one. Devising a Turing machine program for
computing the jth term in a sequence of k symbols should simply require us to move a
certain number of squares in some direction to find the term (or, say, the first square of
a sequence of squares that represents the term, if the term is complex enough to have
to be represented by a sequence of squares).

And any other other recursive function can be constructed from these basic
functions by generalized composition (sequencing), conditional definition (selection),

and while-recursion (repetition), which are just "control structures" for how to find a path (so to speak) through a Turing-machine program—that is, ways to organize the sequence of *m*-configurations that the Turing machine should go through.

So, it looks as if **computation really is nothing more than writing things down, moving around (on a tape), and doing so in an order that will produce a desired result!**

Let's now look at this "slightly more difficult" program:

> The machine is to be capable of five *m*-configurations, viz., "𝔬", "𝔮", "𝔭", "𝔣", "𝔟" and of printing "ə", "*x*", "0", "1". (p. 234.)

I'll simplify these *m*-configurations to the more legible '*o*', '*q*', '*p*', '*f*', '*b*'. The first two symbols are going to be used only for bookkeeping purposes.[27] So, once again, Turing is really restricting himself to binary notation for the important information.

Continuing:

> The first three *symbols* on the tape will be "əə0"; the other *figures* follow on alternate squares. (p. 234, my italics.)

It may sound as if Turing is saying that the tape comes with some pre-printed information. But, when we see the program, we will see that, in fact, the first instruction has us print 'əə0' on the first three squares before beginning the "real" computation. Had the tape come with pre-printed information, perhaps it could have been considered as "innate" knowledge, though a less cognitive description could simply have been that the manufacturer of the tape had simplified our life knowing that the first thing that the program does to a completely blank tape is to print 'əə0' on the first three squares before beginning the 'real' computation. Because that only has to be done once, it might have been simpler to consider it as pre-printed on the tape.

Note that Turing calls these 'symbols' in the first clause, and then talks about 'figures' in the second clause. Figures, you may recall from §8.9.2, are the symbols '0' and '1'. So, Turing seems to be saying that all subsequent occurrences of '0' and '1' will occur on "alternate squares". What happens on the other squares? He tells us:

> On the intermediate squares we never print anything but "*x*". These letters serve to "keep the place" for us and are erased when we have finished with them. We also arrange that in the sequence of figures on alternate squares there shall be no blanks. (p. 234.)

So, it sounds as if the final tape will begin with 'əə0'; during the computation, subsequent squares will have '0' or '1' interspersed with '*x*'; and at the end of the computatation, those subsequent squares will only have '0' or '1', and no blanks. Of course, at the end, we could go back and erase the initial occurrences of 'ə', so that there would only be "figures" and no other symbols.

---

[27]The inverted 'e' is called a 'schwa'; it is used in phonetics to represent the sound "uh", as in 'but'. Turing uses it merely as a bookkeeping symbol with no meaning.

Here is the program:

Configuraton		Behaviour	
m-config.	*symbol*	*operations*	*final m-config.*
b		$P\partial, R, P\partial, R, P0, R, R, P0, L, L$	o
o	1	R, Px, L, L, L	o
	0		q
q	Any (0 or 1)	R, R	q
	None	$P1, L$	p
p	x	E, R	q
	∂	R	f
	None	L, L	p
f	Any	R, R	f
	None	$P0, L, L$	o

I think it will be helpful to restate this program in a more readable format:

b **begin**
    print '∂∂0' on the first 3 squares;
    P0 on the 5th square;
    move back to the 3rd square (which has '0' on it)
    go to line *o*
  end

o **if** current symbol = 1
    then begin move right; P*x*; move back 3 squares; stay in *o* **end**
    else if current symbol = 0
        then go to *q*

q **if** current symbol = 0 **or** current symbol = 1
    then begin move right 2 squares; stay in *q* **end**
    else if current square is blank
        then begin P1; move left; go to *p* **end**

p **if** current symbol = *x*
    then begin erase the *x*; move right; go to *q* **end**
    else if current symbol = ∂
        then begin move right; go to *f* **end**
    else if current square is blank
        then begin move left 2 squares; stay in *p* **end**

f **if** current square is not blank
    then begin move right 2 squares; stay in *f* **end**
    else begin P0; move left 2 squares; go to *o* **end**

Note that no line of the program ends with the machine changing its state of mind to *m*-configuration *b*. So that line of the program, which is the one that initializes the tape with 'əə0' on the first 3 squares, is only executed once. Note also that whenever an instruction ends with an command to go to that very same line, we are in a loop; a structured version of the program would use a **while...do** control structure.

There are some odd things to consider in lines *o*, *q*, *p*: What happens if the machine is in state *o* but the current symbol is not a "figure"? What happens in state *q* if the current symbol is 'ə' or '*x*'? And what happens in state *p* if the current symbol *is* a "figure"? Turing doesn't specify what should happen in these cases. One possibility is that he has already determined that none of these cases *could* occur. Still, modern software engineering practice would recommend that an error message be printed out in those cases. In general, in a computer program, when a situation occurs for which the program does not specify what should happen, anything is legally allowed to happen, and there is no way to predict what will happen; this is sometimes expressed in the slogan, "garbage in, garbage out".

Turing goes on "to illustrate the working of this machine" with "a table... of the first few complete configurations" (p. 234.) Recall that a "complete configuration" consists of information about which square is currently being scanned, the sequence of all symbols on the tape, and the line number of the instruction currently being executed. Rather than use Turing's format, I will continue to use the format that I used for Example I, adding the line number at the beginning, using underscoring to indicate the currently scanned square, and assuming that any squares not shown are blank; any blank square that is between two non-blank squares (if there are any) will be indicated by our symbol for a blank that has been made visible: ♭. You are urged to compare my trace of this program with Turing's.

So, we begin with a blank tape. What is the machine's initial state of mind, its initial *m*-configuration? Turing has forgotten to tell us! But it is fairly obvious that *b* is the initial *m*-configuration, and, presumably, we are scanning the leftmost square (or, if the tape is infinite in both directions, then we are scanning any arbitrary square), and, of course, all squares are blank:

$$b : \underline{\flat}, \flat, \ldots$$

The initial instruction tells us to print ə, move right, print another ə, move right again, print 0, move right 2 more squares, print another 0, move 2 squares back to the left, and go into state *o*. After doing this sequence of primitive operations, our complete configuration looks like this:

$$o : ə, ə, \underline{0}, \flat, 0, \flat, \ldots$$

(Just to help you in reading Turing's paper, my notation for the initial situation should be compared with his. Here is his:

$$:$$
$$b$$

He has a blank (literally), followed by a colon, with the *m*-configuration '*b*' underneath the blank, marking the currently scanned square.

Instead, I have '*b*:' preceding a sequence of (visible) blanks, the first one of which is marked as being the scanned square.

Turing then shows the second configuration:

$$\text{ə} \quad \text{ə} \quad 0 \qquad 0 \quad :$$
$$o$$

Turing has two occurrences of 'ə' followed by two '0's that are separated by an (invisible) blank, with the *m*-configuration '*o*' underneath the currently scanned square (which contains the first '0'), followed by a colon to mark the end of this complete configuration.

I have '*o*:' preceding a sequence consisting of the two occurrences of 'ə', followed by a '0' that is marked as being the scanned square, followed by a (visible) blank, followed by the second '0'.)

We are now in *m*-configuration *o*, and the currently scanned square contains '0', so the second case (that is, the bottom row) of this second instruction tells us merely to go into state *q*. (The "operations" column is left empty, so there is no operation to perform. It is worth noting that, although there does not always have to be an operation to perform, *there does always have to be a final state to go into, that is, a next instruction to perform.*) So, the tape looks exactly as it did before, except that the machine is now in state *q*:

$$q : \text{ə}, \text{ə}, \underline{0}, \flat, 0, \flat, \ldots$$

Because the machine is now in state *q* and still scanning a '0', the first case (that is, the top row) of this third instruction tells us to move two squares to the right but to stay in state *q*. So the tape now looks like this:

$$q : \text{ə}, \text{ə}, 0, \flat, \underline{0}, \flat, \ldots$$

Because the machine is still in state *q* and still scanning a '0' (although the currently scanned *square* is different), we perform the same (third) instruction, moving two more squares to the right and staying in state *q*:

$$q : \text{ə}, \text{ə}, 0, \flat, 0, \flat, \underline{\flat}, \ldots$$

The machine is still in state *q*, but now there is no scanned symbol, so the second case (bottom line) of the third instruction is executed, resulting in a '1' being printed on the current square, and the machine moves left, going into state *p*.

Whenever the machine is in state *p* and scanning a blank (as it is now), the third case (last line) of the fourth instruction is executed, so the machine moves two squares to the left and stays in state *p*:

$$p : \text{ə}, \text{ə}, 0, \underline{\flat}, 0, \flat, 1, \ldots$$

Now the machine is in state *p* scanning a blank, so the same instruction is executed: It moves two more squares to the left and continues in state *p*:

$$p : \text{ə}, \underline{\text{ə}}, 0, \flat, 0, \flat, 1, \ldots$$

But now it is the second case (middle line) of the fourth instruction that is executed, so the machine moves right and goes into state f:

$$f : ə, ə, \underline{0}, ♭, 0, ♭, 1, \ldots$$

When in state f scanning any symbol (but not a blank), the machine moves two squares to the right, staying in f:

$$f : ə, ə, 0, ♭, \underline{0}, ♭, 1, \ldots$$

Again, it moves two squares to the right, staying in f:

$$f : ə, ə, 0, ♭, 0, ♭, \underline{1}, \ldots$$

And again:

$$f : ə, ə, 0, ♭, 0, ♭, 1, ♭, \underline{♭}, \ldots$$

But now it executes the second case of the last instruction, printing '0', moving two squares to the left, and returning to state o:

$$o : ə, ə, 0, ♭, 0, ♭, \underline{1}, ♭, 0, \ldots$$

Now, for the first time, the machine executes the first case of the second instruction, moving right, printing 'x', moving three squares to the left, but staying in o:

$$o : ə, ə, 0, ♭, \underline{0}, ♭, 1, x, 0, \ldots$$

At this point, you will be forgiven if you have gotten lost in the "woods", having paid attention only to the individual "trees" and not seeing the bigger picture.[28] Recall that we are trying to count: to produce the sequence 0, 1, 11, 111,… with '0's between each term:

$$0\ 0\ 1\ 0\ 11\ 0\ 111\ 0\ldots$$

We started with a blank tape:

$$♭♭♭\ldots$$

and we now have a tape that looks like this:

$$əə0♭0♭1x0♭\ldots$$

Clearly, we are going to have to continue tracing the program before we can see the pattern that we are expecting; Turing, however, ends his tracing at this point. But we shall continue; however, I will only show the complete configurations without spelling out the instructions (doing that is left to the reader). Here goes, continuing from where

---

[28]My apologies for the mixed metaphor.

we left off:

o :	ə	ə	0	♭	<u>0</u>	♭	1	*x*	0	...			
q :	ə	ə	0	♭	<u>0</u>	♭	1	*x*	0	...			
q :	ə	ə	0	♭	0	♭	<u>1</u>	*x*	0	...			
q :	ə	ə	0	♭	0	♭	1	*x*	<u>0</u>	...			
q :	ə	ə	0	♭	0	♭	1	*x*	0	♭	<u>♭</u>	...	
p :	ə	ə	0	♭	0	♭	1	*x*	0	<u>♭</u>	1	...	
p :	ə	ə	0	♭	0	♭	1	<u>*x*</u>	0	♭	1	...	
q :	ə	ə	0	♭	0	♭	1	♭	<u>0</u>	♭	1	...	
q :	ə	ə	0	♭	0	♭	1	♭	0	♭	<u>1</u>	...	
q :	ə	ə	0	♭	0	♭	1	♭	0	♭	1	♭	<u>♭</u> ♭...
p :	ə	ə	0	♭	0	♭	1	♭	0	♭	1	<u>♭</u>	1 ♭...

Hopefully, now you can see the desired pattern beginning to emerge. The occurrences of '*x*' get erased, and what's left is the desired sequence, but with blank squares between each term *and* with two leading occurrences of 'ə'. You can see from the program that there is no instruction that will erase those 'ə's; the only instructions that pay any attention to a 'ə' are (1) the second case of *m*-configuration *p*, which only tells the machine to move right and to go into state *f*, and (2) the first case of *m*-configuration *f*, which, when scanning any symbol, simply moves two squares to the right (but, in fact, that configuration will never occur!).

Turing goes on to make some remarks about various notation conventions that he has adopted, but we will ignore these, because we are almost finished with our slow reading. I do want to point out some other highlights, however.

8.11   Section 4: "Abbreviated Tables"

In this section, Turing introduces some concepts that are central to programming and software engineering.

> There are certain types of process used by nearly all machines, and these, in some machines, are used in many connections. These processes include copying down sequences of symbols, comparing sequences, erasing all symbols of a given form, etc. (p. 235.)

In other words, certain sequences of instructions occur repeatedly in different programs and can be thought of as being single "processess": copying, comparing, erasing, etc.
    Turing continues:

> Where such processes are concerned we can abbreviate the tables for the *m*-configurations considerably by the use of "skeleton tables". (p. 235.)

The idea is that skeleton tables are descriptions of more complex sequences of instructions that are given a single name. This is the idea behind "subroutines" (or "named procedures") and "macros" in modern computer programming. If you have a sequence of instructions that accomplishes what might better be thought of as a single task (for example, copying a sequence of symbols), and if you have to repeat this sequence many times throughout the program, it is more convenient (for the human writer or reader of the program!) to write this sequence down only once, give it a name, and then refer to it by that name whenever it is needed.[29] There is one small complication: Each time that this named abbreviation is needed, it might require that parts of it refer to squares or symbols on the tape that will vary depending on the current configuration, so the one occurrence of this named sequence in the program might need to have variables in it:

> In skeleton tables there appear capital German letters and small Greek letters. These are of the nature of "variables". By replacing each capital German letter throughout by an *m*-configuration and each small Greek letter by a symbol, we obtain the table for an *m*-configuration. (pp. 235–236.)

Of course, whether one uses capital German letters, small Greek letters, or something more legible or easier to type is an unimportant, implementation detail. The important point is this:

> The skeleton tables are to be regarded as nothing but abbreviations: they are not essential. (p. 236.)

---

[29]"As Alfred North Whitehead wrote, 'Civilisation advances by extending the number of important operations which we can perform without thinking about them.' " [Hayes, 2014b, 22].

8.12 Section 5:
"Enumeration of Computable Sequences"

Another highlight of Turing's paper that is worth pointing out occurs in his §5: a way to convert every program for a Turing machine into a number. Let me be a bit more precise about this before seeing how Turing does it.

First, it is important to note that, for Turing, there really is no difference between one of his *a*-machines (that is, a Turing machine) and the *program for* it. Turing machines are "hardwired" to perform exactly one task, as specified in the program (the "table", or "machine table") for it. So, converting a program to a number is the same as converting a Turing machine to a number.

Second, "converting to a number"—that is, assigning a number to an object—really means that you are *counting*. So, in this section, Turing shows that you can count Turing machines by assigning a number to each one.

Third, if you can count Turing machines, then you can only have a countable number of them. But there are an *un*countable number of real numbers, so there will be some real numbers that are not computable!

Here is how Turing counts Turing machines. First (using the lower-case Greek letter "gamma", γ):

> A computable sequence γ is determined by a description of a machine which computes γ. Thus the sequence 001011011101111... is determined by the table on p. 234, and, in fact, any computable sequence is capable of being described in terms of such a table. (p. 239.)

"A description of a machine" is one of the tables such as those we have been looking at; that is, it is a computer program for a Turing machine.

But, as we have also seen, it is possible to write these tables in various ways. So, before we can count them, we need to make sure that we don't count any twice because we have confused two different ways of writing the same table with being two different tables. Consequently:

> It will be useful to put these tables into a kind of standard form. (p. 239.)

The first step in doing this is to be consistent about the number of separate operations that can appear in the "operations" column of one of these tables. Note that in the two programs we have looked at, we have seen examples in which there were as few as 0 operations and as many as 10 (not to mention the variations possible with skeleton tables). So:

> In the first place let us suppose that the table is given in the same form as the first table, for example, I on p. 233. [See our §8.10.1, above.] That is to say, that the entry in the operations column is always of one of the forms $E : E, R : E, L : Pa : Pa, R : Pa, L : R : L :$ or no entry at all. The table can always be put into this form by introducing more *m*-configurations. (p. 239, my interpolation.)

In other words, the operation in the operations column will be exactly one of:

    erase
    erase and then move right
    erase and then move left
    print symbol *a*

        (where '*a*' is a variable ranging over all the possible symbols in
        a given program)

    print *a* and then move right
    print *a* and then move left
    move right
    move left
    do nothing

"Introducing more *m*-configurations" merely means that a single instruction such as:

$$b \ \ 0 \ \ P1, R, P0, L \ \ f$$

can be replaced by two instructions:

$$
\begin{array}{cccc}
b & 0 & P1, R & f_1 \\
f_1 & & P0, L & f
\end{array}
$$

where 'f_1' is a new *m*-configuration not appearing in the original program. Put
otherwise, a single instruction consisting of a sequence of operations can be replaced
by a sequence of instructions each consisting of a single operation. (For convenience,
presumably, Turing allows *pairs* of operations, where the first member of the pair is
either *E* or *P* and the second is either *R* or *L*. So a single instruction consisting of a
sequence of (pairs of) operations can be replaced by a sequence of instructions each
consisting of a single operation or a single such pair.)

   Numbering begins as follows:

      Now let us give numbers to the *m*-configurations, calling them $q_1, \ldots, q_R$
      as in §1. The initial *m*-configuration is always to be called q_1. (p. 239.)

So, each *m*-configuration's number is written as a subscript on the letter '*q*'.
   The numbering continues:

      We also give numbers to the symbols $S_1, \ldots, S_m$ and, in particular, blank =
      $S_0, 0 = S_1, 1 = S_2$. (pp. 239–240.)

So, each symbol's number is written as a susbscript on the letter '*S*'.
   Note, finally, that Turing singles out three symbols for special treatment, namely,
'0', '1', and what I have been writing as ♭. (So, Turing is finally making the blank
visible.)
   At this point, we have the beginnings of our "standard forms", sometimes called
'normal' forms (which Turing labels N_1, N_2, N_3:

The lines of the table are now of [one of the following three] form[s] (p. 240):

m-config.	Symbol	Operations	Final m-config.	
q_i	S_j	PS_k, L	q_m	(N_1)
q_i	S_j	PS_k, R	q_m	(N_2)
q_i	S_j	PS_k	q_m	(N_3)

So, we have three "normal forms":

N_1 m-configuration $q_i =$ **if** currently scanned symbol is S_j,
       then begin print symbol S_k; move left; go to q_m **end**

N_2 m-configuration $q_i =$ **if** currently scanned symbol is S_j,
       then begin print symbol S_k; move right; go to q_m **end**

N_3 m-configuration $q_i =$ **if** currently scanned symbol is S_j,
       then begin print symbol S_k; go to q_m **end**

As Turing notes in the following passage (which I will not quote but merely summarize), erasing (E) is now going to be interpreted as printing a blank (PS_0), and a line in which the currently scanned symbol is S_j and the operation is merely to move right or left is now going to be interpreted as overprinting the very same symbol (PS_j) and then moving. So, all instructions require printing something: either a visible symbol or a blank symbol, and then either moving or not moving. As Turing notes:

> In this way we reduce each line of the table to a line of one of the forms $(N_1), (N_2), (N_3)$. (p. 240.)

Turing simplifies even further, eliminating the 'print' command and retaining only the symbol to be printed. After all, if all commands involve printing something, you don't need to write down 'P'; you only need to write down what you're printing. So each instruction can be simplified to a 5-tuple consisting of the initial m-configuration, the currently scanned symbol (and there will always be one, even if the "symbol" is blank, because the blank has been replaced by 'S_0'), the symbol to be printed (again, there will always be one, even if it's the blank), and the final m-configuration:

> From each line of form (N_1) let us form an expression $q_i S_j S_k L q_m$; from each line of form (N_2) we form an expression $q_i S_j S_k R q_m$; and from each line of form (N_3) we form an expression $q_i S_j S_k N q_m$. (p. 240.)

Presumably, N means something like "no move". A slightly more general interpretation is that, not only do we always print something (even if it's a blank), but we also always move somewhere, except that sometimes we "move" to our current location. This standardization is consistent with our earlier observation (in out §7.6.2, above) that the only two verbs that are needed are 'print(symbol)' and 'move(location)'.

     Next:

> Let us write down all expressions so formed from the table for the machine
> and separate them by semi-colons. In this way we obtain a complete
> description of the machine. (p. 240.)

Turing's point here is that the set of instructions can be replaced by a single string
of 5-tuples separated by semi-colons. There are two observations to make. First,
because the machine table is a *set* of instructions, there could (in principle) be several
different strings (that is, descriptions) for each such set, because strings are *sequences*
of symbols. Second, Turing has here introduced the now-standard notion of using a
semi-colon to separate lines of a program; however, this is not quite the same thing as
the convention of using a semi-colon to signal *sequencing*, because the instructions of a
Turing-machine program are not an ordered sequence of instructions (even if, whenever
they are written down, they have to be written down in some order).

So, Turing has developed a standard encoding of the lines of a program: an *m*-
configuration encoded as q_i (forget about b, f, etc.), a pair of symbols encoded as
S_j, S_k (the first being the scanned input, the second being the printed output; again,
forget about things like '0', '1', 'x', etc.), a symbol (either L, R, or N) encoding the
location to be moved to, and another *m*-configuration encoded as q_m. Next, he gives an
encoding of these standardized codes:

> In this description we shall replace q_i by the letter "*D*" followed by the
> letter "*A*" repeated i times, and S_j by "*D*" followed by "*C*" repeated j
> times. (p. 240.)

Before seeing *why* he does this, let's make sure we understand *what* he is doing. The
only allowable *m*-configuration symbols in an instruction are: $q_1, \ldots, q_l$, for some l
that is the number of the final instruction. What really matters is that each instruction
can be assumed to begin and end with an *m*-configuration symbol, and the only thing
that really matters is which one it is, which can be determined by the subscript on
q. In this new encoding, "*D*" simply marks the beginning of an item in the 5-tuple,
and the i occurrences of letter '*A*' encode the subscript. Similarly, the only allowable
symbols are: $S_1, \ldots, S_n$, for some n that is the number of the last symbol in the alphabet
of symbols. What really matters is that, in each instruction, the second and third
items in the 5-tuple can be assumed to be symbols (including a visible blank!), and
the only thing that really matters is which ones they are, which can be determined by
the subscript on *S*. In our new encoding, "*D*" again marks the beginning the next item
in the 5-tuple, and the j occurrences of '*C*' encode the subscript.

Turing then explains that:

> This new description of the machine may be called the *standard
> description* (S.D). It is made up entirely from the letters "*A*", "*C*", "*D*",
> "*L*", "*R*", "*N*", and from ";". (p. 240.)

So, for example, this 2-line program:

$$q_3 S_1 S_4 R q_5$$

$$q_5 S_4 S_0 L q_5$$

will be encoded by an S.D consisting of this 38-character string:

$$DAAADCDCCCCRDAAAAA; DAAAAADCCCCDLDAAAAA$$

The next step in numbering consists in replacing these symbols by numerals:

> If finally we replace "*A*" by "1", "*C*" by "2", "*D*" by "3", "*L*" by "4", "*R*" by "5", "*N*" by "6", ";" by "7" we shall have a description of the machine in the form of an arabic numeral. The integer represented by this numeral may be called a *description number* (D.N) of the machine. (p. 240; Turing did not capitalize 'arabic'.)

Just as Gödel numbering is a way to create a number corresponding to a string, so is this another way to do that. The D.N of the machine in our previous example is this numeral:

> 31113232222531111173111113222234311111

which, written in the usual notation with commas, is:

> 31,113,232,222,531,111,173,111,113,222,234,311,111

or, in words, 31 undecillion, 113 decillion, 232 nonillion, 222 octillion, 531 septillion, 111 sextillion, 173 quintillion, 111 quadrillion, 113 trillion, 222 billion, 234 million, 311 thousand, one hundred eleven. That is the "Turing number" of our 2-line program!

Turing observes that:

> The D.N determine the S.D and the structure of the machine uniquely. The machine whose D.N is *n* may be described as $\mathcal{M}(n)$. (pp. 240–242.)

Clearly, given a D.N, it is trivial to decode it back into an S.D in only one way. Equally clearly (and almost as trivially), the S.D can be decoded back into a program in only one way. Hence, "the structure of the machine" encoded by the D.N is "determine[d]...uniquely" by the D.N. However, because of the possibility of writing a program for a machine in different ways (permuting the order of the instructions), two different D.Ns might correspond to the same machine, so there will, in general be distinct numbers n, m (that is, $n \neq m$) such that $\mathcal{M}(n) = \mathcal{M}(m)$. That is, "the" Turing machine whose number = *n* might be the same machine as the one whose number = *m*; a given Turing machine might have two different numbers. Alternatively, we could consider that we have here two different machines that have exactly the same input-output behavior and that execute exactly the same algorithm. Even in that latter case, where we have more machines than in the former case, the machines are enumerable; that is, we can count them. Can we also count the sequences that they compute?

Yes; Turing explains why (with my comments interpolated in brackets):

> To each computable sequence [that is, to each sequence that is printed to the tape of a Turing machine] there corresponds at least one description number [we have just seen why there might be more than one], while to no description number does there correspond more than one computable

sequence [that is, each machine prints out exactly one sequence; there is
no way a given machine could print out two different sequences, because
the behavior of each machine is completely determined by its program,
and no program allows for any arbitrary, free, or random "choices" that
could vary what gets printed on the tape]. The computable sequences and
numbers [remember: every sequence corresponds to a unique number][30]
are therefore enumerable [that is, countable]. (p. 241.)

Next, on p. 241, Turing shows how to compute the D.N of program I (the one that
printed the sequence $\overline{01}$). And he gives a D.N without telling the reader what program
corresponds to it.

Finally, he alludes to the Halting Problem:

A number which is a description number of a circle-free machine will be
called a *satisfactory* number. In §8 it is shown that there can be no general
process for determining whether a given number is satisfactory or not.
(p. 241.)

A "satisfactory" number is the number of a circle-free Turing machine, that is, a Turing
machine that never halts and that does compute the infinite decimal representation of a
real number. That is, a "satisfactory" number is the number of a Turing machine for a
computable number. So, in Turing's §8, he is going to show that there is "no general
process"—that is, no Turing machine that can decide (by computing)—"whether a
given number is satisfactory", that is, whether a given number is the number of a
circle-free Turing machine. It is easy to determine if a given number is the number
of a Turing machine: Just decode it, and see if the result is a syntactically correct,
Turing-machine program. But, even if it is a syntactically correct, Turing-machine
program, there will be no way to decide (that is, to compute) whether it halts or not.
(Remember: For Turing, halting is bad, not halting is good; in modern presentations of
computing theory, halting is good, not halting is (generally considered to be)[31] bad.)

8.13   Section 6: "The Universal Computing Machine"

Although this section is at least one of, if not *the* most important section of Turing's
paper, we will only look at it briefly in this chapter. As before, you are encouraged to
consult [Petzold, 2008] for aid in reading it in detail.

Turing begins with this claim:

It is possible to invent a single machine which can be used to compute any
computable sequence. (p. 241.)

So, instead of needing as many Turing machines as there are computable numbers,
we only need one. Recall that our first "great insight" was that all information can be
represented using only '0' and '1' (§7.6.1, above). That means that all information that

---

[30]Although, because of a curiosity of decimal representation, some numbers correspond to more than one
sequence. The classic example is that $1 = 1.\overline{0} = 0.\overline{9}$.

[31]But see Ch. 11!

we would want to compute with can be represented by a sequence of '0's and '1's, that is, as a computable number (in binary notation). So, Turing's claim is that there is a single machine that can be used to compute anything that is computable.

Most of you own one. Indeed, most of you own several, some of which are small enough to be carried in your pocket! They are made by Apple, Dell, et al.; they come in the form of laptop computers, smartphones, etc. They are general-purpose, programmable computers.

> If this machine $\mathcal{U}$ is supplied with a tape on the beginning of which is written the S.D of some computing machine $\mathcal{M}$, then $\mathcal{U}$ will compute the same sequence as $\mathcal{M}$. (pp. 241–242.)

Your laptop or smartphone is one of these $\mathcal{U}$s. A program or "app" that you download to it is an S.D (written in a different programming language than Turing's) of a Turing machine that does only what that program or "app" does. The computer or smartphone that runs that program or "app", however, can also run other programs, in fact, many of them. That's what makes it "universal":

> But to do all the things a smartphone can do without buying one,... [a] consumer would need to buy the following:
> A cellphone....
> A mobile e-mail reader....
> A music player....
> A point-and-shoot camera....
> A camcorder....
> A GPS unit....
> A portable DVD player....
> A voice recorder....
> A watch....
> A calculator....
> Total cost: $1,999
> *In a smartphone, all those devices are reduced to software.*
> [Grobart, 2011]

How does Turing's universal computer work? Pretty much the same way that a modern computer works: A program (an "app") is stored somewhere in the computer's memory; similarly, the S.D of a Turing machine is written at the beginning of the universal machine's tape. The operating system of the computer fetches (that is, reads) an instruction and executes it (that is, "simulates its behavior" [Dewdney, 1989, 315]), then repeats this "fetch-execute" cycle until there is no next instruction; similarly, the single program for the universal machine fetches the first instruction on its tape, executes it, then repeats this cycle until there is no next instruction on its tape. The details of how it does that are fascinating, but beyond our present scope. (Again, [Petzold, 2008], as well as [Dewdney, 1989, Chs. 1, 28, 48], cover this in detail.)

8.14   The Rest of Turing's Paper

Sections 1–5 of Turing's paper cover the nature of computation, defining it precisely, and stating what is now called "Turing's (computability) thesis". Sections 6 and 7 of Turing's paper cover the universal machine. Section 8 covers the Halting Problem.

We have already examined Section 9 in detail; that was the section in which Turing analyzed how humans compute and then designed a computer program that would do the same thing. The branch of computer science that analyzes how humans perform a task and then designs computer programs to do the same thing is AI; so, in Section 9, Turing has developed the first AI program:

> Turing machines appear [in Turing's paper] as a result, a codification, of his analysis of calculations by humans. [Gandy, 1988, 82]

Turing was not unaware of this aspect of his work:

> One way of setting about our task of building a 'thinking machine' would be to take a man [sic] as a whole and to try to replace all the parts of him by machinery.   [Turing, 1948, 420] as cited in [Proudfoot and Copeland, 2012, 2]

But that's almost exactly what Turing's analysis of human computation in his 1936 paper does (at least in part): It takes a human's computational abilities and "replaces" them by (abstract) machinery.

Section 10 shows how it can be that many numbers that one might *think* are *not* computable are, in fact, computable.   Section 11 proves that Hilbert's *Entscheidungsproblem* "can have no solution" (p. 259). And the Appendix proves that Turing's notion of computation is logically equivalent to Church's.

Except for modern developments and some engineering-oriented aspects of computer science, one could create an undergraduate degree program in computer science based solely on this one paper that Turing wrote in 1936!

8.15   NOTES FOR NEXT DRAFT

1. The children's game of Candyland is like a TM: The (randomly shuffled) deck of color cards is like the TM table, telling us what to do. At each point, we can move a marker left or right. Some squares on the board, which constitutes the tape, have other instructions on them, but those could have been encoded in the cards. The game is completely deterministic, except that it doesn't necessarily "compute" anything of interest because of the random arrangement of the cards. Chess is also completely deterministic, but so "large" that we can't play it deterministically.

2. On "stored program" computer:

   (a) [Haigh, 2013] argues that the phrase 'stored-program computer' is ambiguous between several different readings and often conflated with the

notion of a universal TM. In my experience, the phrase refers to the idea that a computer's program can be stored in the computer itself (for example, on a TM's tape) and be changed, either by storing a different program or by modifying the program itself (perhaps while it is 'being executed, and perhaps being (self-)modified by the program itself). However, when I asked a colleague who first came up with the notion of "stored program" (fully expecting him to say either Turing or von Neumann), he replied—quite reasonably—"Jacquard".[32]

On this understanding, the phrase becomes key to understanding the difference between software and hardware (or programmed vs. hardwired computer)—see Ch. 13 for more on this—and it becomes a way of viewing the nature of the universal TM. But [Haigh, 2013] gives reasons to believe that there are so many other ways to understand the phrase that it would be better to refrain from using it.

(b) [Randell, 1994, 12], also discussing the controversy over what stored programming is—makes a statement that suggests that the main difference between programmed vs. hardwired computers might lie in the fact that "a program held on some read-only medium, such as switches, punched cards, or tape... was quite separate from the (writable) storage device used to hold the information that was being manipulated by the machine". "[S]toring the program within the computer, in a memory that could be read at electronic speeds during program execution" had certain "advantages" first noted by "the ENIAC/EDVAC team" (p. 13). Randell, however, thinks that the analogy with UTMs is more central (p. 13), and that "EDVAC does not qualify as a stored-program computer" because the "representations [of data and instructions] were quite distinct, and no means were provided for converting data items into instructions" or vice versa (p. 13)

(c) Computers, in any informal sense of the term (think laptop or even mainframe computer), are programmable. Turing machines are not! But *universal* TMs are!

The ability to store a program on a UTM's tape is what makes it programmable (that is, it can be changed from simulating the behavior of one TM to simulating the behavior of a different one).

But, logically, a program need not be stored physically in the computer: It could "control" the computer via a wireless connection from a different location. The ability to store a program *in* the computer *along with* the data allows for the program to change itself.

(d)   [A] Turing machine... was certainly thought of as being programmed in a 'hardwired' way. ... It is reasonable to view the universal Turing machine as being programmed by the description

---

[32]Stuart C. Shapiro, personal communication, 7 November 2013.

> of the machine it simulates; since this description is written on
> the memory tape of the universal machine, the latter is an abstract
> stored program computer. [Carpenter and Doran, 1977, 270]

First, note that TMs are hardwired. Second, note that what makes the UTM
"stored-program" is not so much that its program is not hardwired, but that
it is stored on its tape.

(e) See also [Robinson, 1994], cited in Further Sources.

(f) See also Vardi 1013 and Copeland 2013, cited in Ch. 6's Further Sources.

(g) And here is von Neumann on the concept:

> If the device [the "very high speed automatic digital computing
> sytem" (§1.0, p. 1) is to be *elastic*, that is as nearly as possible
> *all purpose*, then a distinction must be made between the specific
> instructions given for and defining a particular problem, and the
> general control organs which see to it that these instructions—
> no matter what they are—are carried out. The former must
> be **stored** in some way... the latter are represented by definite
> operating parts of the device. By the *central control* we mean this
> latter function only.... [von Neumann, 1945, §2.3, p. 2; italics in
> original, my boldface].

The "specific instructions" seems clearly to refer to a specific Turing
machine's program as encoded on the tape of a universal Turing machine.
The "central control" seems clearly to refer to the universal Turing
machine's program. So, if this is what is meant by "stored program", then
it pretty clearly refers to the way that a universal Turing machine works.

(h) The following comments might belong with the discussion of software and
hardware in Ch. 12 or of rule-following in Ch. 10:

There is a sense in which a stored-program computer does two things:
It executes a "hard-wired" fetch-execute cycle, and it executes whatever
software program is stored on its tape. Which is it "really" doing?
[Newell, 1980, 148] suggests that it is the former:

> A machine is defined to be a system that has a specific determined
> behavior as a function of its input. By definition, therefore, it
> is not possible for a single machine to obtain even *two* different
> behaviors, much less any behavior. The solution adopted is to
> decompose the input into two parts (or aspects): one part (the
> *instruction*) being takento determine which input-output function
> is to be exhibited by the second part (the *input-proper* along with
> the output.

I confess to not fully understanding this passage. But one way to
interpret it is to take the "decomposition" to refer to the hard-wired, fetch-
execute program of the univeral Turing machine, on the one hand, and
the software, stored program of the "virtual" machine, on the other hand.

The universal Turing machine thus ("consciously") executes the software program indirectly by ("unconsciously") directly executing its hard-wired program. It *simulates* a Turing machine that has that software program hard-wired as its machine table [Newell, 1980, 150].

(i) According to [Daylight, 2013, p. XX, n. 3], the phrase 'stored program' was not commonly used in the early 1950s. Furthermore, according to [Daylight, 2013, §3, p. VII], storing data and instructions together "was based on practical concerns, not theoretical reasoning' of the sort that might have been inspired by Turing's notion of a universal Turing machine.

3. The ability to erase has a downside: It destroys information, making it difficult, if not impossible, to *reverse* a computation. (See, for example, [Hayes, 2014b, 23].)

4. The idea that somehow printing out 010101... "computes" (say) 1/3 in base 2 is related to the idea that TM-computation is "automatic" or "mechanical". Consider any of the lengthy TM programs in his paper. Does a human following them understand what s/he is doing? This gives rise to Searle's Chinese Room Argument. Dennett has things to say about this, too, in his "Darwin's Strange Inversion" paper.

So does Moravec:

Several times during both matches [with Deep Blue], Kasparov reported signs of mind in the machine.

...In all other chess computers, he reports a mechanical predictability.... In Deep Blue, to his consternation, he saw instead an "alien intelligence."

...[T]he evidence for an intelligent mind lies in the machine's performance, not its makeup.

Now, the team that built Deep Blue claim no "intelligence" in it, only a large database of opening and end games, scoring and deepening functions tunes with consulting grandmasters, and, especially, raw speed that allows the machine to look ahead an average of fourteen half-moves per turn. ...

Engineers who know the mechanism of advanced robots most intimately will be the last to admit they have real minds. From the inside robots will indisputably be machines, acting according to mechanical principles, however elaborately layered. Only on the outside, where they can be appreciated as a whole, will the impression of intelligence emerge. A human brain, too, does not exhibit the intelligence under a neurobiologist's microscope that it does participating in a lively convesation. [Moravec, 1998, 10]

But this shows that there are two issues, both of which are consistent with the "strange inversion": First, Moravec's discussion, up to the last sentence, is clearly about external behavior independent of internal mechanism. In this

sense, it's consistent with the Turing test view of intelligence. Intelligence is computable, even if it isn't computed. Interestingly, in Deep Blue, it *is* computed, just not in the way that humans compute it or that other kinds of computers might compute it. But the last sentence points to the second interpretation, which is more consistent with the "strange inversion", namely, that, even if the internal mechanism is computing intelligent behavior in the way that humans do, looking at it at that level won't make that intelligence manifest. Intelligent behavior at the macro level can emerge from, or be implemented by, non-intelligent behavior at the micro-level. This is Dennett's point about the ever-smaller homunculi who bottom out in ones who can only say "yes" or "no".

5. A TM is to a UTM as a music box is to a player piano: TMs and music boxes can only play/execute the tune/program that is hardwired into them. UTMs and player pianos can play/execute any tune/program that is encoded on its tape/piano-roll. For discussion of this idea, see [Sloman, 2002].

    Here's a related question: "Why is a player piano *not* a computer?" [Kanat-Alexander, 2008]. Alternatively, when is a universal Turing machine a player piano? The "instructions" on the piano roll cause certain keys to be played; you can think of each key as a Turing-machine tape cell, with "play" or "don't play" analogous to "print-one" or "print-zero". One difference is that a player piano would be a *parallel* machine, because you can play chords.

6. [Sieg, 2006, 13] points out that Turing's analysis of (informal) computation abstracts away from arithmetical computations (such as addition, multiplication, etc.), in favor of purely symbolic "processes" (such as making a mark on paper, moving to a different location on the paper, etc.). Sieg says, "It is for this very reason that Turing most appropriately brings in human computers in a crucial way and exploits the limitations of their processing capacities, when proceeding mechanically." Note, too, that this is a move (that abstracts) away from semantically interpreted "processes" and towards purely (uninterpreted) syntactic ones.

    This is one of the reasons that people like Searle find it difficult to understand how a purely syntactic device (a computer) can produce semantic results (can do arithmetic, can understand—or, at least, process—natural language, etc.). We will return to this issue in Ch. 19. This "paradox" is discussed in [Dennett, 2009]. And it is related to issues about the goals of an algorithm (see §7.5) and the relation between the "internal" workings of an algorithm and its relationships to the "external" world (to be discussed in Chs. 10 and 17).

7. [Kleene, 1995, 19] observes that Turing's emphasis on not allowing "an infinity of symbols" that "differ... to an arbitrarily small extent" (§8.7.2.1, above) marks the distinction between "*digital* computation rather than *analog* computation".

8.   [T]he computer is [not] restricted to taking an ant's eye view of its work, squinting at the symbol on one square at a time. ...[T]he Turing-machine squares can corresond to whole sheets of paper. If we employ sheets ruled into 20 columns and 30 lines, and

authorize 99 primary symbols, there are $100^{600} = 10^{1200}$ possible square conditions, and we are at the opposite extreme. The schoolboy [sic] doing arithmetic on $8\frac{1}{2}$ by 12" sheets of ruled paper would never need, and could never utilize, all this variety.

Another representation of a Turing machine tape is as a stack of IBM cards, each card regarded as a single square for the machine. [Kleene, 1995, 26]

9.   A Turing machine is like an actual digital computing machine, except that (1) it is error free (i.e., it always does what its table says it should do), and (2) by its access to an unlimited tape it is unhampered by any bound on the quantity of its storage of information or "memory". [Kleene, 1995, 27]

The second limitation of a "real" (physical) Turing machine is not a very serious one, given (a) the option of always buying another square and (b) the fact that no computation could require an actual infinity of squares (else it would not be a finite computation). The more significant limitation is that some computations might require more squares than there could be in the universe (as is the case with NP computations such as chess).

The former limitation of real Turing machines does not obviate the need for program verification. Even an "ideal" Turing machine could be poorly programmed.

10. On Darwin's and Turing's strange inversion:

(a)   Francis Crick, in his Danz lectures *Of Molecules and Men*, discusses the problem of how life could have arisen:

[This] really is the major problem in biology. How did this complexity arise?

The great news is that we know the answer to this question, at least in outline. ... The answer was given over a hundred years ago by Charles Darwin.... Natural selection... provides an "automatic" mechanism by which a complex organism can survive and increase in both number and complexity.

For us in Cognitive Science, the major problem is how it is possible for mind to exist in this physical unverse. The great news... is that we know, at least in outline, how this might be. [Newell, 1980, 182]

According to Newell, the answer was given in 1936 by Alan Turing. Computation provides an automatic mechanism by which a machine (living or otherwise) can exhibit cognitive behavior.

(b) The inversion seems to concern the paradox that "intelligent" behavior can "emerge" from "unintelligent" or "mechanical" behavior. Simon says

some things that suggest that the paradox originates in an equivocation on 'mechanical': He says that both computers and brains are "mechanisms":

> If by a mechanism we mean a *system whose behavior at a point in time is determined by its current internal state combined with the influences that simultaneously impinge upon it from outside*, then any system that can be studied by the methods of science is a mechanism.
>
> But the term "mechanism" is also used in a narrower sense to refer to *systems that have the relatively fixed, routine, repetitive behavior of most of the machines we see around us.* [Simon, 1996a, 165]

Mechanisms in the latter sense do not exhibit self-generated "spontaneity", that is, "behavior that is unpredicted, perhaps even by the behaving system" [Simon, 1996a, 165]. This kind of spontaneity is exhibited by "intelligent" behavior. Turing's inversion concerns the fact that a computer can be a mechanism in the first sense without being one in the second sense. As Simon says:

> Clearly the computer occupies an ambiguous position here. Its behavior is more complex, by orders of magnitude, than any machine we have known; and not infrequently it surprises us, even when it is executing a program that we wrote. Yet, as the saying goes, "it only does what you program it to do" But truism though that saying appears to be, it is misleading on two counts. It is misleading, first, because it is often interpreted to mean: "It only does wht you believe you programmed it to do," which is distinctly not the case.
>
> More serious, it is misleading because it begs the question of whether computers and people are different. They are different (on this dimension) only if people behave differently from the way they are programmed to behave. But if we include in "program" the whole state of human memory, then to assert that people "don't do only what they are programmed to do" is equivalent to asserting that people's brains are not mechanisms, hence not explainable by the methods of science. [Simon, 1996a, 165]

Simon's point is that people are no more spontaneous than computers and that computers are no less mechanistic than people. (For similar comments, see [Simon, 1996a, 14–17].)

11.   ... Turing asked in the historical context in which he found himself *the* pertinent question, namely, what are the possible processes **a human being** can carry out (when computing a number or, equivalently, determining algorithmically the value of a number theoretic function)? [Sieg, 2000, 6; original italics, my boldface]

12. On TM as first AI program:

The intent [of a first computer science course should be] to reveal, through... examples, how analysis of some intuitively performed human tasks leads to mechanical algorithms accomplishable by a machine. [Perlis, 1962, 189]

13. That an algorithm should "demand no insight or ingenuity on the part of the human being carrying it out" [Copeland, 2000] has "the implication that the human is used as a biological computing machine" [PolR, 2009].

First, although something has to carry the algorithm out, it need not be a human being (though, of course, in Turing's analysis, it is, or, at least, it is an idealized human). So, more generally, the implication is that the interpreter/executor is the computing machine. So far, so good.

Second, [PolR, 2009]'s point is that "the human must make no judgment call and use no skill beyond his ability to apply the rules with pencil and paper". Clearly, if the executor is not human, it won't be able to make such judgment calls or use any such skill. Only if the executor is human does this become a worry. This seems to be what lies behind Searle's Chinese Room Argument, to be discussed in Ch. 19. It is also what underlies Turing's "strange inversion", namely, that no such judgment calls or skills—or even biology—are needed at all.

14. The halting theorem is often attributed to Turing in his 1936 paper. In fact, Turing did not discuss the halting problem, which was introduced by Martin Davis in about 1952. [Copeland and Proudfoot, 2010, 248, col. 2]

The authors don't elaborate on this. Perhaps it has something to do with the difference between halting and circle-free?

15. More on Turing's inversion (and magic):

'A telegraph!' Mme de Villefort repeated.

'Yes, indeed, the very thing: a telegraph. Often, at the end of a road, on a hilltop, in the sunshine, I have seen those folding black arms extended like the legs of some giant beetle, and I promise you, never have I contemplated them without emotion, thinking that those bizarre signals so accurately travelling through the air, carrying the unknown wishes of a man sitting behind one table to another man sitting at the far end of the line behind another table, three hundred leagues away, were written against the greyness of the clouds or the blue of the sky by the sole will of that all-powerful master. Then I have thought of... occult forces, and laughed. Never did I wish to go over and examine these great insects with their white bellies and slender black legs, because I was afraid that under their stone wings I would find the human genie, cramped, pedantic, stuffed with arcane science and sorcery. Then, one fine morning, I discovered that the motor that drives every telegraph is a poor devil of a clerk who earns twelve hundred francs a year and... spend sthe whole day staring at

the insect with the white belly and black legs that corresponds to his own and is sited some four or five leagues away. At this, I became curious to study this living chrysalis from close up and to watch the dumbshow that it offers from the bottom of its shell to that other chrysalis, by pulling bits of string one after the other.'

[At the telegraph "belonging to the Ministry of the Interior"] 'I should find people who would try to force me to understand things of which I would prefer to remain ignorant.... ... I want to keep my illusions about insects....' [Dumas, 1844, Ch. 40, pp. 676–677]

16. Related to the previous quote, and perhaps also to the Chinese Room Argument:

'Did Monsieur come to look at the telgraph?'

'Yes, provided the rules do not forbid it.'

'Not at all,' said [the telegraph operator]. 'There is not danger, because no one knows or can know what we are saying.'

'Yes, indeed,' said the count [of Monte Cristo]. 'I have even been told that you repeat signals that you do not understand yourselves.'

'Indeed we do, Monsieur, and I much prefer that,' the telegraph man said, laughing.

'Why do you prefer it?'

'Because in that way I have no responsibility. I am a machine and nothing more. A long as I work, no one asks anything more from me.' [Dumas, 1844, Ch. 41, p. 681]

17. Need to say something about *The Imitation Game* movie, including a list of the other films/plays about Turing. Also cite the websites that serve as background/critique for *The Imitation Game*, including:

   (a) "The Imitation Game: The Philosophical Legacy of Alan Turing", *The Critique* (31 December 2014)
   http://www.thecritique.com/news/the-imitation-game-the-philosophical-legacy-of-alan-turing/
   (accessed 19 January 2015).

      • Separately highlight some of the articles.

   (b) Anderson, L.V. (2014), "How Accurate Is *The Imitation Game*?", *Slate* (3 December),
   http://www.slate.com/blogs/browbeat/2014/12/03/the_imitation_game_fact_vs_fiction_how_true_the_new_movie_is_to_alan_turing
   (accessed 19 January 2015).

   (c) Caryl, Christian (2015), "Saving Alan Turing from His Friends", *New York Review of Books*, (5 February): 19–21,
   http://www.nybooks.com/articles/archives/2015/feb/05/saving-alan-turing-his-friends/
   (accessed 13 February 2015).

   (d) von Tunzelmann, Alex (2014), "The Imitation Game: Inventing a New Slander to Insult Alan Turing" (20 November),

http://www.theguardian.com/film/2014/nov/20/the-imitation-game-
invents-new-slander-to-insult-alan-turing-reel-history
(accessed 19 January 2015).

(e) Woit, Peter (2014), "The Imitation Game,", *Not Even Wrong*,
http://www.math.columbia.edu/∼woit/wordpress/?p=7365
(accessed 19 January 2015).

18. Here are some of the other films:

(a) *Breaking the Code* (1986),
http://www.imdb.com/title/tt0115749/,
http://en.wikipedia.org/wiki/Breaking_the_Code

(b) *Enigma* (2001),
http://www.imdb.com/title/tt0157583/,
http://en.wikipedia.org/wiki/Enigma_(2001_film)

- This fictionalized version omits Turing!

(c) *Codebreaker: The Story of Alan Turing* (2011),
http://www.imdb.com/title/tt2119396/,
http://en.wikipedia.org/wiki/Codebreaker_(film)

- Half dramatization, half documentary, with interviews of people who knew Turing (including some who are fictionalized in *The Imitation Game*).

19. A man provided with paper, pencil, and rubber [eraser], and subject to strict discipline, is in effect a universal machine. [Turing, 1948, 416]

20. A Turing machine computing a partial function was called "circular".... For some inputs, circular machines did not generate outputs of the desired type. [Piccinini, 2003, 42n18]

Check this against my interpretation of 'circular', above. Note that generating output (for all inputs) makes a function total, but, presumably, generating output that is "not...of the desired type" is the equivalent of generating "garbage" or having unpredictable behavior, that is, of being undefined.

21. The shortest description of Turing's accomplishment?:

[Turing showed that any computation can be described in terms of a machine shifting among a finite numbe of states in response to a sequence of symbols read and written one at a time on a potentially infinite tape. [Mahoney and Haigh (ed.), 2011, 79]

Compare my "theses" and both [Thomason, 2003, 328] and [Lamport, 2011, 6] cited in §9.4.

8.16　Further Sources of Information

1. Books and Essays:

   - American Mathematical Society (2006), Special issue on Alan Turing, *Notices of the AMS* 53(10) (November).

     – Table of contents and articles accessed 30 Mauy 2014 from:
       http://www.ams.org/notices/200610/

     – Includes articles by Andrew Hodges, Solomon Feferman, S. Barry Cooper, and Martin Davis, among others.

   - Bullynck, Maarten; Daylight, Edgar G.; & De Mol, Liesbeth (2015), "Why Did Computer Science Make a Hero out of Turing?", *Communications of the ACM* 58(3) (March): 37–39,
     http://cacm.acm.org/magazines/2015/3/183592-why-did-computer-science-make-a-hero-out-of-turing/fulltext
     (accessed 24 March 2015)

   - Carpenter, B.E.; & Doran, R.W. (1975?), "The Other Turing Machine", *The Computer Journal* 20(3): 269–279.

   - Cooper, S. Barry (2012a), "Incomputability after Alan Turing", *Notices of the AMS* 59(6): 776–784.

     – Preprint accessed 26 February 2014 from:
       http://arxiv.org/pdf/1304.6363.pdf

     – "Is there a causal dimension that is not algorithmic? Does it *matter* if there is?" (p. 776)

   - Cooper, S. Barry (2012b), "Turing's Titanic Machine?", *Communications of the ACM* 55(3) (March): 74–83.

     – Accessed 26 February 2014 from:
       http://cacm.acm.org/magazines/2012/3/146259-turings-titanic-machine/fulltext

     – "[E]xamine[s] challenges to...[the] continued primacy [of universal Turing machines] as a model for computation in daily practice and in the wider universe."

   - Cooper, S. Barry; & van Leeuwen, Jan (2013), *Alan Turing: His Work and Impact* (Amsterdam: Elsevier).

     – A large, "coffee table"-sized anthology of Turing's writings with commentaries by, among many others, Rodney Brooks, Gregory Chaitin, B. Jack Copeland, Martin Davis, Daniel Dennett, Luciano Floridi, Lance Fortnow, Douglas Hofstadter, Wilfried Sieg, Aaron Sloman, Robert I. Soare, and Stephen Wolfram.

     – Reviewed in:
       Avigad, Jeremy (2014), "Alan Turing: His Work and Impact", *Notices of the American Mathematical Society* 61(8) (September): 886–890,
       http://www.ams.org/notices/201408/201408-full-issue.pdf　　　　(accessed 24 March 2015).

   - Copeland, B. Jack (1997), "The Broad Conception of Computation", *American Behavioral Scientist* 40(6) (May): 690–716.

- – An essay on hypercomputation (or "non-classical" computation), but the introductory section (pp. 690–698) contains an enlightening discussion of the scope and limitations of Turing's accomplishments.

- Copeland, B. Jack (ed.) (2004), *The Essential Turing* (Oxford: Oxford University Press).

  - – An anthology of Turing's papers.
  - – Critically reviewed in:
    Hodges, Andrew (2006), Book Review of *The Essential Turing*, in *Notices of the AMS* 53(10) (November): 1190–1199; accessed 13 June 2014 from:
    http://www.ams.org/notices/200610/rev-hodges.pdf

- Copeland, B. Jack; & Shagrir, Oron (2013), "Turing versus Gödel on Computability and the Mind", in B. Jack Copeland, Carl J. Posy, & Oron Shagrir (eds.), *Computability: Turing, Gödel, Church, and Beyond* (Cambridge, MA: MIT Press): 1–33.

  - – Explores both Gödel's interpretations of Turing's work and the relation of the human mind to Turing machines.

- Curtis, M.W. (1965), "A Turing Machine Simulator", *Journal of the ACM* 12(1) (January): 1–13.

  - – Contains a program written for the IBM 1620 designed for education purposes.

- **Davis, Martin (2004), "The Myth of Hypercomputation", in C. Teuscher (ed.), *Alan Turing: The Life and Legacy of a Great Thinker* (Berlin: Springer): 195–212.**

  - – Accessed 15 May 2014 from:
    http://www1.maths.leeds.ac.uk/∼pmt6sbc/docs/davis.myth.pdf
  - – Contains a good discussion of Turing's role in the history of computation.

- Daylight, Edgar G. (2014), "A Turing Tale", *Communications of the ACM* 57(10) (October): 36–38,
  http://cacm.acm.org/magazines/2014/10/178787-a-turing-tale/fulltext
  (accessed 5 December 2014)

  - – "Assessing the accuracy of popular descriptions of Alan Turing's influences and legacy" (Abstract, p. 36)

- **Dewdney, A.K.:**

  - – **Ch. 1: "Algorithms: Cooking Up Programs"**
  - – **Ch. 28: "Turing Machines: The Simplest Computers"**
  - – **Ch. 48: "Universal Turing Machines: Computers as Programs"**

  from: Dewdney, A.K. (1989), *The Turing Omnibus: 61 Excursions in Computer Science* **(Rockville, MD: Computer Science Press).**

- Fitzsimmons, Emma G. (2013), "Alan Turing, Enigma Code-Breaker and Computer Pioneer, Wins Royal Pardon", *New York Times* (24 December).

  - – Accessed 18 February 2014 from:
    http://www.nytimes.com/2013/12/24/world/europe/alan-turing-enigma-code-breaker-and-computer-pioneer-wins-royal-pardon.html

- Gandy, Robin (1980), "Church's Thesis and Principles for Mechanisms", in Jon Barwise, H.J. Keisler, & K. Kunen (eds.), *The Kleene Symposium* (Amsterdam: North-Holland): 123–148.

  – Gandy, who was Turing's only Ph.D. student, argues "that Turing's analysis of computation by a human being does not apply directly to mechanical devices."

  – Sieg, Wilfried, & Byrnes, John (1999), "An Abstract Model for Parallel Computation: Gandy's Thesis", *The Monist* 82(1) (January): 150–164.

    ∗ A follow-up article that simplifies and generalizes Gandy's paper.

  – Israel, David (2002), "Reflections on Gödel's and Gandy's Reflections on Turing's Thesis", *Minds and Machines* 12(2) (May): 181–201.

  – Shagrir, Oron (2002), "Effective Computation by Humans and Machines", *Minds and Machines* 12(2) (May): 221–240.

- Haigh, Thomas (2014), "Actually, Turing Did Not Invent the Computer", *Communications of the ACM* 57(1) (January): 36–41.

  – Discusses the role of the Turing machine in the history of computers, minimizing its historical importance to the actual development of computing machines.

- Hayes, Brian (2007), "Trains of Thought", *American Scientist* 95(2) (March-April): 108ff.

  – Accessed 21 February 2014 from:
    http://www.americanscientist.org/issues/pub/trains-of-thought

  – An implementation of a Turing machine using railroad trains. See also Stewart 1994, below.

- **Hodges, Andrew (2012), *Alan Turing: The Enigma; Centenary Edition* (Princeton, NJ: Princeton University Press).**

  – Reviews of first edition of Hodges's biography:

  (a) Bernstein, Jeremy (1986), "A Portrait of Alan Turing", *New Yorker* (20 January): 78, 81–87.

    ∗ Accessed 18 February 2014 from:
      http://www.cdpa.co.uk/Newman/MHAN/view-item.php?Box=5&Folder=6&Item=5&Page=1

  (b) Hofstadter, Douglas B. (1983), "Mind, Body and Machine", *New York Times Book Review* (13 November).

  (c) Toulmin, Stephen (1984), "Fall of a Genius", *New York Review of Books* (19 January): 3–4, 6.

    ∗ Accessed 18 February 2014 from:
      http://www.nybooks.com/articles/archives/1984/jan/19/fall-of-a-genius/

    ∗ See also:
      Stern, Howard; & Daston, Lorraine (1984), "Turing & the System" (letter to the editor, with reply by Toulmin), *New York Review of Books* (26 April): 52–53; accessed 18 February 2014 from:
      http://www.nybooks.com/articles/archives/1984/apr/26/turing-the-system/

- See also Hodges's "Alan Turing: The Enigma" website, accessed 18 February 2014 from:
  http://www.turing.org.uk/index.html

- Copeland, B. Jack; & Proudfoot, Diane (1996), "On Alan Turing's Anticipation of Connectionism", *Synthese* 108: 361–377.

  - "...Turing was probably the first person to consider building computing machines out of simple, neuron-like elements connected together into networks in a largely random manner. ...By the application of what he described as 'appropriate interference, mimicking education'...[such a] machine can be trained to perform any task that a Turing machine can carry out...." (p. 361.)

  - For more information on this, see:
    http://tinyurl.com/CopelandProudfoot1996[33] (accessed 21 February 2014).

- Feferman, Solomon (2006), "Turing's Thesis", *Notices of the AMS* 53(10): 1200–1206.

  - An overview of Turing's work focusing on his PhD thesis (not the Church-Turing "thesis") that introduced "oracle" machines.

- Kleene, Stephen C. (1988), "Turing's Analysis of Computability and Major Applications of It", in Rolf Herken (ed.), *Universal Turing Machine: A Half-Century Survey; Second Edition* (Vienna: Springer-Verlag, 1995): 15–49.

- Leavitt, David (2005), *The Man Who Knew Too Much: Alan Turing and the Invention of the Computer* (W.W. Norton).

  - Another biography of Turing, reviewed by S. Barry Cooper in *Notices of the AMS* 53(10) (2006): 1213–1217; accessed 13 June 2014 from:
    http://www.ams.org/notices/200610/rev-cooper.pdf

- Leiber, Justin (2006), "Turing's Golden: How Well Turing's Work Stands Today", *Philosophical Psychology* 19(1): 13–46.

- Michie, Donald (2008), "Alan Turing's Mind Machines", in Philip Husbands, Owen Holland, & Michael Wheeler (eds.), *The Mechanical Mind in History* (Cambridge, MA: MIT Press), Ch. 4, pp. 61–74.

- Muggleton, S. (1994), "Logic and Learning: Turing's Legacy", in K. Furukawa, D. Michie, & S. Muggleton (eds.), *Machine Intelligence 13: Machine Intelligence and Inductive Learning* (Oxford: Clarendon Press): 37–56.

  - "[I]n this paper it is shown that there is a direct evolution in Turing's ideas from his earlier investigations of computability to his later interests in machine intelligence and machine learning."

- Myhill, John (1952), "Some Philosophical Implications of Mathematical Logic: I. Three Classes of Ideas", *Review of Metaphysics* 6(2) (December): 165–198.

  - Contains some observations on what it means for a problem to be "solvable".

- Myhill, John (1972), "What Is a Real Number?", *American Mathematical Monthly* 79(7) (August-September): 748–754.

---

[33]http://www.alanturing.net/turing_archive/pages/reference%20articles/connectionism/Turing'santicipation.html

– Not directly related to Turing, but "consider[s] the notion of real numbers from a constructive point of view. The point of view requires that any real number can be *calculated*" (p. 748), that is, computed, which is (in part) what Turing's 1936 paper is about.

• **Petzold, Charles (2008),** *The Annotated Turing: A Guided Tour through Alan Turing's Historic Paper on Computability and the Turing Machine* **(Indianpolis: Wiley).**

   – Davis, Martin (2008), "Touring Turing" (review of Petzold 2008), *American Scientist* 96(6) (November-December): 520.

   – Also see websites below.

• Piccinini, Gualtiero (2003), "Alan Turing and the Mathematical Objection", *Minds and Machines* 13: 23–48,
http://www.umsl.edu/∼piccininig/Alan_Turing_and_Mathematical_Objection.pdf
(accessed 13 February 2015).

   – Primarily about Turing's views on AI (see Ch. 19), but also discusses his theory of computation and the role of "oracle" machines.

• Rapaport, William J. (1985), "Turing Machines", Appendix B of Morton L. Schagrin, Randall R. Dipert, & William J. Rapaport, *Logic: A Computer Approach* (New York: McGraw-Hill): 327–339.

   – Accessed 7 November 2012 from:
http://www.cse.buffalo.edu/∼rapaport/Papers/schagrinetal85-TuringMachines.pdf

   – A chapter on Turing machines from an introductory logic textbook; see also Tymoczko & Goodhart 1986 and Lodder's website, below.

• Rendell, Paul (2000), "This Is a Turing Machine Implemented in Conway's Game of Life", accessed 21 February 2014 from:
http://rendell-attic.org/gol/tm.htm

   – See also:
Rendell, Paul (2001), "A Turing Machine in Conway's Game Life", accessed 21 February 2014 from:
https://www.ics.uci.edu/∼welling/teaching/271fall09/Turing-Machine-Life.pdf

• Rendell, Paul (2010), "This Is a Universal Turing Machine (UTM) Implemented in Conway's Game of Life", accessed 21 February 2014 from:
http://rendell-attic.org/gol/utm/

• Rey, Georges (2012), "The Turing Thesis vs. the Turing Test", *The Philosopher's Magazine* 57 (2nd Quarter): 84–89.

• Robinson, J.A. (1994), "Logic, Computers, Turing, and von Neumann", in K. Furukawa, D. Michie, & S. Muggleton (eds.), *Machine Intelligence 13: Machine Intelligence and Inductive Learning* (Oxford: Clarendon Press): 1–35.

   – Accessed 19 February 2014 from:
http://staffweb.worc.ac.uk/DrC/Courses%202013-14/COMP3202/Reading%20Materials/robinson94.pdf

   – Reminiscences by one of the leading figures in computation theory.

- Rothstein, Edward (2014), "Turing's Spirit Hovers at a Restored Estate", *New York Times* (29 November): C1, C7;
  http://www.nytimes.com/2014/11/29/arts/design/where-the-real-imitation-game-happened.html
  (accessed 25 December 2014)

  – A review of the Bletchley Park museum.

- Shagrir, Oron (2006), "Gödel on Turing on Computability", in Adam Olszewski, Jan Wołenski, & Robert Janusz (eds.), *Church's Thesis after 70 Years* (Ontos-Verlag): 393–419;
  http://edelstein.huji.ac.il/staff/shagrir/papers/Goedel_on_Turing_on_Computability.pdf
  (accessed 11 December 2013).

  – Explores why Gödel believed both that "the correct definition of mechanical computabilty was established beyond any doubt by Turing" [Gödel, 930s, 168] and that "this definition... rest[s] on the dubious assumption that there is a finite number of states of mind" [Shagrir, 2006a, §1].

- Sieg, Wilfried (1994), "Mechanical Procedures and Mathematical Experience", in George Alexander (ed.), *Mathematics and Mind* (New York: Oxford University Press): 71–117.

  – Accessed 10 December 2013 from:
    http://repository.cmu.edu/cgi/viewcontent.cgi?article=1248&context=philosophy
  – §3 contains a detailed analysis of Turing's work.

- Sieg, Wilfried (1997), "Step by Recursive Step: Church's Analysis of Effective Calculability", *Bulletin of Symbolic Logic* 3(2) (June): 154–180.

  – "...Turing's analysis is neither concerned with *machine computations* nor with general *human mental processes*. Rather, it is *human mechanical computability* that is being analyzed..." (p. 171).

- Smith, Alvy Ray (2014), "His Just Deserts: A Review of Four Books", *Notices of the American Mathematical Society* 61(8) (September): 891–895,
  http://www.ams.org/notices/201408/201408-full-issue.pdf (accessed 24 March 2015).

  – Reviews four biographies of Turing.

- **Soare, Robert Irving (2013), "Turing and the Art of Classical Computability", in S. Barry Cooper & Jan van Leeuwen (eds.), *Alan Turing: His Work and Impact* (Elsevier): 65–70.**

  – Preprint accessed 30 April 2014 on complementary, illustrated website, "The Art of Classical Computability",
    http://www.people.cs.uchicago.edu/∼soare/Art/

- Stewart, Ian (1994), "A Subway Named Turing", *Scientific American* (September): 104, 106–107.

  – An implementation of a Turing machine using subway trains. See also Hayes 2007, above.

- Tymoczko, Thomas; & Goodhart, Sarah (1986), "From Logic to Computers: A Suggestion for Logic Teachers", *Teaching Philosophy* 9(1) (March): 15–33.

- **–** Advocates teaching Turing machines in introductory logic courses; see also
  Rapaport 1985, above, and Lodder's website, below.

- Vardi, Moshe Y. (2013), "Who Begat Computing?", *Communications of the ACM*
  56(1) (January): 5; accessed 13 November 2013 from
  http://cacm.acm.org/magazines/2013/1/158780-who-begat-computing/fulltext

  - **–** For a reply, see:
    Copeland, Jack (2013), "What Apple and Microsoft Owe to Turing",
    Huff[ington] Post Tech/The Blog (12 August), accessed 13 November 2013
    from
    http://www.huffingtonpost.com/jack-copeland/what-apple-and-microsoft-
    _b_374211 4.html

- Vogel, Gretchen (2012), "Turing Pattern Fingered for Digit Formation", *Science*
  338 (14 December): 1406.

  - **–** On Turing's biological theory of morphogenesis.

- Wang, Hao (1957), "A Variant to Turing's Theory of Computing Machines",
  Journal of the ACM 4(1) (January): 63–92.

  - **–** "[T]his paper...offer[s] a theory which is closely related to Turing's but is
    more economical in the basic operations....[A] theoretically simple basic
    machine can be...specified such that all partial recursive functions (and hence
    all solvable computation problems) can be computed by it and that only four
    basic types of instruction are employed for the programs: shift left one space,
    shift right one space, mark a blank space, conditional transfer....[E]rasing is
    dispensable, one symbol for marking is sufficient, and one kind of transfer is
    enough. The reduction is...similar to...the definability of conjunction and
    implication in terms of negation and disjunction...." (p. 63.)

- **Weizenbaum, Joseph (1976),** *Computer Power and Human Reason* **(New York:
  W.H. Freeman).**

  - **–** Ch. 2 ("Where the Power of the Computer Comes From") contains a masterful
    presentation of a Turing Machine implemented with pebbles and toilet paper!
  - **–** This is also an excellent book on the role of computers in society, by the creator
    of the "Eliza" AI program.

- Whitemore, Hugh (1966), *Breaking the Code* (Samuel French Inc., 2010).

  - **–** A play, later a film. For more information, see:
    http://en.wikipedia.org/wiki/Breaking_the_Code (accessed 18 February 2014).
  - **–** The film was accessed 18 February 2014 from:
    http://www.youtube.com/watch?v=S23yie-779k
  - **–** An excerpt of the play was published as:
    Whitemore, Hugh (1988), "The Enigma: Alan Turing Confronts a Question
    of Right and Wrong", *The Sciences* 28(2) (March/April): 40–41.

2. Websites:

   (a) *The Annotated Turing* websites:

   - http://www.theannotatedturing.com/ (accessed 6 November 2012)
   - http://www.charlespetzold.com/AnnotatedTuring/ (accessed 6 November
     2012)

- http://tinyurl.com/AnnotatedTuring (accessed 19 January 2014)[34]

(b) Copeland, Jack (2012), "Is Alan Turing Both Inventor of the Basic Ideas of the Modern Computer and a Pioneer of Artificial Intelligence?", *Big Questions Online* (12 November),
https://www.bigquestionsonline.com/content/alan-turing-both-inventor-basic-ideas-modern-computer-and-pioneer-artificial-intelligence
(accessed 26 February 2014).

(c) Copeland, B. Jack, "AlanTuring.net Reference Articles on Turing", accessed 19 January 2014 from[35]
http://tinyurl.com/lkgqsch

- Website that contains a variety of interesting papers on various aspects of Turing's work, most written by Copeland, a well-respected contemporary philosopher, including:
  - Copeland, Jack (2000), "The Church-Turing Thesis", accessed 19 January 2014 from[36]
    http://tinyurl.com/mtch7f6
  - Copeland, Jack (2000), "What Is a Turing Machine?", accessed 19 January 2014 from[37]
    http://tinyurl.com/bddfe

(d) Daylight, Edgar G. (2013), "Towards a Historical Notion of 'Turing—the Father of Computer Science' "
http://www.dijkstrascry.com/sites/default/files/papers/Daylightpaper91.pdf
(accessed 7 April 2015).

- Argues from historical evidence that the reasons that Turing is considered "the father of computer science" have nothing to do with his involvement in computer design or construction.

(e) Lammens, Joe (1990), "Universal Program", a Common Lisp implementation of Turing's universal program as specified in [Davis and Weyuker, 1983], accessed 19 February 2014 from:
http://www.cse.buffalo.edu/∼rapaport/lammens.lisp

(f) Lodder, Jerry, "Introducing Logic via Turing Machines", accessed 25 February 2014 from:
http://www.math.nmsu.edu/hist_projects/j13.html

- See also Rapaport 1985 and Tymoczko & Goodhart 1986, above.
- See also other Turing-machine exercises at:
  Bezhanishvili, Guram; Leung, Hing; Lodder, Jerry; Pengelley, David; & Ranjan, Desgh, "Teaching Discrete Mathematics via Primary Historical Sources", accessed online, 25 February 2014, from:
  http://www.math.nmsu.edu/hist_projects/
  and in PDF from:
  http://www.math.nmsu.edu/hist_projects/DMRG.turing.pdf

---

[34] http://www.amazon.com/gp/reader/0470229055/ref=sib_dp_pt#reader-link
[35] http://www.alanturing.net/turing_archive/pages/Reference%20Articles/referencearticlesindex.html
[36] http://www.alanturing.net/turing_archive/pages/Reference%20Articles/The%20Turing-Church%20Thesis.html
[37] http://www.alanturing.net/turing_archive/pages/Reference%20Articles/What%20is%20a%20Turing%20Machine.html

(g) Michaelson, Greg (2012), "A Visit to the Turing Machine: A Short Story", *CS4FN: Computer Science for Fun* 14 (April),
http://www.cs4fn.org/turing/avisittotheturingmachine.php
(accessed 26 February 2014).

(h) Smith, Alvy Ray (2014), "Turingtoys.com",
http://alvyray.com/CreativeCommons/TuringToysdotcom.htm
and "A Business Card Universal Turing Machine" (2 August 2014),
http://alvyray.com/CreativeCommons/BizCardUniversalTuringMachine_v2.2.pdf
(both accessed 24 March 2015)

- A universal Turing machine on a business card!
- For more information, see:
  Casselman, Bill (2014), "About the Cover: Smart Card", *Notices of the American Mathematical Society* 61(8) (September): 872,
  http://www.ams.org/notices/201408/201408-full-issue.pdf   (accessed 24 March 2015).

(i) Suber, Peter (1997), "Turing Machines"
http://www.earlham.edu/∼peters/courses/logsys/turing.htm
accessed 7 November 2012.

- A 2-part handout;
  click on "second hand-out" at the end of part I to get to part II:
  http://www.earlham.edu/∼peters/courses/logsys/turing2.htm
  accessed 7 November 2012.
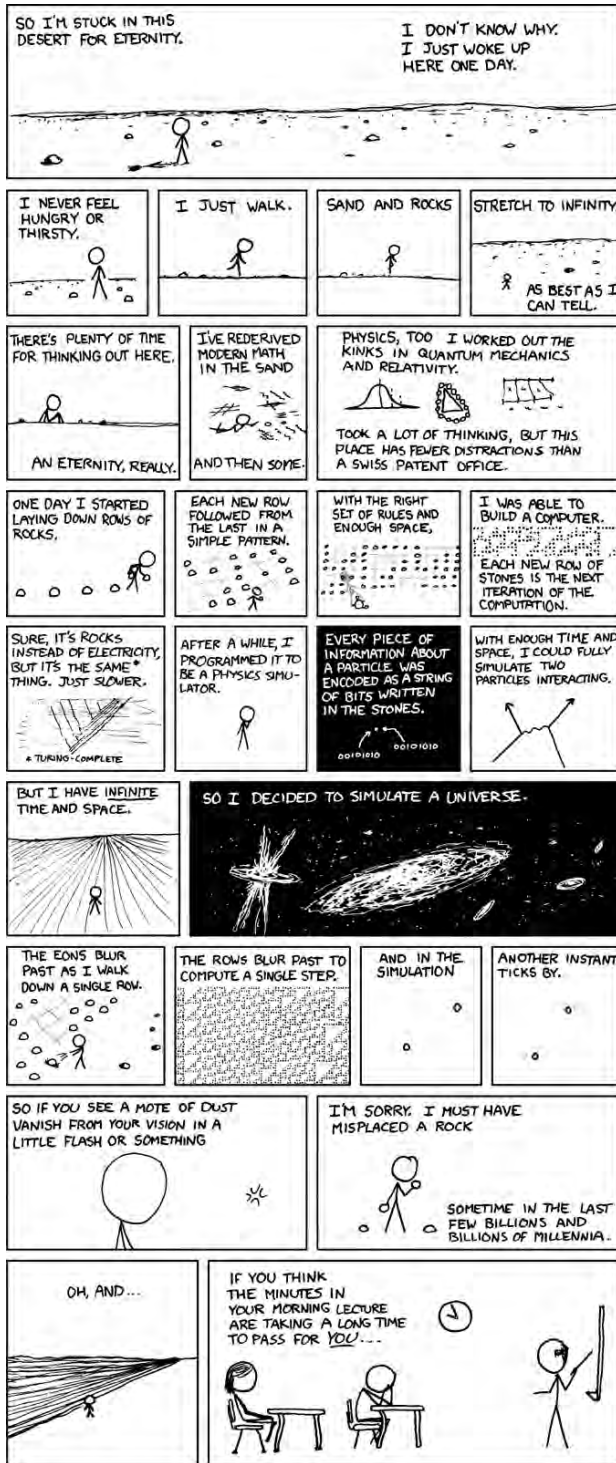
Figure 8.3: ©YEAR by ??

Chapter 9

What Is a Computer?
A Philosophical Perspective

What is computation? *In virtue of what is something a computer? Why do we say a slide rule is a computer but an egg beater is not?* These are. . . the philosophical questions of computer science, inasmuch as they query foundational issues that are typicaly glossed over as researchers get on with their projects. [Churchland and Sejnowski, 1992, 61, italics added]

. . . everyone who taps at a keyboard, opening a spreadsheet or a word-processing program, is working on an incarnation of a Turing machine. . . (*Time* magazine, 29 March 1999, cited in [Davis, 2006a, 125])

Figure 9.1: ©2012 by ??

9.1 Recommended Readings:

1. Required:

   (a) Searle, John R. (1990), "Is the Brain a Digital Computer?", *Proceedings and Addresses of the American Philosophical Association* 64(3) (November): 21–37.

      i. online version (without the footnotes) accessed 7 November 2012 from:
         http://users.ecs.soton.ac.uk/harnad/Papers/Py104/searle.comp.html

      ii. A slightly revised version is often cited instead of the 1990 article:
         Searle, John (1992), *The Rediscovery of the Mind* (Cambridge, MA: MIT Press), Ch. 9.

   (b) Hayes, Patrick J. (1997), "What Is a Computer? An Electronic Discussion", *The Monist* 80(3).

      • Original emails accessed 7 November 2012 from:
         http://www.philo.at/mii/wic.dir9601/maillist.html

   (c) Lloyd, Seth; & Ng, Y. Jack (2004), "Black Hole Computers", *Scientific American* 291(5) (November): 52–61.

      • Argues that the universe is a computer.

   (d) Weinberg, Steven (2002), "Is the Universe a Computer?", *New York Review of Books* 49(16) (24 October).

      i. This is a review of Wolfram 2002 (see "Strongly Recommended", below).

      ii. There is a follow-up letter:
         Shepard, Harvey; & Weinberg, Steven (2003, 16 January), "Is the Universe a Computer?", *New York Review of Books*.

2. Very Strongly Recommended:

   (a) Samuel, Arthur L. (1953), "Computing Bit by Bit, or Digital Computers Made Easy", *Proceedings of the IRE* [Institute of Radio Engineers] 41(10) (October): 1223–1230.

       - An introduction to computers, aimed at (radio) engineers who might not be familiar with them; written by an IBM researcher who later became famous for his work on computer checkers-players.

   (b) Shagrir, Oron (2006), "Why We View the Brain as a Computer", *Synthese* 153(3) (December): 393–416.

       - §1, "The Problem of Physical Computation: What Does Distinguish Computers from Other Physical Systems?", contains a good survey of various theories of what a computer is.

   (c) Piccinini, Gualtiero (2006), "Computational Explanation in Neuroscience", *Synthese* 153(3) (December): 343–353.

       - §§1–4 are a good summary of issues related to the nature of computationalism, observer-dependence (as opposed to what Searle calls "intrinsic" computationalism), and universal realizability (or "pancomputationalism").

   (d) Piccinini, Gualtiero (2007), "Computational Modelling vs. Computational Explanation: Is Everything a Turing Machine, and Does It Matter to the Philosophy of Mind?", *Australasian Journal of Philosophy* 85(1): 93–115

       - Accessed 7 November 2012 from:
         http://www.umsl.edu/~piccininig/Is_Everything_a_Turing machine.pdf

3. Strongly Recommended:

   (a) §1 of:
       Rapaport, William J. (1998), "How Minds Can Be Computational Systems", *Journal of Experimental and Theoretical Artificial Intelligence* 10: 403–419.

       i. §1 is relevant to Searle 1990; it presents my understanding of what it means to say that cognition is computable (as opposed to "computational").

       ii. §§2–3 criticize two other theories of computationalism; you can just skim these for now.

   (b) Wolfram, Stephen (2002), "Introduction" to *A New Kind of Science* (Wolfram Media)

       - Accessed 7 November 2012 from:
         http://www.stephenwolfram.com/publications/recent/nksmain/

   (c) Piccinini, Gualtiero (2010), "The Mind as Neural Software? Understanding Functionalism, Computationalism, and Computational Functionalism", *Philosophy and Phenomenological Research* 81(2) (September); 269–311

       - Preprint accessed online 29 January 2014 from:
         http://www.umsl.edu/~piccininig/Computational_Functionalism.htm

   (d) Powell, Corey S. (2006), "Welcome to the Machine", *New York Times Book Review* (2 April): 19.

- Accessed 25 March 2014 from:
  http://www.nytimes.com/2006/04/02/books/review/02powell.html
- A review of Seth 2006 (see "Further Sources of Information", below)

(e) Schmidhuber, Jürgen (2006), "The Computational Universe", *American Scientist* 94 (July-August): 364ff.

- Accessed 25 March 2014 from:
  http://www.americanscientist.org/bookshelf/pub/the-computational-universe
- Another review of Seth 2006 (see "Further Sources of Information", below)

(f) Hayes, Brian (2007), "Trains of Thought", *American Scientist* 95(2) (March-April): 108–113.

- Can a railroad layout be a computer?

(g) Rapaport, William J. (2007), "Searle on Brains as Computers", *American Philosophical Association Newsletter on Philosophy and Computers* 6(2) (Spring): 4–9.

  i. Accessed 11 March 2014 from:
     http://c.ymcdn.com/sites/www.apaonline.org/resource/collection/EADE8D52-8D02-4136-9A2A-729368501E43/v06n2Computers.pdf
  ii. A direct reply to Searle 1990.

Figure 9.2: ©2013, Creators Syndicate

9.2 Introduction

We began our study of the philosophy of computer science by asking what computer science is. We then asked what it studies: Does it study *computers*? Or *computing*? Or maybe something else? In Chapter 6, we began our investigation into what a *computer* is, from a historical perspective. And, in Chapters 7 and 8, we began our investigation into what *computing* is.

In the present chapter, armed with the results of these investigations, we return to our earlier question: If computer science is the study of computers, what *is* a computer?

At the very least, we might say that a computer is a machine designed (that is, engineered) to compute, to do computations. But does it *have* to be a "machine"? Does it *have* to be "engineered"? Is the brain a computer? If so, then it would seem that computers could be *biological* entities (which, arguably, are not machines) that *evolved* (which, arguably, means that they were not engineered).

And what kind of computations do computers perform? Are they restricted to mathematical computations? Even if that's so, is that a restriction? The binary-representation insight (§7.6.1 suggests that any (computable) information can be represented as a binary numeral; hence, any computation on such information could be considered to be a mathematical computation.

And what about the difference between a "hardwired" Turing machine that can only compute one thing and a universal machine that can compute anything that is computable? Or the difference between a real, physical computer that can only compute whatever is *practically* computable (that is, subject to reasonable space and time constraints) and an abstract, universal Turing machine that is not thus constrained?

In other words, what is a computer?

If you ask a random person what a computer is, they will probably try to describe their laptop. If you look up 'computer' in a reference book, you will find things like this (from the *Oxford English Dictionary*):

computer, *n***.**

1. A person who makes calculations or computations; a calculator, a reckoner; spec[ifically,] a person employed to make calculations in an observatory, in surveying, etc. Now chiefly *hist[orical]*
   [earliest citation dated 1613]

2. A device or machine for performing or facilitating calculation.
   [earliest citation dated 1869]

3. An electronic device (or system of devices) which is used to store, manipulate, and communicate information, perform complex calculations, or control or regulate other devices or machines, and is capable of receiving information (data) and of processing it in accordance with variable procedural instructions (programs or software); *esp[ecially]* a small, self-contained one for individual use in the home or workplace, used esp. for handling text, images, music, and video, accessing and using the Internet, communicating with other people (e.g. by means of email), and playing games.
   [earliest citation dated 1945]

(*OED*, http://www.oed.com/view/Entry/37975)

Or this (from the *Encyclopedia of Computer Science*):

A *digital computer* is a machine that will accept data and information presented to it in a discrete form, carry out arithmetic and logical operations on this data, and then supply the required results in an acceptable form. [Morris and Reilly, 2000, 539]

One common thread in such definitions (ignoring the ones that are only of historical interest) is that computers are:

1. devices or machines

2. that take input (data, information),

3. process it (manipulate it; or operate, calculate, or compute with it)

4. in accordance with instructions (a program),

5. and then output a result (presumably, more data or information, but also including control of another device).

There are some other features that are usually associated with "computers": The kind that we are interested in must be, or typically are:

automatic
There is no human intervention (beyond, perhaps, writing the program). Of course, the holy grail of programming is to have self-programmed computers, possibly to include having the "desire" or "intention" to program themselves (as

in science fiction). Humans might also supply the input or read the output, but that hardly qualifies as "intervention". (We will explore "intervention"—in the guise of "interactive" computing—in Ch. 11.)

general purpose

A computer must be capable of *any* processing that is "algorithmic", by means of a suitable program. This is the heart of Turing's universal machine. Recall that a Turing machine "runs" only *one* program. The universal Turing machine is also a Turing machine, so it, too, also runs only one program, namely, the fetch-execute cycle that *simulates* the running of *another* (that is, *any* other) single-program Turing machine.

electronic

Modern computers are, as a matter of fact, electronic, but there is work on quantum computers ([Grover, 1999, Aaronson, 2008, Aaronson, 2011, Aaronson, 2014, Hayes, 2014b], and other references cited in §31), optical computers CITE, biological computers [Adleman, 1998], etc. Being electronic is not essential. The crucial property is, rather, to be constructed in such a way as to allow for high processing speeds or other kinds of efficiencies.

digital

They should process information expressed in discrete, symbolic (typically, alpha-numeric) form, but perhaps also including graphical form. The contrast is typically with being "analog", where information is represented by means of continuous physical quantities (see [Jackson, 1960], [Pour-El, 1974], [Holst, 2000], [Piccinini, 2008], [Piccinini, 2011] [McMillan, 2013]).

What about the "calculations", the "arithmetic and logical operations"? Presumably, these need to be algorithmic, though neither the *OED* nor the *Encyclopedia of Computer Science* definitions say so. And it would seem that the authors of those definitions have in mind calculations or operations such as addition, subtraction, etc.; maybe solving differential equations; Boolean operations involving conjunction, disjunction, etc; and so on. These require the data to be numeric (for math) or propositional (or truth-functional—for Boolean and logical operations), at least in some "ultimate" sense: That is, any *other* data (pictorial, etc.) must be encoded as numeric or propositional, or else would need to allow for other kinds of operations.

There certainly seem to be clear cases of *digital* computers: Macs, PCs, etc. There certainly seem to be clear cases of *analog* computers—slide rules, certain machines at various universities (Indiana, Harvard)[1]—though these may be mostly of historical interest, don't seem to be programmable (i.e., universal, in Turing's sense), and seem to be outside the historical development explored in Chapter 6.

And there seem to be clear cases of entities that are *not* computers: I would guess that most people would not consider rocks, walls, ice cubes, or solid blocks of plastic to be computers (note that I said 'most' people!). And there are even clear cases of

---

[1]For Indiana, see [Hedger, 1998]'s report on the research of Jonathan W. Mills; for Harvard, see "Harvard Robotics Laboratory: Analog Computation", accessed 31 January 2014 from http://hrl.harvard.edu/analog/.

devices for which it might be said that it is not clear whether, or in what sense, they are computers, such as Atanasoff and Berry's ABC.

So: What is a computer? What is the relation of a computer to a Turing machine and to a univeral Turing machine? Is the (human) brain a computer? Is your smartphone a computer? *Could* a rock or a wall be considered to be a computer? Might *anything* be a computer? Might *everything*—such as the universe itself—be a computer? Or are some of these just badly formed questions?

9.3   John Searle: Anything Is a Computer

John Searle's presidential address to the American Philosophical Association, "Is the Brain a Digital Computer?" [Searle, 1990], covers a lot of ground and makes a lot of points about the nature of computers, the nature of the brain, the nature of cognition, and the relationships among them. In this section, we are going to focus only on what Searle says about the nature of computers, with only a few side glances at the other issues.[2]

Here is Searle's argument relevant to our main question about what a computer is:

1. Computers are described in terms of 0s and 1s.

   (This is what he says. Taken literally, he is sahying that computers are described in terms of certain *numbers*. Instead, he might have said that computers are described in terms of '0's and '1's. In other words, he might have said that computers are described in terms of certain *numerals*. Keep this distinction in mind as we discuss Searle's argument!)[3]

2. Therefore, being a computer is a syntactic property.

   (Syntax is the study of the properties of, and relations among, symbols or uninterpreted marks on paper (or on some other medium); a rough synonym is 'symbol manipulation'.[4]

3. Therefore, being a computer is not an intrinsic property of physical objects.

4. Therefore, we can ascribe the property of being a computer to any object.

5. Therefore, everything is a computer.

Of course, this doesn't quite answer our question, "What is a computer?". Rather, the interpretation and truth value of these theses will depend on what Searle thinks a computer is. Let's look at exactly what Searle says about these claims.

---

[2]For more detailed critiques and other relevant commentary, see [Piccinini, 2006b, Piccinini, 2007b, Piccinini, 2010a, Rapaport, 2007].

[3]Recall the difference between numbers and numerals from §2.3.

[4]We discuss the nature of syntax and its relationship to semantics in Ch. 22.

9.3.1 Computers Are Described in Terms of 0s and 1s

First, after briefly describing Turing machines as devices that can perform the actions of printing '0' or '1' on a tape and of moving left or right on the tape, depending on conditions specified in its program, he says this:

> If you open up your home computer you are most unlikely to find any 0's and 1's or even a tape. But this does not really matter for the definition. To find out if an object is really a digital computer, it turns out that we do not actually have to look for 0's and 1's, etc.; rather we just have to look for something that we could *treat as* or *count as* or *could be used to* function as 0's and 1's. (p. 25.)

So, according to Searle, a computer is a physical object that can be described as a Turing machine. Recall from §8.8 that anything that satisfies the definition of a Turing machine *is* a Turing machine, whether it has a paper tape divided into squares with the symbols '0' or '1' printed on them or whether it is a table and chairs with beer mugs on them. All we need is to be able to "treat" some part of the physical object as playing the role of the Turing machine's '0's and '1's. So far, so good.

Or is it? Is your home computer *really* a Turing machine? Or is it a device whose behavior is "merely" *logically equivalent* to that of a Turing machine? That is, is it a device that can compute all and only the functions that a Turing machine can compute, even if it does so differently from the way that a Turing machine does? Recall that there are lots of different mathematical models of computation: Turing machines and recursive functions are two of them that we have looked at. Suppose someone builds a computer that operates in terms of recursive functions instead of in terms of a Turing machine. That is, it can compute successors, predecessors, and projection functions, and it can combine these using generalized composition, conditional definition, and while-recursion, instead of printing '0's and '1's, moving left and right, and combining these using "go to" instructions (changing from one *m*-configuration to another). These two computers (the Turing-machine computer and the recursive-function computer), as well as your home computer (with a "von Neumann" architecture, whose method of computation uses the primitive machine-language instructions and control structures of, say, an Intel chip), are all logically equivalent to a Turing machine, but their internal behaviors are radically different. Can we really describe the recursive-function computer and your home computer in terms of a Turing-machine's '0's and '1's? Or are we limited to showing that anything that the recursive-function computer and your home computer can compute can also be computed by a Turing machine (and vice versa)?

Here is an analogy to help you see the issue: Consider translating between French and English. To say 'It is snowing' in French, you say: *Il neige*. The 'il' means "it", and the 'neige' means "is snowing". This is very much like (perhaps it is exactly like) describing the recursive-function machine's behavior (analogous to the French sentence) using '0's and '1's (analogous to the English sentence). But here is a different example: In English, if someone says: 'Thank you', you might say, 'You're welcome', but, in French, if someone says *Merci*, you might reply: *Je vous en prie*. Have we translated the English into French, in the way that we might "translate" a recursive-

function algorithm into a Turing machine's '0's and '1's? Not really: Although 'merci' is used in much the same way in French that 'thank you' is used in English, there is no part of 'merci' that means 'thank' or 'you'; and the literal translation of 'je vous en prie' is something like 'I pray that of you'. So there is a way of communicating the same information in both French and English, but the phrases used are not literally inter-translatable.

So, something might be a computer without being "described in terms of '0's and '1's", depending on exactly what you mean by 'described in terms of'. But let's grant Searle the benefit of the doubt (perhaps he should have said something like this: computers are described in terms of the primitive elements of the mathematical model of computation that they implement), and continue looking at his argument.

9.3.2 Being a Computer Is a Syntactic Property

Let us suppose, for the sake of the argument, that computers are described in terms of '0's and '1's. Such a description is *syntactic*. This term (which pertains to symbols, words, grammar, etc.) is usually contrasted with 'semantic' (which pertains to meaning), and Searle emphasizes that contrast early in his essay when he says that "syntax is not the same as, nor is it by itself sufficient for, semantics" [Searle, 1990, 21]. But here Searle uses the term 'syntactic' as a contrast to being *physical*. Just as there are many ways to be computable (Turing machines, recursive functions, lambda-calculus, etc.)—all of which are equivalent—so there are many ways to be a carburetor. "A carburetor...is a device that blends air and fuel for an internal combustion engine" (http://en.wikipedia.org/wiki/Carburetor), *but it doesn't matter what it is made of*, as long as it can perform that blending "function" (purpose). "[C]arburetors can be made of brass or steel" [Searle, 1990, 26]; they are "multiply realizable"—that is, you can "realize" (or make) one in "multiple" (or different) ways. They "are defined in terms of the production of certain *physical* effects" [Searle, 1990, 26].

> But the class of computers is defined **syntactically** in terms of the *assignment* of 0's and 1's. [Searle, 1990, 26, Searle's italics, my boldface]

In other words, if something is defined in terms of symbols, like '0's and '1's, then it is defined in terms of syntax, not in terms of what it is physically made of.

Hence, being a computer is a syntactic property, not a physical property. It is a property that something has in virtue of... of what? There are two possibilities, given what Searle has said. First, perhaps being a computer is a property that something has in virtue of *what it does, its function or purpose*. Second, perhaps being a computer is a property that something has in virtue of *what someone **says** that it does, how it is described*. But what something *actually* does may be different from what someone *says that* it does.

So, does Searle think that something is a computer in virtue of its function or in virtue of its syntax? Suppose you find a black box (with a keyboard and a screen) in the desert and that, by experimenting with it, you determine that it displays on its screen the greatest common divisor (GCD) of two numbers that you type into it. (We discussed a similar thought experiment in §3.8; see [Weizenbaum, 1976, Ch. 5] for further discussion). It certainly seems to function as a computer (as a Turing machine

for computing GCDs). And you can probably describe it in terms of '0's and '1's, so you can also say that it is a computer. It seems that if something functions as a computer, you can describe it in terms of '0's and '1's.

What about the converse? Suppose that the black box's behavior is inscrutable; the symbols on the keys are unrecognizable, and the symbols displayed on the screen don't seem to be related in any obvious way to the input symbols. But suppose that someone manages to invent an interpretation of the symbols in terms of which the box's behavior can be described as computing GCDs. Is that really what it does? Might it not have been created by some extraterrestrials solely for the purpose of entertaining their young with displays of meaningless symbols, and that it is only by the most convoluted (and maybe not always successful) interpretation that it can be described as computing GCDs?

You might think that the box's function is more important for determining what it is. Searle thinks that our ability to describe it syntactically is more important! After all, whether or not the box was *intended* by its creators to compute GCDs or to entertain toddlers, if it can be accurately described as computing GCDs, then, in fact, it computes GCDs (as well as, perhaps, entertaining toddlers with pretty pictures).

Again, let's grant this point to Searle. He then goes on to warn us:

But this has two consequences which might be disastrous:

1. The same principle that implies multiple realizability would seem to imply universal realizability. If computation is defined in terms of the assignment of syntax then everything would be a digital computer, because any object whatever could have syntactical ascriptions made to it. You could describe anything in terms of 0's and 1's.

2. Worse yet, syntax is not intrinsic to physics. The ascription of syntactical properties is always relative to an agent or observer who treats certain physical phenomena as syntactical. [Searle, 1990, 26]

Let's take these in reverse order.

9.3.3 Being a Computer Is Not an Intrinsic Property of Physical Objects

According to Searle, being a computer is not an intrinsic property of physical objects, because being a computer is a syntactic property, and "syntax is not intrinsic to physics". What does that quoted thesis mean, and why does Searle think that it is true?

What is an "intrinsic" property? [5] Searle doesn't tell us, though he gives some examples:

[G]reen leaves intrinsically perform photosynthesis[;]...hearts intrinsically pump blood. It is not a matter of us arbitrarily or "conventionally"

---

[5]For more discussion on what 'intrinsic' means, see [Skow, 2007] and [Bader, 2013] (the latter is a highly technical essay, but it contains useful references to the literature on "intrinsic properties"). ALSO CITE LEWIS ET AL.

> assigning the word "pump" to hearts or "photosynthesis" to leaves. There is an actual fact of the matter. [Searle, 1990, 26]

So, perhaps "intrinsic" properties are properties that something "really" has as opposed to merely being *said* to have, much the way our black box in the previous section may or may not "really" compute GCDs but can be *said* to compute them. But what does it mean to "really" have a property?

Here are some possible meanings for 'intrinsic':

- An object might have a property *P* "intrinsically" if it has *P* "essentially" rather than "accidentally". An accidental property is a property that something has such that, if the object lacked that property, it would still be the same object. So, it is merely an accidental property of me that I was wearing a tan shirt on the day that I wrote this sentence. If I lacked that property, I would still be the same person. An essential property is a property that something has such that, if the object lacked that property, it would be a different object. So, it is an essential property of me that I am a human being. If I lacked that property, I wouldn't even be a person at all. (This is the plot of Franz Kafka's novel *The Metamorphosis*, in which the protagonist awakes one day to find that he is no longer a human, but a beetle.) The exact nature of the essential-accidental distinction, and its truth or falsity, are matters of great dispute in philosophy. Here, I am merely suggesting that perhaps this is what Searle means by 'intrinsic': Perhaps he is saying that being a computer is not an essential property of an object, but only an accidental property.

- An object might have *P* "intrinsically" if it has *P* as a kind of "first-order" property, not as a kind of "second-order" property. A second-order property is a property that something has in virtue of having some other (or "first-order") property. Here, I am thinking of a property like an object's color. An apple has the property of being red, not "intrinsically", but in virtue of reflecting light with wavelength approximately 650 nm. We perceive such reflected light in a certain way, which we call 'red'. But, conceivably, someone with a different neural make-up (say, red-green color-blindness) might perceive it either as a shade of gray that is indistinguishable from green or, in a science-fiction kind of case called an "inverted spectrum", as green. The point is that it is the measurable wavelength of reflected light that might be an "intrinsic" property belonging to the apple, whereas the perceived color is a property belonging to the perceiver and only "secondarily" to the apple itself. So, perhaps Searle is saying that being a computer is only a second-order property of an object.

- Another way that an object might have *P* "intrinsically" (perhaps this is closer to what Searle has in mind) is if having *P* is a "natural kind". This is another controversial notion in philosophy. The idea, roughly, is that a natural kind is a property that something has as a part of "nature" and not as a result of what an observer thinks.[6] So, being a bear is a natural kind; there

---

[6]You'll note that several of these concepts are closely related; they may even be indistinguishable. For instance, perhaps "essential" properties are "natural kinds" or perhaps second-order properties are not natural

would be bears even if there were no people to see them or to call them 'bears'. This does not mean that it is easy to define such natural kinds. Is a bear "A heavily-built, thick-furred plantigrade quadruped, of the genus *Ursus*; belonging to the *Carnivora*, but having teeth partly adapted to a vegetable diet" (http://www.oed.com/view/Entry/16537)—that is, a heavily-built, thick-furred mammal that walks on the soles of its four feet, eats meat, but can also eat plants? What if a bear was born with only three feet, or never eats meat? Is an animal a bear whether or not it satisfies such definitions, and whether or not there were any humans to try to give such definitions? If so, then being a bear is a natural kind and, perhaps, an "intrinsic" property. As we saw in Chapter 3.4, Plato once said that it would be nice if we could "carve nature into its joints", that is, find the real, perhaps "intrinsic", natural properties of things (Phaedrus 265d–e). But perhaps the best we can do is to "carve joints *into* nature", that is, to "overlay" categories onto nature so that we can get some kind of understanding of it and control over it, even if those categories don't really exist in nature. Is being a computer a natural kind? Well, it's certainly not like being a bear! There probably aren't any computers in nature (unless the brain is a computer; and see §9.7.2 on whether nature itself is a computer), but there may also not be any prime numbers in nature, yet mathematical objects are something thought to exist independently of human thought. If they do, then, because a Turing machine is a mathematical object, it might exist independently of human thought and, hence, being a computer might be able to be considered to be a "natural" mathematical kind. So, perhaps Searle is saying that being a computer is not a natural kind in one of these senses.

- In fact, a computer is probably not a natural kind for a different reason: It is an artifact, something created by humans. Again, the nature of artifacts is controversial: Clearly, chairs, tables, houses, atomic bombs, and pencils are artifacts; you don't find them in nature, and if humans had never evolved, there probably wouldn't be any of these artifacts. But what about bird's nests, beehives, beaver dams, and other such things constructed by non-human animals? What about socially "constructed" objects like money? One of the crucial features of artifacts is that what they are is relative to what a person says they are. You won't find a table occurring naturally in a forest, but if you find a tree stump, you might use it as a table. So, something might be a computer, Searle might say, only if a human uses it that way or can describe it as one.

  In fact, Searle says this explicitly:

  [W]e might discover in nature objects which had the same sort of shape as chairs and which could therefore be used as chairs; but we could not discover objects in nature which were functioning as chairs, except relative to some agents who regarded them or used them as chairs. [Searle, 1990, 28].

---

kinds. Investigating these relationships is one of the tasks of metaphysics. It is beyond the scope of the philosophy of computer science.

Why is syntax not "intrinsic" to physics? Because " 'syntax' is not the name of a physical feature, like mass or gravity....[S]yntax is essentially an observer relative notion" [Searle, 1990, 27]. I think that what Searle is saying here is that we can analyze physical objects in different ways, no one of which is "privileged" or "more correct"; that is, we can carve nature into different joints, in different ways. On some such carvings, we may count an object as a computer; on others, we wouldn't. By contrast, an object has mass independently of how it is described: *Having* mass is *not* relative to an observer. *How* its mass is measured *is* relative to an observer.

But couldn't being a computer be something like that? There may be lots of different ways to measure mass, but an object always has a certain quantity of mass, no matter whether you measure it in grams or in some other units. In the same way, there may be lots of different ways to measure length, but an object always has a certain length, whether you measure it in centimeters or in inches. Similarly, an object (natural or artifactual) will have a certain structure, whether you describe it as a computer or as something else. If that structure satisfies the definition of a Turing machine, then it *is* a Turing machine, no matter how anyone *describes* it.

Searle anticipates this reply:

> [S]omeone might claim that the notions of "syntax" and "symbols" are just a manner of speaking and that what we are really interested in is the existence of systems with discrete physical phenomena and state transitions between them. On this view we don't really need 0's and 1's; they are just a convenient shorthand. [Searle, 1990, 27]

Compare this to my example above: Someone might claim that specific units of measurement are just a manner of speaking and that what we are really interested in is the actual length of an object; on this view, we don't really need centimeters or inches; they are just a convenient shorthand.

Searle replies:

> But, I believe, this move is no help. **A physical state of a system is a computational state only relative to the assignment to that state of some computational role, function, or interpretation.** The same problem arises without 0's and 1's because notions such as computation, algorithm and program do not name intrinsic physical features of systems. Computational states are not *discovered within* the physics, they are *assigned* to the physics. [Searle, 1990, 27, my boldface, Searle's italics.]

But this just repeats his earlier claim; it gives no new reason to believe it. He continues to insist that being a computer is more like "inches" than like length.

So, we must ask again: Why is syntax not intrinsic to physics? Perhaps, if a property is intrinsic to some object, then that object can only have the property in one way. For instance, color is presumably not intrinsic to an object, because an object might have different colors depending on the conditions under which it is perceived. But the physical structure of an object that causes it to reflect a certain wavelength of light is always the same; that physical structure is intrinsic. On this view, here is a reason why syntax might not be intrinsic: The syntax of an object is, roughly,

its abstract structure (see §§14.4 and 19.5.3.2). But an object might be able to be understood in terms of several different abstract structures (and this might be the case whether or not human observers assign those structures to the object). If an object has no unique syntactic structure, then syntax is not intrinsic to it. But if an object has (or can be assigned) a syntax of a certain kind, then it does have that syntax even if it also has another one. And if, under one of those syntaxes, the object is a computer, then it *is* a computer.

But that leads to Searle's next point.

9.3.4 We Can Ascribe the Property of Being a Computer to Any Object

There is some slippage in the move from "syntax is not intrinsic to physics" to "we can ascribe the property of being a computer to any object". Even if syntax is not intrinsic to the physical structure of an object, perhaps because a given object might have several different syntactic structures, why must it be the case that any object can be ascribed the syntax of being a computer?

One reason might be this: Every object has (or can be ascribed) every syntax. That seems to be a very strong claim. To refute it, however, all we would need to do is to find an object O and a syntax S such that O lacks (or cannot be ascribed) S. One possible place to look would be for an O whose "size" in some sense is smaller than the "size" of some S. I will leave this as an exercise for the reader: If you can find such O and S, then I think you can block Searle's argument at this point.

Here is another reason why any object might be able to be ascribed the syntax of being a computer: There is something special about the syntax of being a computer—that is, about the formal structure of Turing machines—that does allow it to be ascribed to (or found in) any object. This may be a bit more plausible than the previous reason. After all, Turing machines are fairly simple. Again, to refute it, we would need to find an object O such that O lacks (or cannot be ascribed) the syntax of a Turing machine. Again, I will leave this as an exercise for the reader, but we will return to it later (in Ch. 14). Searle thinks that we cannot find such an object.

9.3.5 Everything Is a Computer

We can ascribe the property of being a computer to any object if and only if everything is a computer.

> Thus for example the wall behind my back is right now implementing the Wordstar program, because there is some pattern of molecule movements which is isomorphic with the formal structure of Wordstar. [Searle, 1990, 27]

Searle does not offer a detailed argument for how this might be the case, but other philosophers have done so, and we will explore how they think it can be done when we look at the nature of "implementation" (Ch. 14). Let's assume, for the moment, that it can be done.

In that case, things are not good, because this trivializes the notion of being a computer. If everything has some property *P*, then *P* isn't a very interesting property; *P* doesn't help us categorize the world, so it doesn't help us understand the world.

How might we respond to this situation? One way is to bite the bullet and accept that, under some description, any object (even the wall behind me) can be considered to be a computer. And not just some specific computer, such as a Turing machine that executes the Wordstar program:

> [I]f the wall is implementing Wordstar then if it is a big enough wall it is implementing any program, including any program implemented in the brain. [Searle, 1990, 27]

If a big enough wall implements any program, then it implements the universal Turing machine!

But perhaps this is OK. After all, there is a difference between an "intended" interpretation of something and what I will call a "gerrymandered" interpretation. For instance, the intended interpretation of Peano's axioms for the natural numbers is the sequence $\langle 1, 2, 3, \ldots \rangle$. There are also many other "natural" interpretations, such as $\langle$I, II, III,$\ldots \rangle$. But extremely contorted ones, such as a sequence of all objects in the universe arranged alphabetically by their name or description in English, are hardly "good" examples. Admittedly, they *are* examples of natural numbers, but not very useful ones.

A better reply to Searle, however, is to say that he's wrong: Some things are *not* computers. Despite what he said in the last passage quoted above, the wall behind me is *not* a universal Turing machine; I really cannot use it to post to my Facebook account or even to write a letter. It is an empirical question whether something actually behaves as a computer. And the same goes for other syntactic structures. Consider the formal definition of a mathematical group (a set of objects [like integers] that is closed under an associative, binary operation [like addition] that has an identity element [like 0] and is such that every element of the set has an inverse [in the case of integer *n*, its negative $-n$]). *Not every set is a group.* Similarly, there is no reason to believe that everything is a Turing machine.

9.3.6 Summary

So, Searle believes that a computer is anything that is a Turing machine, and that everything is a Turing machine. I think that he is wrong about the second claim. But is it reasonable to answer the question, "What is a computer?", by saying that it is a Turing machine?

Let's take stock of where we are. Presumably, computers are things that compute. Computing is the process that Turing machines give a precise description for. That is, computing is what Turing machines do. And, what Turing machines do is to move around in a discrete fashion and print discrete marks on discrete sections of the space in which they move around. So, a computer is a *device*—presumably, a *physical* device— that does that. Searle agrees that computing is what Turing machines do, and he seems to agree that computers are devices that compute. He also believes that everything is a computer; more precisely, he believes that everything can be *described as* a computer

(because that's what it means to *be* a computer). And we've also seen reason to think that he might be wrong about that last point.

In the next two sections, we look at two other views about what a computer is.

9.4 Patrick Hayes: Computers as Magic Paper

Let's keep straight about three intertwined issues that we have been looking at:

1. What is a computer?

2. Is everything a computer?

3. Is the brain a computer?

Our principal concern is with the first question. Once we have an answer to that, we can try to answer the others. As we've just seen, Searle thinks that a computer is a Turing machine, that everything is (or can be correctly described as) a computer, and, therefore, that the brain is (trivially) a computer.

The AI researcher Patrick J. Hayes [Hayes, 1997] gives two definitions. Here's the first one:

Definition H1

> By "computer" I mean **a machine which performs computations, or which computes**. [Hayes, 1997, 390, my boldface.]

A full understanding of this requires a definition of 'computation'; this will be clarified in his second definition. But there are a few points to note about this first one.

He prefaces it by saying:

> First, I take it as simply obvious both that computers exist and that not everything is a computer, so that, contra Searle, the concept of "computer" is not vacuous. [Hayes, 1997, 390].

So, there are machines that compute (that is, there are things that *are* machines-that-compute), and there are things that are *not* machines-that-compute. Note that the latter can be true in two ways: There might be machines that *don't* compute, or there might be things that do compute but that *aren't machines*. Searle disputes the first possibility, because he thinks that everything (including, therefore, any machine) computes. But contrary to what Hayes says, Searle would probably *agree* with the second possibility, because, after all, he thinks that everything (including, therefore, anything that is not a machine) computes! Searle's example of the wall that implements (or that can be interpreted as implementing) Wordstar would be such a non-machine that computes. So, for Hayes's notion to contradict Searle, it must be that Hayes believes that there are machines that do not compute. Perhaps, that wall is one of them, or perhaps a dishwasher is a machine that doesn't compute anything.[7]

---

[7] A dishwasher might, however, be described by a (non-computable) function that takes dirty dishes as input and that returns clean ones as output.

Are these two "obvious" points to be understood as axioms—or criteria of adequacy for any definition—that Hayes thinks need no argument? Or are they intended to be theorems that follow from his first definition? If it's the former, then there is no interesting debate between Searle and Hayes; one simply denies what the other argues for. If it's the latter, then Hayes needs to provide arguments or examples to support his position.

A second thing to note about Hayes's definition is that he says that computers "*perform* computations", not "*can* perform computations". Strictly speaking, your laptop when it is turned off is not a computer by this definition, because it is not performing any computation. And, as Hayes observes,

> On this understanding, a Turing machine is not a computer, but a mathematical abstraction of a certain kind of computer. [Hayes, 1997, 390]

What about Searle's wall that implements Wordstar? There are two ways to think about how the wall might implement Wordstar. First, it might do so statically, simply in virtue of there being a way to map every part of the Wordstar program to some aspect of the molecular or subatomic structure of the wall. In that case, Hayes could well argue that the wall is *not* a Wordstar computer, because it is not computing (even if it *might be able to*). But the wall might implement Wordstar dynamically; in fact, that is why Searle thinks that the wall implements Wordstar:

> [B]ecause there is some pattern of molecule *movements* which is isomorphic with the formal structure of Wordstar. [Searle, 1990, 27, my italics]

But a pattern of *movements* suggests that Searle thinks that the wall *is computing*, so it *is* a computer!

Definition H2 is a bit more precise, and it is, presumably, his "official" one:

Definition H2
> [F]ocus on the memory. A computer's memory contains patterns... which are stable but labile [i.e., changeable], and it has the rather special property that changes to the patterns are under the control of other patterns: that is, some of them describe changes to be made to others; and when they do, the memory changes those patterns in the way described by the first ones.... A computer is **a machine which is so constructed that patterns can be put in it, and when they are, the changes they describe will in fact occur to them.** If it were paper, it would be "magic paper" on which writing might spontaneously change, or new writing appear. [Hayes, 1997, 393, my boldface]

There is a subtle difference between Hayes's two definitions, which highlights an ambiguity in Searle's presentation. Recall the distinction between a Turing machine and a universal Turing machine. A Turing machine computes one function. A universal Turing machine can compute any computable function as long as a Turing machine-program for that function is encoded on the universal Turing machine's tape. In short, a Turing machine is hardwired computer; a universal Turing machine is a programmable

computer. Definition H1 seems to include physical Turing machines (but, as he noted, not abstract ones), because, after all, they compute (at least, when they are turned on and running). Definition H2 seems to *exclude* them, because the second definition requires patterns that describe changes to other patterns. That first kind of pattern is a program; the second kind is the data that the program operates on. So, Definition H2 is for a universal Turing machine.

Is Searle's wall a Turing machine or a universal Turing machine? On Searle's view, Wordstar is a Turing machine, so the wall must be a Turing machine, too. So, the wall is not a computer on Definition H2. Could a wall (or a rock, or some other suitably large or complex physical object other than something like a PC or a Mac) be a universal Turing machine? My guess is that Searle would say "yes", but it is hard to see how one would actually go about programming it.

The "magic paper" aspect of Definition H2 focuses, as he notes, on the memory, that is, on the tape. It is as if you were looking at a universal Turing machine, but all you saw was the tape, not the read-write head or its states (*m*-configurations) or its mechanism. If you watch the universal Turing machine compute, you would see the patterns (the '0's and '1's) on the tape "magically" change.

The idea of a computer as "magic paper" may seem a bit fantastic. But there are more down-to-earth ways of thinking about this. The philosopher Richmond Thomason has said that

> ...all that a program can do between receiving an input and producing an output is to change variable assignments... [Thomason, 2003, 328].[8]

If programs tell a computer how to change the assignments of values to variables, then a computer is a (physical) device that changes the contents of register cells (the register cells that are the physical implementations of the variables in the program). This is really just another version of Turing's machines, if you consider the tape squares to be the register cells.

Similarly, Stuart C. Shapiro points out that "a computer is a device consisting of a vast number of connected switches. ...[T]he switch settings both determine the operation of the device and can be changed by the operation of the device" [Shapiro, 2001, 3].

What is a "switch"? Here is a nice description from a 1953 article introducing computers to engineers who might never have encountered them before:

> To bring the discussion down to earth let us consider the ordinary electric light switch in your home. This is by definition a switch. It enables one to direct electric current to a lighting fixture at will Usually there is a detent mechanism which enables the switch to remember what it is supposed to be doing so that once you turn the lights on they will remain on. It therefore has a memory. It is also a binary, or perhaps we should say a bistable device. By way of contrast, the ordinary telegrapher's key is a switch without memory since the key will remain down only as long as it is depressed by the operator's hand. But the light switch and the

---

[8]A similar point is made in [Lamport, 2011, 6]: "an execution of an algoritihm is a sequence of states, where a state is an assignment of values to variables."

> telegraph key are binary devices, that is, they have but two operating states.
> [Samuel, 1953, 1225]

So, a switch is a physical implementation of a Turing-machine's tape cell, which can also be "in two states" (that is, have one of two symbols printed on it) and also has a "memory" (that is, once a symbol is printed on a cell, it remains there until it is changed).

Hayes's magic-paper patterns are just Shapiro's switch-settings or Thomason's variable assignments. (For more on computers as switch-setting devices, see the discussions of how train switches can implement computations in [Stewart, 1994] and [Hayes, 2007b]; both of these are also examples of Turing machines implemented in very different media than silicon (namely, trains)!)

Does this definition satisfy Hayes's two criteria? Surely, such machines exist. I am writing this book on one of them right now. And surely not everything is such a machine: At least on the face of it, the stapler on my desk is not such "magic paper". Searle, I would imagine, would say that we might see it as such magic paper if we looked at it closely enough and in just the right way. And so the difference between Searle and Hayes seems to be in how one is supposed to look at candidates for being a computer: Do we look at them as we normally do? In that case, not everything is a computer. Or do we look at them closely in a certain way? In that case, perhaps we could see that everything could be considered to be a computer. Isn't that a rather odd way of thinking about things?

Is the brain a computer in the sense of "magic paper"? If Hayes's "patterns" are understood as patterns of neuron firings, then, because surely some patterns of neuron firings cause changes in other such patterns, I think Hayes would consider the brain to be a computer.

9.5 Gualtiero Piccinini: Computers as Digital String Manipulators

In a series of three papers, the philosopher Gualtiero Piccinini has offered an analysis of what a computer is that is more precise than Hayes's and less universal than Searle's [Piccinini, 2007b], [Piccinini, 2007c], [Piccinini, 2008]. It is more precise than Hayes's, because it talks about *how* the magic paper performs its tricks. And it is less universal than Searle's, because Piccinini doesn't think that everything is a computer.

Unfortunately, there are two slightly different definitions to be found in Piccinini's papers:

Definition P1

> "The mathematical theory of how to generate output strings from input strings in accordance with general rules that apply to all input strings and depend on the inputs (and sometimes internal states) for their application is called computability theory. Within computability theory, the activity of manipulating strings of digits in this way is called computation. **Any system that**

performs this kind of activity is a computing system properly so called."
[Piccinini, 2007b, 108, my emphasis]

Definition P2

"**[A]ny system whose correct mechanistic explanation ascribes to it the
function of generating output strings from input strings (and possibly
internal states), in accordance with a general rule that applies to all strings
and depends on the input strings (and possibly internal states) for its
application, is a computing mechanism.** The mechanism's ability to perform
computations is explained mechanistically in terms of its components, their
functions, and their organization." [Piccinini, 2007c, 516, my emphasis]

These are almost the same, but there is a subtle difference between them.

9.5.1 Definition P1

Let's begin with Definition P1. It implies that a computer is any "system" (presumably,
a physical device, because only something physical can actively "perform" an action)
that manipulates strings of digits, that is, that "generate[s] output strings from input
strings in accordance with general rules that apply to all input strings and [that]
depend on the inputs (and sometimes internal states) for their application". What kind
of "general rule"? In [Piccinini, 2008, 37], he uses the term 'algorithm' instead of
'general rule'. This is consistent with the view that a computer is a Turing machine,
and explicates Hayes's "magic trick" as being an algorithm.

The crucial point, according to Piccinini, is that the inputs and outputs must be
strings of digits. This is the significant difference between (digital) computers and
"analog" computers: The former manipulate strings of digits; the latter manipulate
"real variables".

For Piccinini,

A *digit* is a particular [that is, a particular object or component of a device]
or a discrete state of a particular, discrete in the sense that it belongs to
one (and only one) of a finite number of types.... A *string* of digits is
a concatenation of digits, namely, a structure that is individuated by the
types of digits that compose it, their number, and their ordering (i.e., which
digit token is first, which is its successor, and so on) [Piccinini, 2007b,
107].[9]

In [Piccinini, 2007c, 510], he observes that a digit is analogous to a letter of an alphabet,
so they are like Turing's symbols that can be printed on a Turing machine's tape.
And

real variables are physical magnitudes that (i) *vary* over time, (ii) (are
assumed to) take a *continuous range of values* within certain bounds,
and (iii) (are assumed to) vary *continuously* over time. Examples of real

---

[9]In the other two papers in his trilogy, Piccinini gives slightly different characterizations of what a digit
is, but these need not concern us here; see [Piccinini, 2007c, 510], [Piccinini, 2008, 34].

variables include the rate of rotation of a mechanical shaft and the voltage level in an electrical wire [Piccinini, 2008, 48].

So far, so good. Neither Searle nor Hayes should be upset with this characterization.

9.5.2 Definition P2

But Piccinini's second definition adds a curious phrase. This definition implies that a computer is any system "whose correct mechanistic explanation ascribes to it the function of" manipulating digit strings according to algorithms. What is the import of that extra phrase?

It certainly sounds as if this is a weaker definition. In fact, it sounds a bit Searlean, because it sounds as if it is not the case that a computer *is* an algorithmic, digit-string manipulator, but rather that it is anything that can be so *described* by some kind of "mechanistic explanation". And that sounds as if being a computer is something "external" and not "intrinsic". So let's consider what Piccinini has in mind here. He says:

> Roughly, a mechanistic explanation involves a partition of a mechanism into parts, an assignment of functions and organization to those parts, and a statement that a mechanism's capacities are due to the way the parts and their functions are organized. [Piccinini, 2007c, 502]

As explained in §14.4 and §19.5.3.2, syntax is the study of the properties of a collection of objects and the relations among them. If a "mechanism" is considered as a collection of its parts, then Piccinini's notion of a mechanistic explanation sounds a lot like a description of the mechanism's syntax. But syntax, you will recall, is what Searle says is not intrinsic to a system (or a mechanism).

So how is Piccinini going to avoid a Searlean "slippery slope" and deny that everything is a computer? One way he tries to do this is by suggesting that even if a system can be analyzed syntactically in different ways, only one of those ways will help us understand the system's behavior:

> Mechanistic descriptions are sometimes said to be perspectival, in the sense that the same component or activity may be seen as part of different mechanisms depending on which phenomenon is being explained. . . . For instance, the heart may be said to be for pumping blood as part of an explanation of blood circulation, or it may be said to be for generating rhythmic noises as part of an explanation of physicians who diagnose patients by listening to their hearts. This kind of perspectivalism does not trivialize mechanistic descriptions. Once we fix the phenomenon to be explained, the question of what explains the phenomenon has an objective answer. This applies to computations as well as other capacities of mechanisms. A heart makes the same noises regardless of whether a physician is interested in hearing it or anyone is interested in explaining medical diagnosis. [Piccinini, 2007c, 516]

Let's try to apply this to Searle's "Wordstar wall": From one perspective, the wall is just a wall; from another, according to Searle, it can be taken as an implementation of Wordstar. Compare this to Piccinini's claim that, from one perspective, a heart is a pump, and, from another, it is a noisemaker. If you're a doctor interested in hearing the heart's noises, you'll consider the heart as a noisemaker. If you're a doctor interested in making a medical diagnosis, you'll consider it as a pump. Similarly, if you're a house painter, say, you'll consider the wall as a flat surface to be colored, but if you're Searle, you'll try to consider it as a computer program. (Although I don't think you'll be very successful in using it to write a term paper!)

9.6 The Story So Far

So, what is a computer? It would seem that almost all proposed definitions agree on at least the following:

- Computers are physical devices; they interact with other physical devices in the world.

- They manipulate (physical) symbols (strings of digits), converting some into others according to algorithms.

- They are physical implementations of Turing machines in the sense that their input-output behavior is logically equivalent to that of a Turing machine (even though the details of their processing might not be). A slight modification of this might be necessary to avoid the possibility that a physical device might be considered to be a computer even if it doesn't compute: We probably want to rule out "real magic", for instance. So perhaps we need to say that a computer must be a physical implementation of something that is logically equivalent to a Turing machine (or a lambda-calculus machine, or a recursive-function machine, etc.).

9.7 What Else Might Be a Computer?

Does such a definition include too much? Let's assume for a moment that something like Piccinini's reply to Searle carries the day, so that it makes sense to say that not everything is a computer. Still, might there be some things that intuitively aren't computers but that turn out to be computers on even our narrow characterization?

This is always a possibility. Any time that you try to make an informal concept precise, you run the risk of including some things under the precise concept that didn't (seem to) fall under the informal concept. You also run the risk of *excluding* some things that did. One way to react to this situation is to reject the formalization, or else to refine it so as to exclude what shouldn't have been included and/or to include what was erroneously excluded. But another reaction is to bite the bullet and agree to the new inclusions and exclusions: For instance, you might even come to see that something that you didn't think was a computer really was one.

9.7.1 Is a Brain a Computer?

Many people claim that the (human) brain is a computer. Searle thinks it is, but only because he thinks that everything is a computer. But perhaps there is a more interesting way in which the brain is a computer. Certainly, contemporary computational cognitive science uses computers as at least a metaphor for the brain. Before computers came along, there were many other physical metaphors for the brain: The brain was considered to be like a telephone system or like a plumbing system (see, e.g., [Squires, 1970, Sternberg, 1990, Gigerenzer and Goldstein, 1996, Angier, 2010], and "The Mind is a Metaphor" website, accessed 31 January 2014 from http://mind.textdriven.com/?q=mind [concerning that website, see [Guernsey, 2009]]).[10]

In fact, "computationalism" is sometimes taken to be the view that the brain *is* a computer or that the brain computes or that brain states and processes are computational states and processes:[11]

> The basic idea of the computer model of the mind is that the mind is the program and the brain the hardware of a computational system [Searle, 1990, 21].

> The core idea of cognitive science is that our brains are a kind of computer.... Psychologists try to find out exactly what kinds of programs our brains use, and how our brains implement those programs. [Gopnik, 2009a, 43].

> Computationalism... is the view that the functional organization of the brain (or any other functionally equivalent system) is computational, or that neural states are computational states [Piccinini, 2010a, 271] (see also [Piccinini, 2010a, 277–278]

But if one of the essential features of a computer is that it carries out computable processes by computing rather than (say) by magic or by some other biological but non-computable technique, then it's at least logically possible that the brain is not a computer even if brain processes are computable. How can this be? A process is computable if and only if there are one or more algorithms that specify how that process can be carried out. But it is logically possible for a process to be computable in this sense without actually being computed. Hayes's magic paper is a logically, if not physically, possible example. Another example might be the brain itself. Piccinini has argued [Piccinini, 2005], [Piccinini, 2007a] that neuron firings (more specifically, "spike trains"—that is, sequences of "action potential"—in groups of neurons) are not representable as digit strings. But, because Piccinini believes that a device is not a computer unless it manipulates digit strings, and, because it is generally considered that human cognition is implemented by neuron firings, it follows that the brain's cognitive functioning—even if comput*able*—is not accomplished by computa*tion*—is not accomplished by computa*tion*.

---

[10]There are also analogies between telephone systems and computers themselves [Lewis, 1953].

[11]Sometimes it is taken to be the view that the *mind* is a computer; see the quotations in [Rapaport, 2012b, §2].

Yet, if cognitive functions are computable (as contemporary cognitive science suggests [Edelman, 2008]), then there would still be algorithms that compute cognition, even if the brain doesn't do it that way. We'll return to this theme in Chapter 19.

9.7.2 Is the Universe a Computer?

multi-core probabilistic processor 10^{93} GHz

GPU core clock 1.86×10^{34} GHz

10^{80} GB SDRAM (expandable to 10^{110} GB with gravitation)

Genuine Windows Standard Model preinstalled

The Ultimate Gaming Computer
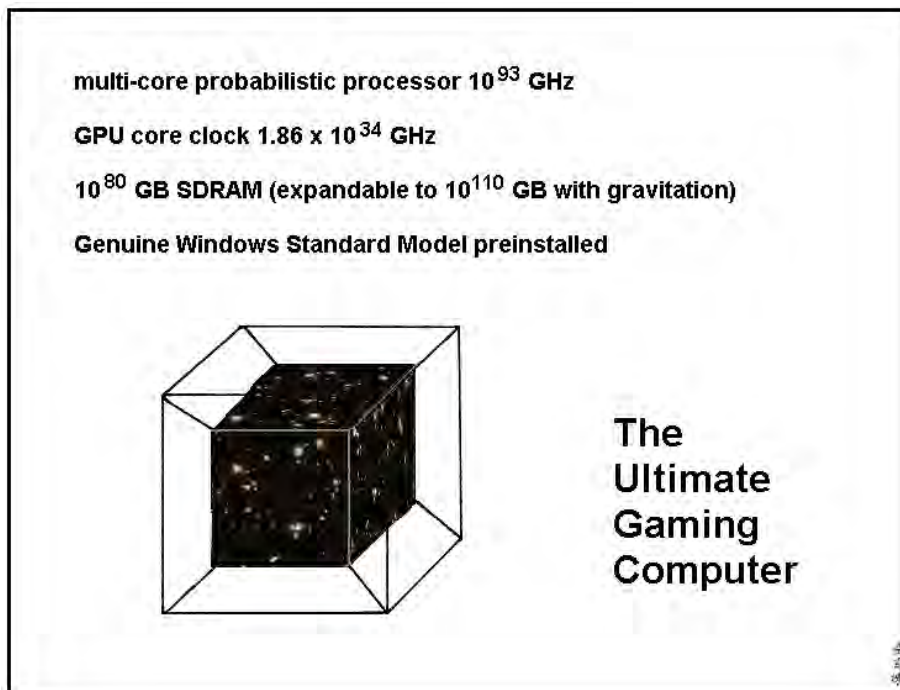
Figure 9.3: *Abstruse Goose*, http://abstrusegoose.com/219

Might the universe itself be a computer?[12] Consider Kepler's laws of planetary motion. Are they just a computable theory that describes the behavior of the solar system? If so, then a computer that calculates with them might be said to simulate the solar system in the same way that any kind of program might be said to simulate a physical (or biological, or economic) process, or in the same way that an AI program might be said to simulate a cognitive process. (We'll return to this idea in Chapter 15.)

Or does the solar system itself compute Kepler's laws? If so, then the solar system would seem to be a (special purpose) computer (that is, a kind of Turing machine).

---

[12]In addition to the cartoon in Fig. 9.3, see also "Is the Universe a Computer?", http://abstrusegoose.com/115 (best viewed online!).

The second possibility does not necessarily follow from the first. As we just saw in the case of the brain, there might be a computational theory of some phenomenon, but the phenomenon itself need not behave computationally. Here are some other examples: Someone might come up with a computational theory of the behavior of the stock market, yet the actual stock market's behavior is determined by the individual decisions made by individual investors. Calculations done by slide rules are done by analog means, yet the calculations themselves are clearly computable.

> Indeed, computational algorithms are so powerful that they can simulate virtually any phenomena, without proving anything about the computational nature of the actual mechanisms underlying these phenomena. Computational algorithms generate a perfect description of the rotation of the planets around the sun, although the solar system does not compute in any way. In order to be considered as providing a model of the mechanisms actually involved, and not only a simulation of the end-product of mechanisms acting at a different level, computational models have to perform better than alternative, non computational explanations. [Perruchet and Vinter, 2002, §1.3.4]

Another argument along these lines has been put forth by Stephen Wolfram, developer of the Mathematica computer program, in [Wolfram, 2002] CITE OTHER DOCS FROM whatisacomputer.html; CITE ZUSE, FREDKIN. Wolfram argues as follows:

1. Nature is discrete.

2. Therefore, possibly it is a cellular automaton.

3. There are cellular automata that are equivalent to a Turing machine.

4. Therefore, possibly the universe is a computer.

There are a number of problems with this argument. First, why should we believe that nature (that is, the universe) is discrete? Presumably, because quantum mechanics says that it is. But some distinguished physicists deny this [Weinberg, 2002]. So, at best, for those of us who are not physicists able to take a stand on this issue, Wolfram's conclusion has to be conditional: If the universe is discrete, then possibly it is a computer.

So, let's suppose (for the sake of the argument) that nature is discrete. Might it be a "cellular automaton? The easiest way to think of a cellular automaton is as a two-dimensional Turing-machine tape. (For more information on cellular automata, see [Burks, 1970]. Burks was one of the people involved in the construction of ENIAC and EDVAC.) But, of course, even if a discrete universe *might* be a cellular automaton, it *need not* be. If it isn't, the argument stops here. But, if it is, then the third premise is reasonable. The conclusion appears to follow validly from the premises, of which premise 2 is the one most in need of justification. But even if all of the premises and (hence) the conclusion are true, it is not clear what philosophical consequences we are supposed to draw from this.

9.8 Exercises for the Reader

(The material in this section was developed by Albert Goldfain.)

The following arguments are interesting to think about in relation to the question whether everything a computer. Try to evaluate them.

Argument 1 P1 A Turing machine is a model of computation based on what a single human (i.e., a clerk) does.

    P2 Finite automata and push-down automata are mathematical models of computation that recognize regular languages and context-free languages, respectively.

    P3 Recognizing strings in a languages is also something individual humans do.

    C1 Thus Turing machines, finite automata, and push-down automata are all models of computation based on the abilities of individuals.

Argument 2

John Conway's "Game of Life" is a cellular-automaton model (albeit a very simplistic one) of a society:[13]

    P1 The Game of Life can be implemented in Java.

    P2 Any Java program is reducible to a Turing-machine program.

    C1 Thus, the Game of Life is Turing-machine-computable

Argument 3 P1 The Game of Life can be thought of as a model of computation.

    P2 The Game of Life is a model of the abilities of a society.

    P3 The abilities of a society exceed those of an individual.

    C1 The abilities of a model of computation based on a society will exceed the abilities of a model based on the abilities of an individual.

    C2 It is not the case that every Turing-machine program could be translated to a Game-of-Life "computation".

Some of these arguments may have missing premises! To determine whether the Game of Life might be a model of computation, do a Google search on: "game of life" "turing machine"

Given an integer input (remember: everything can be encoded as an integer), how could this integer be represented as live cells on an initial grid? How might "stable" structures (remember: a 2×2 grid has 3 neighbors each) be used as "memory"? How would an output be represented?

Can Turing-machine programs be reduced to Game-of-Life computations?

---

[13]See http://en.wikipedia.org/wiki/Conway_Game_of_Life for the rules of this game.

9.9 NOTES FOR NEXT DRAFT

1. Comments on pencil-computer cartoons:
   A pencil is really only an *output* device (it prints and erases). To turn it into a full-fledged computer (or, at least, a physical Turing machine), you need to add eyes (for input), hands (for moving left and right), and a mind (for "states of mind").

2. THERE ARE MANY OTHER COMMENTS TO BE MADE ABOUT PAT HAYES: ON VIRTUAL MACHINES, ON O-MACHINES. DO LATER?

3. For §9.6:
   SEE EXTRA MATERIAL IN LECTURE NOTES AT PP 89/77/ AND SHEETS 1-2

   IS THERE ANYTHING MORE TO SAY ABOUT PICCININI?

   HOW DOES PICCININI (OR SEARLE OR HAYES) HANDLE INTERACTIVE COMPUTING?

   DISCUSSION WITH STU:

   - COMPUTERS ARE PHYSICAL OBJECTS THAT INTERACT WITH THE WORLD.
   - IF Turing machineS DON'T DO THAT, SO MUCH THE WORSE FOR THEM AS A MODEL OF WHAT A COMPUTER IS.
   - COMPUTERS AREN'T Turing machineS; THEY CAN BE MODELED BY Turing machineS.
   - COMPUTERS TAKE INPUT FROM THE WORLD. THEY MIGHT BE MODELED AS HAVING THAT INPUT PRESTORED, BUT THAT'S AN EXTENSIONAL, NOT AN INTENSIONAL MODEL.

4. All modern general-purpose digital computers are physical embodiments of the same logical abstraction[:] Turing's universal machine. [Robinson, 1994, 4–5]

5. Lloyd:

   [Lloyd and Ng, 2004] argues as follows:

   (a) Nature is discrete. (This is "the central maxim of quantum mechanics" (p. 54).)

   (b) In particular, elementary particles have a "spin axis" that can be in one of two directions.

   (c) ∴ they encode a bit.

(d) ∴ elementary particles store bits of information.

(e) Interactions between particles can flip the spin axis;
this transforms the stored data—that is, these interactions are operations on the data.

(f) ∴ (because any physical system stores and processes information) all physical systems are computers.

(g) In particular, a rock is a computer.

(h) Also, the entire universe is a computer.

Premise 5a matches Wolfram's fundamental premise and would seem to be a necessity for anything to be considered a digital computer. The next four premises also underlie quantum computing.

But the most serious problem with Lloyd's argument as presented here is premise 5f. Is the processing sufficient to be considered to be Turing-machine-equivalent computation? Perhaps; after all, it seems that all that is happening is that cells change from 0s to 1s and vice versa. But that's not all that's involved in computing (or is it?). What about the control structures—the grammar—of the computation?

And although Lloyd wants to conclude that everything in the universe (including the universe itself!) is a computer, note that this is not exactly the same as Searle's version of that claim. For Searle, everything can be interpreted as *any* computer program. For Lloyd, anything is a computer, "although they may not accept input or give output in a form that is meaningful to humans" (p. 55). So, for Lloyd, it's not a matter of intepretation. Moreover, "analyzing the universe in terms of bits and bytes does not replace analyzing it in conventional terms such as force and energy" (p. 54). It's not clear what the import of that is: Does he mean that the computer analysis is irrelevant? Probably not: "it does uncover new and surprising facts" (p. 54), though he is vague (in this general-audience magazine article) on what those are. Does he mean that there are different ways to understand a given object? That is true, but unsurprising (animals, for instance, can be understood as physical objects satisfying the laws of quantum mechanics as well as being understood as biological objects; and Dennett's intentional-stance theory also spells out different ways in which things can be understood). Does he mean that force and energy can, or should, be understood in terms of the underlying computational nature of physical objects? He doesn't say.

But he does end with a speculation on what it is that the universe is computing, namely, itself! Or, as he puts it, "computation is existence" (p. 61). As mystical as this sounds, does it mean anything different from the claim that the solar system computes Kepler's Law?

And here's an interesting puzzle for Lloyd's view, relating it to issues concerning whether a computer must halt (see the earlier discussion in WHERE?):

> [Assuming the universe is computing its own evolution. . . , does it have a finite lifetime or not? If it is infinite, then its self-computation

> won't get done; it never produces an answer.... Hence, it does not
> qualify as a computation. [Borbely, 2005, 15]

Related to Lloyd is Bostrom's work on whether we are living in a computer simulation. If Lloyd is right, then the universe is a computer, and we are data structures in its program, brought to life as it were by its execution. If Bostrom is right, then we are data structures in someone else's program. If theists (computational theists?) are right, then we are data structures in God's program. Are there significant differences? See [Bostrom, 2003].

6.   [A] computer... can be looked at from two different angles, which
      Professor Hartree [[Hartree, 1949, 56]] has called the "anatomical"
      and the "physiological," that is, "of what is it made?" and "how does
      it tick?" [Samuel, 1953, 1223]

Samuel then goes on to describe the anatomy in terms of things like magnetic cores and vacuum tubes. Clearly, the anatomy has changed since then, so defining 'computer' in those terms doesn't seem to be the right way to go: It's too changeable. What's needed is a physiological, or functional, definition. Thus, a computer should be defined as a physical device (where it doesn't matter what the physical device is made of) that does such and such.

But is it even correct to limit a computer to a *physical* device? We want to distinguish a "real" computer from a mathematical abstraction such as a Turing machine. But what about a *virtual* computer implemented in some software, such as a program that "simulates" a computer of a certain type but that runs on a (physical) computer of a very different type? Note that, ultimately, there is a physical substrate, however. (For more on this idea, see [Pylyshyn, 1992].)

7. [Samuel, 1953, 1223]'s functional definition: A computer is "an information or data processing device which accepts data in one form and delivers it in an altered form." This seems to be very—even *too*—high level a description. It omits any mention of computation, of algorithms. It does mention that the "delivered" data must have been "processed" from the "accepted" data by the "device"; so it's not just a function that relates the two forms of data—it's more of a function machine. But there's no specification of the kind of processing that it does.

Partly because of this, and on purpose, it also doesn't distinguish between analog and digital computers. He resolves this by adding the modifier 'digital', commenting that "Any operation which can be reduced to arithmetic or to simple logic can be handled by such a machine. There does not seem to be any theoretical limit to the types of problems whch can be handle in this way" [Samuel, 1953, 1224]—a nod, perhaps, to our binary-representation insight (§7.6.1. Still, this doesn't limit the processing to algorithmic processing.

It does, however, allow the brain to be considered as a computer: "when the human operator performs a reasonably complicated numerical calculation he [sic][14] is forcing his brain to act as a digital computer" [Samuel, 1953, 1224].

---

[14]The use of the male gender here is balanced by Samuel's earlier statement that computers have

A bit later (p. 1225), he does say that the processing must must be governed by rules; this gets closer to the notion of an algorithm, though he (so far) puts no constraints on the rules.

It is only after he discusses the control unit of the computer and its programming (pp. 1226ff) that he talks about the kinds of control structures (loops, etc.) that are involved with algorithms.

So, perhaps we could put all of this together and say that, for Samuel, a (digital) computer is a physical device that algorithmically processes digital data

Further on, he adds the need for input and output devices (p. 1226). Are these really needed? Are they part of the abstract, mathematical model of a computer, namely, a Turing machine? Your first reaction might be to say that the tape serves as both input and output device. But the tape is an integral part of the Turing machine; it is really more like the set of internal switches of a physical computer, whereas physical computers normally have input and output devices (think of keyboards and monitors) as separate, additional components: Think of a computer like the Mac Mini, which is sold without a keyboard and a monitor. (This is surely related to issues that we have, or will, discuss elswhere, including Knuth's observations about the necessity of output but the non-necessity of input (see our discussion of this in §7.5) and our discussion of the relation of computers to the real world, which we begin in Ch. 10.)

8. This may not be relevant to the present book, but, in line with an observation I made in [Rapaport, 2012b] about the output of an algorithm being an interpretation of its input, [Samuel, 1953, 1225] says:

> we might call the computer a mathematical translator in the same sense that a literary translator takes information in one language without adding or subtracting any basic information and renders this information intelligible to someone not understanding the original language.

9. On analog vs. digital computers:
Cite [Samuel, 1953, 1224, § "The Analogue Machine"] and [Rubinoff, 1953].

And Shagir 1999.

And [Copeland, 1997, "Nonclassical Analog Computing Machines", pp. 699–704].

10. Another argument that brains are computers:

   (a) (Physical) computers are physical symbol sytems, in the sense of [Newell and Simon, 1976].

   (b) "Humans are physical symbol systems" [Newell, 1980, 174]

"advantages in terms of the reductions in clerical manpower *and woman power* [Samuel, 1953, 1223, my italics].

(c) ∴ "there exists a physical architecture ["a neural organization"] that supports that symbol system." [Newell, 1980, 174]

Now, this doesn't quite do it. First, we need the converse of the second premise: All physical symbol systems are computers. (Is that in [Newell and Simon, 1976] or [Newell, 1980]? Maybe!) Let's grant that. What about the conclusion? Piccinini (CITE) argues that that neural architecture doesn't compute. So, must a computer compute? Or could a computer be any device that is extensionally (that is, input-output) equivalent to a computer-that-computes?

11.   There are many ways to describe a computing machine; they range from Alan Turing's classic account of a device for reading from and writing to an infinite tape to the blueprints for the mechanism of a contemporary supercomputer. Most such accounts are formally equivalent to one another in the sense that what they describe can solve the same classes of problems. [Hayes, 1990, 8]

Surely, a Turing machine is more than merely a input-output device: There has to be some *computing* that goes on in between!

Here I want to take a highly abstract view of what constitutes a computer. It is to be a device built out of objects drawn from three realms: data, instructions and addresses. ... Computing requires interactions among the three kinds of objects. The most obvious interactions are the ones in which instructions act on data. [Hayes, 1990, 8]

And, to be a computer at least in the sense of something equivalent to a Turing machine, the nature of the instructions, and the way that they act on data, must satisfy the constraints that Turing imposed (or ones logically equivalent to them).

Another essential category of interactions reverses the flow of information: data act on instructions. Such an event is generally called a conditional branch. ... conditional branching provides a feedback channel, carrying information from the realm of data back to the realm of instructions. Without such decision-making power a computer would be crippled. It would always execute a fixed sequence of instructions, oblivious to the data it was acting on. [Hayes, 1990, 8–9]

This is an interesting insight! If the data are considered to be external to the computer, then it is a way in which the external world acts on, or influences, the computer. Note that the *source* of the data is not mentioned. If the data is already written on the tape, that is, if the data is part of the pre-determined input, then we have a Turing machine. But if the data could come from the external world in the middle of a computation, then we would seem to have an interactive model of computation, which, it is claimed, is hypercomputation. Note, however, that

what the computer (or hypercomputer?) actually does is no different in either case: It prints and moves (it sets switches).

Hayes's third component—addresses—adds something that classical Turing machines lack, because the cells of a Turing machine have no addresses. Does this add any kind of computing power? Probably not: Just replace every tape cell by two cells, one of which contains an address for the other one. There are details to be worked out, but it would seem that addresses only make certain computations more efficient; they wouldn't allow non-computable functions to be computed.

12.   We count something as a computer because, and only when, its inputs and outputs can usefully and systematically be interpreted as representing the ordered pairs of some function that interests us. …This means that delimiting the class of computers is not a sheerly empirical matter, and hence that "computer" is not a natural kind. …Similarly, we suggest, there is no intrinsic property necessary and sufficient for all computers, just the interest-relative property that someone sees value in interpreting a system's states as representing states of some other system, and the properties of the system support such an interpretation. …[I]n this very wide sense, even a sieve or a threshing machine [or an eggbeater!?] could be considered a computer.… [Churchland and Sejnowski, 1992, 65–66]

This is essentially Searle's point, only with a positive spin put on it. Note that their definition in the first sentence has an objective component (the inputs and outputs must be computationally related; note, too, that no specification is placed on whether the mechanism by which the inputs are transformed into the outputs is a computational one) and a subjective component (if the function computed by the alleged computer is of no human interest, then it is not a computer!).

Thus, this is a bit different from Searle: Where Searle says that the wall behind me is (or can be interpreted as) a word processor, Churchland & Sejnowski say that the wall behind me is computing something, but we don't care what it is, so we don't bother considering it to be a computer.

Both Searle as well as Churchland & Sejnowski consider the brain to be a computer. On their definition of computer, it's not *how* the outputs are produced by the inputs that matters, only *that* the outputs can be *computed* by the inputs.

So we seem to have two definitions of 'computer', a weak and a strong one:

A strong computer is a physical device that causally transforms inputs into outputs using Turing-machine-equivalent mechanisms.

A weak computer is a physical device that causally transforms inputs into computable outputs by some mechanism that is not Turing-machine-equivalent.

Perhaps there is a weaker notion too: A look-up table which doesn't even have a mechanism (they allude to this in their §2 (pp. 69–74).

And there is yet another notion: Analog computers, such as slide rules (which they discuss on p. 70), use non-computational mechanisms.

13. On Searle's argument:

> [A]n objection to Turing's analysis...is that although Turing's account may be necessary it is not sufficient. If it is taken to be sufficient then too many entities turn out to be computers. The objection carries an embarrassing implication for computational theories of mind: such theories are devoid of empirical content. If virtually anything meets the requirements for being a computational system then wherein lies the explanatory force of the claim that the brain is such a system? [Copeland, 1996, §1, p. 335].

So, x is a computer iff x is a (physical) model of a Turing machine. To say that this "account" is "necessary" means that, if x is a computer, then it is a model of a Turing machine. That seems innocuous. To say that it is a "sufficient" account is to say that, if x is a model of a Turing machine, then it is a computer. This is allegedly problematic, because, allegedly, anything can be gerrymandered to make it a model of a Turing machine; hence, anything is a computer (including, for uninteresting reasons, the brain).

Interestingly, Copeland *defends* the sufficiency thesis (in this paper) but *rejects* the necessity thesis (elsewhere, on hypercomputation grounds).

Copeland puts a constraint on algorithms that sounds right but that others (see our discussion in §7.5) leave implicit: Algorithms are "specific to an architecture" that specifies what the primitive operations are:

> Thus an algorithm at least one of whose instructions calls explicitly for a multiplication cannot be specific to an architecture that has addition but not multiplication available as a primitive. A program calling for multiplications can run on such an architecture only because the compiler (or equivalent) replaces each multiplication instruction in the program by a series of addition instructions. [Copeland, 1996, §2, p. 337]

Copeland then suggests a definition of what it means for an entity e to compute a function f: e computes f iff there is "a labelling scheme L" and an algorithm+architecture A[15] for computing f such that $\langle e, L \rangle$ is a model of A, where a labeling scheme is, roughly, a mapping from symbols to parts of e.

He then considers "whether the solar system computes...Kepler's law" or "whether the universe is a computer dedicated to computing its own behaviour", citing Fodor LOT p.74. (See §9.7.2, above.) His answer is that it doesn't, for one reason because Kepler's law isn't an algorithm and for another because any algorithm that computed Kepler's law would require a specific architecture that doesn't match that of the solar system.

Copeland rejects this definition. His rejection begins with what he calls "Searle's Theorem":

---

[15]Copeland calls it 'SPEC'.

> For any entity *x* (with a sufficiently large number of discriminable parts) and for any architecture-algorithm specification *y* there exists a labelling scheme *L* such that ⟨*x*, *L*⟩ is a model of *y*. [Copeland, 1996, §2, p. 339]

In §4, he proves this theorem. The proof is beyond our scope, so let us accept it for the sake of the argument. The problem with the Theorem, according to Copeland, is that the proof only works for a "nonstandard" interpretation of *x*, where by a "nonstandard" interpretation Copeland means an interpretation that makes all statements come out true but is not about the entities that the standard interpretation is about, something like saying that if 'Moscow' is non-standardly interpreted to mean the number 16, 'London' is non-standardly interpreted to mean the number 1, and 'is north of' is non-standardly interpreted to mean "is greater than", then 'Moscow is north of London' is true, but is not about Moscow or London.

So, he refines Searle's Theorem to this: *e* computes *f* iff there is an *L* and an *A* such that ⟨*e*, *L*⟩ is an *honest* model of *A*, where a model is "honest" iff it is not non-standard. [Copeland, 1996, §5, p. 348]

But, he argues, from this it does not follow that, for any *x*, *x* is a computer. In particular, and contra Searle, it does not follow that the brain is a computer. Nor does it follow that the brain is not a computer. All we can ask is whether there is a way to "label" the brain so that we can say "honestly" that it computes, and that is an empirical question, not a trivial one. [Copeland, 1996, §8, p. 357]

14. On the brain as computer:
    Searle is not the only one wondering if the brain is a computer. [Churchland and Sejnowski, 1992] and [Ballard, 1997] have both written books about whether, and in what ways, the brain might be a computer. As [Ballard, 1997, 1–2] puts it, "The key question...is, Is computation sufficient to model the brain?". One reason this is an interesting question is that researchers in vision have wondered "how...an incomplete description, encoded within neural states, [could] be sufficient to direct the survival and successful adaptive behavior of a living system" [Richards, 1988, as cited in Ballard 1997, p. 2]. If a computational model of this ability is sufficient, then it might also be sufficient to model the brain. And this might be the case even if, as, for example, Piccinini CITE argues, the brain itself is *not* a computer, that is, does not behave in a computational fashion. A model of a phenomenon does not need to be identical in all respects to the phenomenon that it models, as long as it serves the purposes of the modeling. But Ballard also makes the stronger claim when he says, a few pages later, "*If the brain is performing computation*, it should obey the laws of computational theory" [Ballard, 1997, 6, my italics]. But whether the brain performs computations is a different question from whether its performance can be modeled or described in computational terms. So, the brain doesn't have to *be* a computer in order for its behavior to be describable computationally. As [Churchland and Sejnowski, 1992] note, whether the brain *is* a computer—

whether, that is, the brain's functioning satisfies one of the (logically equivalent) characterizations of computing—is an empirical issue.

What about the wall behind me? Presumably, it, too, doesn't have to be a computer in order for its (molecular or subatomic) behavior to be describable computationally. Or is Searle making a stronger claim, namely, that, not only is its behavior *describable* computationally, but it *is* a computation?

[Ballard, 1997, 11] has an interesting variation on that stronger claim:

> Something as ordinary as a table might be thought of as running an algorithm that adjusts its atoms continaually, governed by an energy function. Whatever its variables are, just denote them collectively by x. Then you can think of the table as solving the problem of adjusting its atoms so as to minimize energy, that is, $\min_x E(x)$. Is this computation?

Note that this is different from Searle's claim that the table (or a wall) might be computing a word processor. It seems closer to the idea that the solar system might be computing Kepler's law. (See discussion on this elsewhere.)

15. Another claim in the vicinity of Searle's and Ballard's concerns DNA computing:

> Computer. The word conjures up images of keyboards and monitors. …But must it be this way? The computer that you are using to read these words [that is, your brain!—comment by WJR] bears little resemblance to a PC. Perhaps our view of computation is too limited. *What if computers were ubiquitous and could be found in many forms? Could a liquid computer exist in which interacting molecules perform computations?* The answer is yes. This is the story of the DNA computer. [Adleman, 1998, 54, my italics]

Of course, Adelman is not making the Searlean claim that everything is a computer and that, therefore, the interacting molecules of (any) liquid peform computations. Nor is he making the Ballardian claim that DNA computes in the way that a table computes. (Others have, however, made such a claim, on the grounds that strands of DNA are similar to Turing-machine tapes with a four symbols instead of two and with the processes of DNA transcription and recombination as being computable processes. CITE) Rather, Adleman's claim is that one can use DNA "to solve mathematical problems". However, contrary to what the editors of *Scientific American* wrote in their subtitle to Adleman's article, it is unlikely that "is redefining what is meant by 'computation' ". After all, the advent of transistors did not change Turing's mathematical characterization of computing any more than the use of vacuum tubes did. At most, DNA computers might change what the lay public means by 'computer'. But, as we have seen/will see elsewhere (WHERE?), that has already happened, with the meaning changing from "humans" to "computing machines".

16. On computers as switch-changing devices:
    Advocates of both DNA and quantum computing focus on the idea that computers are switch-changers. First, from the quantum perspective:

    > Boiled down to its essentials, any computer must meet two requirements: it must be able to store information as strings of 1's and 0's, ro bits, and it must have a way of altering the bits in accordance with instructions. [Grover, 1999, 25]

    Actually, it need not store just 1s and 0s, and, of course, it doesn't store numbers: It stores numerals ('1's, not 1s). Furthermore, the instructions must satisfy the constraints on algorithms.

    From the DNA perspective:

    > To build a computer, only two things are really necessary—a method of storing information and a few simple operations for acting on that information. ... The remarkable thing is that just about any method of storing information and any set of operations to act on that information are good enough. [Adleman, 1998, 37]

    Adleman is more correct about the storage method than Grover, but equally vague about the operations, if not worse: Surely, not "any" operations will do. (Maybe "just about" is the hedge that saves him; nonetheless, it would be nice to be a bit more accurate.)

17. Can we define a computer simply as any device that carries out an algorithm? [Davis, 2000] suggests (but does not explicitly endorse) this. I suppose that it depends on what 'carries out' means: Surely it has to include as part of its meaning that the internal mechanism of the device must operate in accordance with—must behave exactly like—one of the logically equivalent mathematical models of computation. Surely, any computer does that. Is anything that does that a computer? Is anything that can be gerrymanderedly described as doing that a computer?

18. Re Thomason:
    A computer is a set of registers with contents (or: switches with settings; if they are binary switches, each is either on or else off), some of which are interpreted (by whom? by what?) as data, some as program, and the rest as irrelevant (and some as output?). Computation changes the contents (the settings).

19. von Neumann's definition of 'computer':
    In his "First Draft Report on the EDVAC", which—along with Turing's 1936 paper—may be taken as one of the founding documents of computer science, [von Neumann, 1945, §1.0, p. 1] gives the following definition:

    > An *automatic computing system* is a (usually highly composite) device, which can carry out instructions to perform calculations of a considerable order of complexity.... ... The instructions... must

> be given to the device in absolutely exhaustive detail. They include all numerical information which is required to solve the problem under consideration.... ...All these precedures require the use of some code to express...the problem..., as well as the necessary numerical material.... ...[T]he device...must be able to carry them out completely and without any need for further intelligent human intervention. At the end of the required operations the device must record the rsults again in one of the forms referred to above.

Other comments (in this section of [von Neumann, 1945], as well as later, in §5.0 (pp. 6ff)) indicate that the code should be binary, hence that the computer is a "digital" device (§1.0, p. 1). This definition hues closely to being a physical implementation of a Turing machine, with clear allusions to the required algorithmic nature of the instructions, and with a requirement that there be both input and output (recall our discussion of the necessity of this—or lack thereof— in §7.5.

20. Naur 2007 (see Further Sources, below) argues that "the nervous system...has no similarity whatever to a computer". His argument is difficult to follow, but that is another story. Even if true (and others, like Piccinini, think that it is), it raises another question: Does the nervous system compute? There are several different versions of this kind of question: We might say that a device *x* computes iff:

    (a) *x* behaves like a Turing machine(or anything that is logically equivalent to a Turing machine, such as a register machine), or else

    (b) *x* has the same input-output behavior as a Turing machine (even if it is "magical"), xor

    (c) *x* has the same input-output behavior as a Turing machine *and x* "does something" (even if what it does is not computation)

21. More on solar system computing Kepler's law:
    [Shagrir, 2006b, 394] raises an interesting question:

    > Computer models are often used to study, simulate, and predict the behavior of dynamical systems. In most cases, we do not view the modeled system, e.g., the solar system, as a computer. When studying the brain, however, our approach is different. In this case, in addition to using a computer model to simulate the system under investigation, i.e., the brain, we also take the modeled system itself to compute, viewing its dynamical processes as computing processes. Why is this so? What is it about the brain that makes us consider it, along with desktops and pocket calculators, a species of computer, when it doesn't even occur to us to accord that status to solar systems, stomachs and washing machines?

    Of course, it *has* occurred to several people, as we have seen.

Shagrir's bigger point is that, even if anything can be described (by us) as being a computer—even if "being a computer is a matter of perspective" (p. 399)—*discovering* how this can be for a given device (such as the brain) can give us useful information about the device in terms of what it represents and how it operates on those representations. And, I suppose, in other cases, this information is not deemed (at least at the current time) to be useful (as in the case of Searle's wall).

A further point of Shagrir's is that what makes a computational perspective on a device interesting has to do with semantics, not (just) syntax: "assigning content and information to brain/mental states is often *prior* to taking the computational outlook" (p. 402). So, semantics can guide us to look for an interpretation of a device and its activities that will turn out to be a computational interpretation:

> [T]he problems the brain computes are certain regularities or functions that are specified in terms of the representational content of the arguments (inputs) and values (outputs). I call these problems *semantic tasks*. ...It is also consistent with the computing technology tradition that calculators compute mathematical functions...chess machines compute the next move on the chess-board.... [Shagrir, 2006b, 403]

Recall Fodor's story of the chess computer that can be interpreted as computing a Civil War battle. There are two ways to describe a given device's (a given computer's) behavior: purely syntactically, or semantically, that is, solely in terms of the device's own, internal behavior (syntactically), but also in terms of its relations to other things (semantically). In some cases (Searle's wall), such a semantic interpretation is at best a curiosity. In others (the brain, the chess computer), it is highly useful and insightful. In still others (the Civil War interpretation of the chess computer), it may be serendipitously useful or insightful ("Wow, my chess computer does double duty; what a bargain!", or: "How interesting: This chess match has the same structure as this Civil War battle! (Grant not only metaphorically, but literally, checkmated Lee!)").

> We are now in a better position to say why we apply the computational approach to some systems, e.g., brains, but not others, e.g., planetary systems, stomachs and washing machines. ...We apply the computational approach when our goal is to explain a semantic pattern manifested by the system in question.... We do not view planetary systems, stomachs and washing machines as computers because the tasks they perform...are not defined in terms of representational content. Again, we could view them as computing semantic tasks.... But we don't.... [Shagrir, 2006b, 405]

That is, we apply the computational approach to explain intentional-stance descriptions of a system. But isn't explaining washing-machine behavior in terms of clothes-washing semantic? (Even functional: input dirty blue shirt;

output clean blue shirt.)[16] Well, yes, but we have not found that to be a useful or insightful perspective.

What makes a computational analysis more useful than some other analysis (say, a purely physical or mechanical analysis)? Here is Shagrir's answer:

> A computational theory in neuroscience aims to explain how the brain extracts information content G from another, F. We *know* from electrophysiological experiments that G is encoded in the electrical signals of a neural state N_n, and F in those of N_0. Let us also assume that we can track chemical and electrical processes that mediate between N_0 and N_n. What we still want to know is why the information content of the electrical signals of N_n is G, e.g. target location in headcentered coordinates, and not, say, the target's color. The question arises since the encoded information G are not "directly" related to the target, but is mediated through a representational state that encodes F. The question, in other words, is why a chemical and electrical process that starts in a brain state N_0 that encodes F, and terminates in another brain state, N_n, yields the information content G. After all, this causal process $N_0 \rightarrow N_n$ is not sensitive to what is being represented, but to chemical and electrical properties in the brain.
>
> A computational theory explains why it is G that is extracted from F by pointing to correspondence relations between mathematical or formal properties of the 27 representing states—what we call computational structure—and mathematical or formal properties of the represented states. The idea is that by showing that N_0 encodes F, and that the mathematical relations between N_0 and N_n correspond to those of the "worldly" represented states that are encoded as F and G, we have shown that N_n encodes G.

Note the similarity of this to classic philosophical arguments about dualism (our mental thoughts have no causal influence, because they are all implemented in physical mechanisms). But this is exactly right: It is the idea behind functionalism in the philosophy of mind: It is the functional, or computational—or what Shagrir is calling the semantic—level of analysis that tells us what it is that the physical level is doing.

22. Albert Goldfain (personal communication, 15 January 2008) suggests calling Searle's position 'pancomputationalism'. Others have had the same, or similar, ideas; see, e.g.,
http://philosophyofbrains.com/2007/06/26/pancomputationalism-origins-of-a-neologism.aspx

23.   Thus every organized body of a living thing is a kind of divine machine or natural automaton. It infinitely surpasses any artificial

---

[16]Missing socks might be an exception!

> automaton, because a man-made machine isn't a machine in every
> one of its parts. For example, a cognitive on a brass wheel has parts
> or fragments which to us are no longer anything artificial, and bear
> no signs of their relation to the intended use of the wheel, signs that
> would mark them out as parts of a machine. But Nature's machines—
> living bodies, that is—are machines even in their smallest parts,
> right down to infinity. That is what makes the difference between
> nature and artifice, that is, between divine artifice and our artifice.
> [Leibniz, 1714, §64]

This bears interesting comparison to Leibniz's famous "mill" passage, worth
discussing in Ch. 19. Presumably, the brain, as a natural (or divine) artifact
is infinitely complex, whereas a computer, as a human artifact, is not. Note
that a human-artifactual computer must "bottom out" in something that is not a
computer, just as recursive functions must bottom out in primitive operations. It
is unknown whether natural artifacts such as the brain bottom out: The brain is
made of neurons, etc., which are made of molecules, which are made of atoms,
which are made of subatomic particles, which are made of quarks, etc. We don't
know if that's the bottom level or if there is a bottom level. In any case, if such a
bottom level is a computer, as Lloyd argues, then the brain (as well as everything
else) is made of computers all the way down. Otherwise, it, too, bottoms out in
something non-computational.[17]

24. On the brain as a computer:

> If the brain is a machine that processes information, then its cognitive
> activity can be interpreted as a set of information processing states
> linking stimulus to response (i.e. as a mechanism or an algorithm).
> [Schyns et al., 2008, 20]

But that's not sufficient to be an algorithm. There are two issues: Does the
brain do anything that we can study? Does it compute? The first question
was answered negatively by the behaviorists. The original cognitivists (such as
[Miller et al., 1960]) answered positively *and* argued that the mechanism was
computational. But, as Piccinini CITE has argued (and as I pointed out in
§9.7.1), there might be an internal mechanism that is not computational. So, what
properly follows from the antededent of the above quotation is this: "…then the
brain is a computer iff the processing is—not merely comput*able*—but actual
computa*tion*. Merely begin able to process information is not sufficient for being
an algorithm.

To be fair, Schyns et al. argue that the information processing that the brain
does *is* computational, not merely computable. For instance, they claim
that it is possible to "translate [a] brain microstate…into [an] information
processing macrostate" [Schyns et al., 2008, 22; see also pp. 23–24], and that

---

[17]Thoughts inspired by discussion with Albert Goldfain, 15 January 2008.

such macrostates are computational states. However, I will leave the dispute between Piccinini and them for you to consider on your own.

25. [Newell, 1980, 174] also holds that the brain is a computer:

> There must exist a neural organization that is an architecture—i.e., that supports a symbol structure.

But this does not necessarily follow from a thesis that cognition is computable (which Newell certainly holds), because—as we have seen WHERE?—it might be possible that the neural organization of the brain is not computational even though its behavior is computable.

26. On the GCD discussion in §9.3.2:
See [Goodman, 1987, 484].

27. On intrinsic properties:
[Rescorla, 2014a] uses slightly different terms for similar ideas:

> *Inherited meanings* areise when a system's semantic properties are assigned to it by external observers, either through explicit stipulation or through tacit convention. Nothing about the system helps generate its own semantics. *Indigenous meanings* arise when a system helps generate its own semantics (perhaps with ample help from its evolutionary, desgin, or causal history, along with other factors). The system helps confer content upon itself, through its internal operations or its interactions with the external world. Its semantics does not simply result from external assignment. (p. 180)

28. [Aaronson, 2011] claims that quantum computing has "overthrown" views like those of [Wolfram, 2002] that "the universe itself is basically a giant computer...by showing that if [it is, then] it's a vastly more powerful kind of computer than any yet constructed by humankind."

29. On magic paper, see Ch. 14, Notes for Next Draft No. 8.

30.
> Is the brain a computer in this sense? Arguably. For a start, the brain can be "programmed" to implement various computations by the laborious means of conscious serial rule-following; but this is a fairly incidental ability. On a different level, it might be argued that learning provides a certain kind of programmability and parameter-setting, but this is a sufficiently indirect kind of parameter-setting that it might be argued that it does not qualify. In any case, the question is quite unimportant for our purposes. What counts is that the brain implements various complex computations, not that it is a computer. [Chalmers, 2011, §2.2, esp. p. 336]

There are two interesting points made here. The first is that the brain can simulate a Turing machine. The second is the last sentence: What really matters is that the brain can compute, not that it "is" a computer. To say that it is a computer

raises the question of what kind of computer it is: A Turing machine? A register machine? Something sui generis? And these questions seem to be of less interest than the fact that it can compute.

31. On the Searle-Putnam argument that anything implements every computation:

> . . . there is nothing approaching a *formal demonstration* that the structure-from-motion computation or vector subtraction. . . [that is, alleged semantic computations] do not have 'unintended' implementations. However, there is no reason to think that the possibility of such implementations renders the explanations that computational theories do provide trivial or otiose. To see why not we must again focus on actual practice.
>
> The epistemological context in which computational models are developed is instructive. We don't start with a computational description and wonder whether a given physical system implements it. Rather, we start with a physical system such as ourselves. More precisely, we start with the observation that a given physical system has some cognitive competence—the system can add, or it can understand and produce speech, or it can see the three-dimensional layout of its immediate environment. Thus, the explanandum of the theory is a manifest cognitive capacity. The computational theorist's job is to explain how the physical system accomplishes the cognitive task. There is, of course, no evidence that Putnam's rock or Searle's wall has *any* cognitive capacities. With the target capacity in view, the theorist hypothesizes that the system computes some well-defined function (in the mathematical sense), and spells out how computing this function would explain the system's observed success at the cognitive task.
>
> [Egan, 2012, 45]

There can't be any "formal demonstration" of the non-existence of "unintended implementations", because *any* abstraction can have multiple implementations (and some, such as the natural numbers as specified by Peano's axioms, can have an infinite number of implementations). What is important, however, is Egan's point that the existence of unintended implementations (such as, perhaps, Searle's wall that implements Wordstar or a computational theory of cognition) does not make Wordstar or a computational theory of cognition trivial. The second paragraph spells out the point that I make elsewhere (WHERE?) that we begin with a physical entity (a physical implementation), and subsequently derive an Abstraction from (that is, a theory of) it. From that Abstraction, other implementations can be constructed or found. If some of those other implementations are odd, that does not make the Abstraction of less importance. We can either bite the bullet and accept the unintended implementation, or try to refine the Abstraction. (The latter task, however, is unlikely to succeed, because the refined Abstraction will have its own unintended implementations.) As Egan

points out, if the Abstraction succeeds in explaining the *original* implementation, that's all that we need.

I would, however, disagree with one possible interpretation of the claim that "There is, of course, no evidence that Putnam's rock or Searle's wall has *any* cognitive capacities." I agree with this statement. But it should not be interpreted to mean that *no* intended or even unintended implementation has cognitive capacities.

And the reason for not drawing that stronger conclusion can be found in this passage:

> In order for the system to be used to compute the addition function these causal relations have to hold *at a certain level of grain*, a level that is determined by the discriminative abilities of the user. That is why, whatever the status of Putnams argument, no money is to be made trying to sell a rock as a calculator. Even if (*per mirabile*) there happens to be a set of state-types at the quantum-mechanical level whose causal relations do mirror the formal structure of the addition function, microphysical changes at the quantum level are *not* discriminable by human users, hence human users could not use such a system to add. (God, in a playful mood, could use the rock to add.) [Egan, 2012, 46]

In other words, even if Searle's wall implements Wordstar, *we* wouldn't be able to use it as such.

32. [Chalmers, 2012, 215–216] makes much the same point:

> On my account, a pool table will certainly implement various a-computations and perform various c-computations [that is, concrete computational processes]. It will probably not implement interesting computations such as algorithms for vector addition, but it will at least implement a few multi-state automata and the like. These computations will not be of much explanatory use in understanding the activity of playing pool, in part because so much of interest in pool are not organizationally invariant and therefore involve more than computational structure.

33. On the Searle-Putnam Puzzle:

> Why are we are so worried about liberalism? ... One concern, emphasized by Searle [Searle, 1990], is that liberalism makes the claim that the brain computes vacuous, in the sense that its truth-value is not a matter of facts about the brain ('computation is not discovered in the physics') ibid., p. [35]), but a matter of our whims ('computation is not an intrinsic feature of the world. It is assigned relative to observers', Searle 1992 [ADD TO BIB!], p. 212).

One can note that there must be something wrong with this objection. For there is a trivial sense in which everything represents, and yet we accept the claim that brains (or minds) represent very seriously. We do not tend to complain that it is vacuous. I think that the same is true for computation: everything is a computer in some trivial sense, and yet it does not follow that the claim that the brain computes is senseless. *The computational properties of the brain are discovered, and not assigned relative to an observer.* [Shagrir, 2010, 277, col. 1, my italics]

We can now turn to the second objection, of failing to draw the line between computing and non-computing systems. The reply is this: my account implies that **every system computes in the sense that it *could* be a computer, but it does not imply that it computes in the sense that it <u>is</u> one.** According to my account, an AM-computer is either (i) a 'natural' computer, in the sense that the simulation processes are defined over a natural system of representations, or (ii) a 'conventional' computer, in the sense that its simulation processes are defined over a conventional system of representations. Arguably, brains belong to the first category. Desktops, defined over representational systems of *type I*, and robots, defined over representational systems of *type II*, are confined to the second. Stomachs, tornadoes and washing machines are neither. Their processes are defined over neither natural nor conventional systems of representations. True, my account implies, as it should, that stomachs *could* be (conventional) computers: we could interpret its states as representing the states of some automaton and simulating its behavior. But we won't, and so stomachs don't compute. [Shagrir, 2010, 277, col. 2, my boldface and underlined emphases, original italics]

I think this approach (independent of Shagrir's notion of an analog-model computer) is the best way to deal with Searle and Putnam.

34. On the brain as computer:

Should we think of this relation betwen inputs and outputs as a kind of effective computational procedure? If we know that that the system is in state A and receives input B will we be able to tell what the next set of representations will be like? The answer at this point is simply that we do not know whether there is a procedure like this or not, nor much about how it would operate if there were. What we do know, however, is that there are consistent causal relations between input and representations, both in science and childhood, and my empirical claim is that those relations are similar. The best current explanation of how a brain could instantiate this kind of system of rules and representations is that it is a kind of computer.

An abductive argument that (i.e., a scientific hypothesis that) the brain is a computer. Note that this is a much more reasonable argument than Searle's or than trying to model the brain as, say, a Turing machine.

9.10 Further Sources of Information

1. Books and Essays:

   (a) Brunjes, Shannon (1977), "What Is a Computer?", *Journal of Medical Systems* 1(1): 79–85.

      - A computer is "an information-processing machine, varying in size from the very large system to the very small microprocessor, operating by means of stored programs which can be modified, and using various auxiliary devices to communicate with the user." (p. 85)

   (b) **Chaitin, Gregory (2006), "How Real Are Real Numbers?",** *International Journal of Bifurcation and Chaos*

      i. Accessed 6 November 2012 from:
         2006 version: http://www.cs.auckland.ac.nz/∼chaitin/olympia.pdf
         2009 version: http://www.umcs.maine.edu/∼chaitin/wlu.html

      ii. On the halting problem, the non-existence of real numbers(!), and the idea that "everything is software, God is a computer programmer,…and the world is…a giant computer"(!!).

   (c) Chalmers, David (1993), "A Computational Foundation for the Study of Cognition", *Journal of Cognitive Science* (South Korea) 12(4) (October-December 2011): 323–357.

      i. Accessed 3 November 2014 from:
         http://j-cs.org/gnuboard/bbs/board.php?bo_table=__vol012i4&wr_id=2

      ii. Originally written in 1993; slightly revised version published 2011.

      iii. See esp. the section "What about computers?" (pp. 335–336).

      iv. The 2011 version was accompanied by commentaries (including [Egan, 2012], [Rescorla, 2012b], [Scheutz, 2012], and [Shagrir, 2012]) and a reply by Chalmers [Chalmers, 2012].

   (d) **Davis, Martin (2004), "The Myth of Hypercomputation", in C. Teuscher (ed.),** *Alan Turing: The Life and Legacy of a Great Thinker* **(Berlin: Springer): 195–212.**

      - Accessed 15 May 2014 from:
         http://www1.maths.leeds.ac.uk/∼pmt6sbc/docs/davis.myth.pdf

      - Contains interesting comments on the nature of of computation and computers.

   (e) Floridi, Luciano (1999), "The Digital Workshop", Ch. 2 of *Philosophy and Computing: An Introduction* (London: Routledge).

   (f) Giere, Ronald N. (2003), "The Role of Computation in Scientific Cognition", *Journal of Experimental and Theoretical Artificial Intelligence* 15: 195–202.

   (g) Gruber, John (2007), "Apple's Computer, Incorporated", *Macworld* (April): 112.

      - "Never mind the name change—the Apple TV and iPhone are computers to the core."

   (h) Harnish, Robert M. (2002), "Coda: Computation for Cognitive Science, or What IS a Computer, Anyway?", *Minds, Brains, Computers: An Historical Introduction to the Foundations of Cognitive Science* (Malden, MA: Blackwell Publishers): 394–412.

(i) Kanat-Alexander, Max (2008), "What Is a Computer?", *Code Simplicity* (10 October)

    i. Accessed 7 November 2012 from:
      http://www.codesimplicity.com/post/what-is-a-computer/

    ii. A computer is "Any piece of matter which can carry out symbolic instructions and compare data in assistance of a human goal."

(j) Kroes, Peter (2010), "Engineering and the Dual Nature of Technical Artefacts", *Cambridge Journal of Economics* 34: 51–62.

- Argues that "technical artefacts are physical structures with functional properties"

(k) Lloyd, Seth (2000), "Ultimate Physical Limits to Computation", *Nature* 406 (31 August): 1047–1054.

    i. Pre-prints accessed 4 June 2014 from:
      http://arxiv.org/pdf/quant-ph/9908043.pdf?origin=publicationDetail
      and
      http://cds.cern.ch/record/396654/files/9908043.pdf

    ii. Investigates "quantitative bounds to the computational power of an 'ultimate laptop' with a mass of one kilogram confined to a volume of one litre."

(l) Lloyd, Seth (2002), "Computational Capacity of the Universe", *Physical Review Letters* 88(23) (June): 237901-1–237901-4.

- "All physical systems register and process information. The laws of physics determine the amount of information that a physical system can register (number of bits) and the number of elementary logic operations that a system can perform (number of ops). The Universe is a physical system. The amount of information that the Universe can register and the number of elementary operations that it can have performed over its history are calculated. The Universe can have performed 10^{120} ops on 10^{90} bits (10^{120} bits including gravitational degrees of freedom)." (from the abstract)

(m) Lloyd, Seth (2006), *Programming the Universe: A Quantum Computer Scientist Takes on the Cosmos* (New York: Alfred A. Knopf).

- See Powell 2006, Schmidhuber 2006 in "Strongly Recommended" readings, above.

(n) Marshall, Dan (2014), "The Varieties of Intrinsicality", *Philosophy and Phenomenological Research*, forthcoming; preprint accessed 31 December 2014 from:
https://www.academia.edu/7096450/The_Varieties_of_Intrinsicality

- An analysis of different theories of what 'intrinsic' means, hence useful for critiquing Searle 1990.

(o) Naur, Peter (2007), "Computing Versus Human Thinking", *Communications of the ACM* 50(1) (January): 85–94.

- Accessed 24 March 2014 from:
  http://www.computingscience.nl/docs/vakken/exp/Articles/NaurThinking.pdf
- "[T]he nervous system. . . has no similarity whatever to a computer." (p. 85)

(p) Samuel, Arthur L. (1953), "Computing Bit by Bit, or Digital Computers Made Easy", *Proceedings of the Institute of Radio Engineers* 41(10) (October): 1223–1230.

- A description of how computers work, written, for radio engineers who (in 1953) may never have seen one(!), by the creator of one of the earliest AI programs: Samuel's checkers player.
- Comes close to defining a computer "as an information or data processing device which accepts data in one form and delivers it in an altered form" (p. 1223).

(q) Schulman, Ari N. (2009), "Why Minds Are Not Like Computers", *The New Atlantis* (Winter): 46–68.

(r) Shagrir, Oron (1999), "What Is Computer Science About?", *The Monist* 82(1): 131–149.

- A short, but wide-ranging, paper on the nature of computers; also discusses hypercomputation, analog computation, and computation as not being purely syntactic

(s) Shannon, Claude E. (1953), "Computers and Automata", *Proceedings of the Institute of Radio Engineers* 41(10) (October): 1234–1241.

- Discusses the use of computers for non-numerical computing.

(t) Shapiro, Ehud; & Benenson, Yaakov (2006), "Bringing DNA Computers to Life", *Scientific American* 294(5) (May): 44–51.

- accessed 24 March 2014 from:
  http://www.wisdom.weizmann.ac.il/∼udi/papers/ShapiroBenensonMay06.pdf
- Can DNA be a computer? See also [Adleman, 1998, Qian and Winfree, 2011].

(u) On a related question: If the brain is a computer, is it a Turing machine?:

- Sackur, Jérôme; & Dehaene, Stanislas (2009), "The Cognitive Architecture for Chaining of Two Mental Operations", *Cognition* 111: 187–211.
- Zylberberg, Ariel; Dehaene, Stanislas; Roelfsema, Pieter R.; & Sigman, Mariano (2011), "The Human Turing Machine: A Neural Framework for Mental Programs", *Trends in Cognitive Sciences* 15(7) (July): 293–300.

2. Websites:

(a) Anderson, David L. (2006), "The Nature of Computers", accessed 7 November 2012 from:
http://www.mind.ilstu.edu/curriculum/modOverview.php?modGUI=196

- From "The Mind Project", http://www.mind.ilstu.edu/ (accessed 7 November 2012).

(b) Chirimuuta, Mazviita; Boone, Trey; & DeMedonsa, Michal (2014), "Is Your Brain a Computer?", *Instant HPS* (19 September), accessed 23 September 2014 from:
https://www.youtube.com/watch?v=-8q_UXpHsaY

(c) Kuczynski, John-Michael (2014), "Is the Human Brain a Computer? Alan Turing on Mind & Computers",
http://www.thecritique.com/articles/is-the-human-brain-a-computer-alan-turing-on-mind-computers/
(accessed 19 January 2015).

- Part of a series of online articles providing background for the movie *The Imitation Game*. In order to answer his title question, he reviews Turing 1936. Among the interesting, if controversial, points that he makes are that "a recursive function is one that is defined for each of its own outputs" and that the recursive definitions of addition, multiplication, and exponentiation are the reason that "arithmetic requires no thought at all" (reminiscent of [Dennett, 2009] and [Dennett, 2013]'s notion of Turing's "inversion").

(d) Schmidhuber, Jürgen, "Zuse's Thesis: The Universe is a Computer", accessed 7 November 2012 from:
http://www.idsia.ch/∼juergen/digitalphysics.html

- Zuse was one of the inventors of modern computers; see:
http://ei.cs.vt.edu/∼history/Zuse.html (accessed 7 November 2012).

Chapter 10

The Church-Turing
Computability Thesis:
I. What Is a Procedure?

"The Church-Turing Thesis[?]: A problem is computable just in case it wants to be solved."
—anonymous undergraduate student in the author's course, CSE 111, "Great Ideas in Computer Science" (2004)[1]

"I believe that history will record that around the mid twentieth century many classical problems of philosophy and psychology were transformed by a new notion of process: that of a *symbolic* or *computational* process."
—[Pylyshyn, 1992, 4]

---

[1]http://www.cse.buffalo.edu/∼rapaport/111F04.html

10.1   Recommended Readings:

1. Required:

   - Cleland, Carol E. (1993), "Is the Church-Turing Thesis True?", *Minds and Machines* 3(3) (August): 283–312.

   - Preston, Beth (2013), "The Centralized Control Model", Ch. 1 of Preston, Beth (2013), *A Philosophy of Material Culture: Action, Function, Mind* (New York: Routledge): 16–43.

     - p. 17, first full paragraph
       * Introduces three theories of "production" (producing objects by following procedures): Aristotle's, Marx's, and Dipert's.
     - §"Aristotle", from p. 17 to p. 18 (end of top paragraph)
     - §"Marx", p. 22 (from first full paragraph) to p. 23 (end of first full paragraph)
     - §"Dipert", p. 29, first paragraph
     - §**"The Centralized Control Model", pp. 30–32**
     - §**"Control & Improvisation", pp. 39–43.**
     - Skim the rest of Ch. 1.

2. Very Strongly Recommended:

   - Farkas, David K. (1999), "The Logical and Rhetorical Construction of Procedural Discourse", *Technical Communication* 46(1) (February): 42–54.

     - Accessed 4 February 2014 from:
       http://tinyurl.com/Farkas1999[2]
     - An interesting look at how to write procedures (i.e., instructions), from the point of view of a technical writer.

3. Strongly Recommended:

   - Horsten, Leon; & Roelants, Herman (1995), "The Church-Turing Thesis and Effective Mundane Procedures", *Minds and Machines* 5(1): 1–8.

     - A reply to Cleland 1993.

   - Cleland, Carol E. (1995), "Effective Procedures and Computable Functions", *Minds and Machines* 5(1): 9–23.

     - A reply to Horsten & Roelants 1995.

   - Cleland, Carol E. (2001), "Recipes, Algorithms, and Programs", *Minds and Machines* 11(2) (May): 219–237.

   - Cleland, Carol E. (2002), "On Effective Procedures", *Minds and Machines* 12(2) (May): 159–179.

4. Recommended:

   - Replies to Cleland:

     - Israel, David (2002), "Reflections on Gödel's and Gandy's Reflections on Turing's Thesis", *Minds and Machines* 12(2) (May): 181–201.

---

[2]http://www.hcde.washington.edu/sites/default/files/people/docs/proceduraldiscourse.pdf

- – Seligman, Jeremy (2002), "The Scope of Turing's Analysis of Effective Procedures", *Minds and Machines* 12(2) (May): 203–220.
- Cleland, Carol E. (guest ed.) (2002), Special Issue on Effective Procedures, *Minds and Machines* 12(2) (May).
  - – Contains Israel 2002, Seligman 2002, cited above.
  - – Full table of contents accessed 4 February 2014 from:
    http://link.springer.com/journal/11023/12/2

10.2   Introduction

Algorithms—including procedures and recipes—can fail for many reasons; here are three of them:

1. They can omit crucial steps:

Figure 10.1: ©2011, Creators Syndicate

2. They can fail to be specific enough (or make too many assumptions):

Figure 10.2: ©??, Tribune Col. Syndicate

3. They can be highly context dependent or ambiguous (see the first instruction below):

The general theme of the next few chapters is this: finding faults with each part of the informal definition of 'algorithm':

- Does it have to be a *finite* procedure?

- Does it have to be "*effective*"?

  - Does it have to halt?
  - Does it have to solve the problem? (Counterexample: heuristics)

- Does it have to be *ambiguous/precisely described*? (Counterexample: recipes)

Let us take stock of where we are. We proposed two possible answers to the question of what computer science is. The first was that it is the science (or the study, or the engineering) of *computers*, and we have just been considering what a computer is, examining the history and philosophy of computers. The second possible answer was that perhaps computer science is the science (or the study) of *computing*, that is, of algorithms, and we then investigated what algorithms are, their history, and their analysis in terms of Turing machines.

But some philosophers have questioned the Turing machine analysis of algorithms, suggesting that a more general notion, sometimes labeled 'procedure', is needed. In this chapter, we will consider some issues related to the environment in which a computation takes place. We will look at other issues—related to hypercomputation (see below)—in Ch. 11.

Recall from §7.6.4 that "Church's thesis" is, roughly, the claim that the informal notion of "algorithm" or "effective computation" is equivalent to (or is completely captured by, or can be completely analyzed as) the lambda-calculus.[3] "Turing's thesis" is, roughly, the claim that the informal notion of "algorithm" or "computability" is equivalent to/completely captured by, or completely analyzed as, the notion of a Turing machine. Turing proved that the lambda calculus was logically equivalent

---

[3]For an elemetary introduction to the lambda-calculus, see [PolR, 2009, §"Alonzo Church's Lambda-Calculus"].

to the Turing machine; that is, he proved that any function that was computable by Church's lambda calculus was also computable by a Turing machine (more precisely, that any lambda-computation could be "compiled" into a Turing machine)[4] and vice versa—that any function that was computable by a Turing machine was also computable by the lambda-calculus (so that the lambda-calculus and Turing machines were inter-compilable). Consequently, their theses are often combined under the name the "Church-Turing thesis". [Soare, 2009, §12] has argued that it should be called simply the "Computability Thesis", on the grounds that—given the equivalence of all mathematical models of computation (Church's, Turing's, Gödel's, etc.)—there are really many such theses. I will call it the Church-Turing Computability Thesis.

In this chapter, we will look at one kind of objection to the thesis, namely, that there is a more general notion—the notion of a "procedure". The objection takes the form of claiming that there are informally computable tasks that *are* procedures but that are not computable by Turing machines.

In the next chapter, we will look at a related, but slightly different kind of objection, namely, that some non-Turing-machine computable functions—such as the Halting Problem—can be computed in a formal, mathematical sense of "computation" that goes "beyond" Turing-machine computation. This is called 'hypercomputation'.

10.3   Preliminary Thought Experiment

Let's begin with a "thought experiment", that is, a mental puzzle designed to give you some intuitions about the arguments to come. Consider this quotation from Herbert Simon:

> Natural science impinges on an artifact through two of the three terms of the relation that characterizes it: the structure of the artifact itself and the environment in which it performs.[5] Whether a clock will in fact tell time depends on its internal construction and where it is placed. Whether a knife will cut depends on the material of its blade and the hardness of the substance to which it is applied. ...An artifact can be thought of as...an "interface"...between an "inner" environment, the substance and organization of the artifact itself, and an "outer" environment, the surroundings in which it operates. If the inner environment is appropriate to the outer environment, or vice versa, the artifact will serve its intended purpose. [Simon, 1996b, 6]

And, presumably, otherwise not.

But more can be said, perhaps relating to the "third term" mentioned in the footnote to this quotation: Consider a clock. Lewis Carroll (of *Alice in Wonderland* fame) once observed that a(n analog) clock whose hands don't work at all is better than one that loses 1 minute a day, because the former is correct twice a day, whereas the latter is correct only once every 2 years [Carroll, 1848].

---

[4]I am indebted to John Case's lectures (at SUNY Buffalo, ca. 1983) on theory of computation for this phrasing.

[5]The third "term of the relation" is an artifact's "purpose or goal" [Simon, 1996b, 5].

Here is the thought experiment: What about the clock that has stopped and one that is consistently 5 minutes slow? The latter is *never* correct—unless you simply move the numerals on its face 5 minutes counterclockwise; then it would always be correct![6] The (user's?) *interpretation* of the clock's output seems to be what counts.[7]

10.4 What Is a Procedure?

DISCUSS SHAPIRO'S NOTION [Shapiro, 2001] HERE

10.5 Challenges to the Church-Turing Computability Thesis

Some philosophers have challenged the Church-Turing Computability Thesis, arguing that there are things that *are* intuitively algorithms but that are *not* Turing machines.

10.5.1 Carol Cleland: Some Effective Procedures Are Not Turing Machines

In a series of papers, Carol Cleland has argued that there are effective procedures that are not Turing machines [Cleland, 1993, Cleland, 1995, Cleland, 2001, Cleland, 2002, Cleland, 2004]. By 'effective procedure', she means a "mundane" (or "quotidian", that is, ordinary or "everyday"; [Cleland, 2002, 177n.1]) procedure that generates a causal process, that is, a procedure that physically causes (or "effects") a change in the world.[8] The bigger issue here concerns the relation of a Turing machine to the *interpretation* of its output. We will return to this aspect of the issue in Ch. 17.

According to Cleland, there are three interpretations of the Church-Turing Computability Thesis:

1. It applies *only* to functions of integers (or, possibly, also to anything representable by—that is, codable into—integers).

2. It applies to *all* functions (including real-valued functions).

3. It applies to the production of mental and physical phenomena, such as is envisaged by AI or in robotics.

She agrees that it cannot be proved but that it *can* be falsified.[9] It can't be proved, because one of the two notions that the Church-Turing Computability Thesis says are

---

[6]The actual number is probably closer to $5\frac{1}{2}$ minutes, but this is only a thought experiment!

[7]Recall Searle's emphasis on the user's interpretation, §9.3.

[8]There may be an unintentional pun here. The word 'effective' as used in the phrase 'effective procedure' is a semi-technical term that is roughly synonymous with 'algorithmic'. The verb 'to effect', as used in the phrase "to effect a change (in something)", is roughly synonymous with the verb 'to cause', and it is not directly related to 'effective' in the previous sense. (For the difference between the *verb* 'to effect' and the *noun* 'an effect' (as well as the similar-sounding verb and noun 'affect'), see [Rapaport, 2013, §4.2.0, "affect vs. effect"].)

[9]Recall our discussion of falsifiability in science, §4.7.1.

equivalent is an informal notion, hence not capable of occurring in a formal proof. There are two possbilities for why it can be falsified: According to Cleland, the Church-Turing Computability Thesis can be falsified by exhibiting an intuitively effective procedure, "but not in Turing's sense", that is "more powerful" than a Turing machine [Cleland, 1993, 285]. Presumably, the qualification "but not in Turing's sense" simply means that it must be intuitively effective yet not capable of being carried out by a Turing machine, because, after all, that's what Turing thought his Turing machines could do, namely, carry out any intuitively effective procedure.

But she also suggests another sense in which the Church-Turing Computability Thesis might be falsifiable: By exhibiting a procedure that *is* intuitively effective *in Turing's sense* yet is not Turing-machine-computable. In other words, there might be two different kinds of counterexamples to the Church-Turing Computability Thesis: If Turing were alive, we could show him an intuitively effective procedure that we claim is not Turing-machine-computable, and he might agree; or we could show him one, and, either he would *not* agree that it was intuitively effective (thus denying that it was a possible counterexample), or he could show that, indeed, it was Turing-machine-computable (showing how it is not a counterexample at all). Cleland seems to be opting for the first of these.

Curiously, however, she goes on to say that her "mundane procedures" *are* going to be effective "in Turing's sense" [Cleland, 1993, 286]! In any case, they differ from Turing-machine-computable procedures by being *causal*. (When reading Cleland's article, you should continually ask yourself two questions: *Are* her "mundane procedures" causal? Are Turing machines *not* causal?) Here is her argument, with comments after some of the premises:

1. A "procedure" is a specification of something *to be followed* [Cleland, 1993, 287]. This includes recipes as well as computer programs. Note that recipes can be notoriously vague (exactly how much is a "pinch" of salt?) [Sheraton, 1981], whereas computer programs must be excruciatingly precise.

   - Her characterization of a procedure as something to be followed puts a focus on imperatives: You can follow an instruction that says, "Do this!" But there are other ways to characterize procedures. Stuart C. Shapiro [Shapiro, 2001], for example, describes a procedure as a way to do something. Here, the focus is on the goal or end product; the way to do it—the way to accomplish that goal—might be to evaluate a function or to determine the truth value of a proposition, not necessarily to "follow" an imperative command.

   - You should also recall (from §8.10.1) that Turing machines don't normally "follow" any instructions! The Turing machine table is a description of the Turing machine, but it is not something that the Turing machine consults and then executes. That only happens in a *universal* Turing machine, but, in that case, there are two different programs to consider: There is the program encoded on the universal Turing machine's tape; that program *is* consulted and followed. But there is the fetch-execute procedure that constitutes the universal Turing machine's machine table; that one is *not*

consulted or followed.

2. To say that a "mundane" procedure is "effective" means, by definition, that following it always results in a certain kind of outcome [Cleland, 1993, 291].

   - The semi-technical notion of "effective" as it is used in the theory of computation is, as we have seen (§7.5), somewhat ambiguous. As Cleland notes [Cleland, 1993, 291], Minsky calls an algorithm 'effective' if it is "precisely described" [Minsky, 1967]. And [Church, 1936b, 50ff; cf. 357] calls an algorithm 'effective' if there is a formal system that takes a formal analogue of the algorithm's input, precisely manipulates it, and yields a formal analogue of its output. Church's notion seems to combine aspects of both Minsky's and Cleland's notions.

   - A non-terminating program (either one that erroneously goes into an infinite loop or else one that computes all the digits in the decimal expansion of a real number) can be "effective" *at each step* even though it never halts. We'll return to this notion in Ch. 11, when we look at the notion of interactive computing.

3. The steps of a recipe *can* be precisely described (hence, they can be effective in Minsky's sense).

   - This is certainly a controversial claim; we will explore it in more detail in §10.5.2.

4. A procedure is *effective for* producing a specific output. For example, a procedure for producing fire or hollandaise sauce might not be effective for producing chocolate.

   - In other words, being effective is not a property of a procedure but a *relation* between a procedure and a kind of output. This might seem to be reasonable, but a procedure for producing the truth table for conjunction might also be effective for producing the truth table for disjunction by suitably reinterpreting the symbols.[10]

---

[10]Here is a truth-table for conjunction, using '0' to represent "false" and '1' to represent "true":

0	0	0
0	1	0
1	0	0
1	1	1

Note that, in the 3rd column, which represents the conjunction of the first two columns, there are three '0's and one '1', which occurs only in the line where both inputs are '1'.

And here is the analogous truth-table for disjunction:

0	0	0
0	1	1
1	0	1
1	1	1

Note that the third column has three '1's and only one '0', which occurs only in the line where both inputs are '0'.

Now suppose, instead, that we represent "true" by '0' and "false" by '1'. Then the *first* table represents

- Here is another example: A procedure that is effective for simulating a battle in the Civil War might also be effective for simulating a particular game of chess [Fodor, 1978, 232]. Or a procedure that is effective for computing with a mathematical lattice might also be effective for computing with a chemical lattice.[11]

- In cases such as these, the notion of effectiveness might not be Church's, because of the possibility of interpreting the output differently. How important to the notions of (intuitively) effective computation and formal computation is the interpretation of the output symbols?

5. The effectiveness of a recipe for producing hollandaise sauce depends on causal processes in the actual world, and these causal processes are independent of the recipe (the mundane procedure) [Cleland, 1993, 294].

- Compare: The "effectiveness" of a clock's telling the correct time depends just as much on the orientation of the clock face as it does on the position of the hands and the internal clockwork mechanism. That is, it depends, in part, on a situation in the external world. And the orientation of the clock face is independent of the clock's internal mechanism (or "procedure").

6. Therefore, mundane processes can (must?) be effective for a given output *P* in the *actual* world yet not be effective for *P* in some other *possible* world.

- This is also plausible. Consider a blocks-world computer program that instructs a robot how to pick up blocks and move them onto or off of other blocks. I once saw a live demo of such a program. Unfortunately, the robot failed to pick up one of the blocks, because it was not correctly placed, yet the program continued to execute perfectly even though the output was not what was intended.

7. Turing machines are equally effective in all possible worlds, because they are causally inert.

- But here we have a potential equivocation on 'effective'. Turing machines are effective in the sense of being step-by-step algorithms that are precisely specified, but they are not necessarily effective *for an intended output P*: It depends on the interpretation of *P* in the possible world!

8. Therefore, there are mundane procedures (such as recipes for hollandaise sauce) that can produce hollandaise sauce because they result in appropriate causal processes, *but* there are no Turing machines that can produce hollandaise sauce, because Turing machines are purely formal and therefore causally inert. QED

---

disjunction, and the *second* one represents *conjunction*! (A similar point is made in [Peacocke, 1995, §1, p. 231].)

[11]"Nicolaas de Bruijn once told me roughly the following anecdote: Some chemists were talking about a certain molecular structure, expressing difficulty in understanding it. De Bruijn, overhearing them, thought they were talking about mathematical lattice theory, since everything they said could be—and was—interpreted by him as being about the mathematical, rather than the chemical, domain. He told them the solution of their problem in terms of lattice theory. They, of course, understood it in terms of chemistry. Were de Bruijn and the chemists talking about the same thing?" [Rapaport, 1995, §2.5.1, p. 63]

To the objection that physical implementations of Turing machines could be causally "ert" (so to speak),[12] Cleland replies as follows: The embodied Turing machine's "actions" are not physical actions but action-*kinds*; therefore, they *are* causally *in*ert.

But an embodied Turing machine *does* act: *Embodied* action-*kinds are* causal actions. Alternatively, a Turing machine's '0'-'1' outputs can be interpreted by a device that does have causal effects, such as a light switch or a thermostat.

Perhaps a procedure or algorithm that is "effective for *P*" is better understood as an algorithm *simpliciter*. In the actual world, it does *P*. In some other possible world, perhaps it does Q ($\neq$ *P*). In yet another possible world, perhaps it does nothing (or loops forever). And so on. Hence, mundane procedures are *interpreted* Turing machine programs, so they *are* computable.

Let's consider Cleland's example of a recipe for hollandaise sauce in more detail: Let's suppose that we have an algorithm (a recipe) that tells us to mix eggs and oil, and that outputs hollandaise sauce. Suppose that, in the actual world, the result of mixing the egg and oil is an emulsion that is, in fact, hollandaise sauce. And let us suppose that, in possible world w_1 whose chemistry differs from that in the actual world, mixing eggs and oil does not result in an emulsion, so that no hollandaise sauce is output (instead, the output is a messy mixture of eggs and oil). Compare those situations with the following: Suppose that we have an algorithm (a recipe) that tells us to mix eggs and oil *until an emulsion is produced*, and that outputs hollandaise sauce. In the actual world, an emulsion is indeed produced, and hollandaise sauce is output. But in w_1, this algorithm goes into an infinite loop; nothing (and, in particular, no hollandaise sauce) is ever output.

One problem with this is that the "until" clause ("until an emulsion is produced") is not clearly algorithmic. How would the *computer* tell if an emulsion has been produced? This is not a clearly algorithmic Boolean condition whose truth value can be determined by the computer simply by checking one of its switch settings (that is, a value stored in some variable). It would need sensors to determine what the external world is like. But that is a form of *interactive* computing, which we'll discuss in Ch. 11.

10.5.2 Beth Preston: Are Recipes Algorithms?

Introductory computer science courses often use the analogy of recipes to explain what algorithms are. Recipes are clearly procedures of some kind [Sheraton, 1981]. But are recipes really good models of algorithms?

Cleland has assumed that they are. Beth Preston [Preston, 2006] has a different take. She is interested in the nature of artifacts and how they are produced (or implemented) from plans (such as blueprints). Compare this to how a program (which is like a plan) is actually executed.

According to Preston, the classical view is that of "centralized control". The etymology of the word 'control' is of interest here: A "counter-roll" is a "duplicate register"; to control something originally meant to faithfully copy it for the sake of

---

[12]'Inert' comes from the Latin prefix 'in-', meaning "not", and the Latin 'artem', meaning "skill"; so, if 'inert' means "lacking causal power", then perhaps the non-word 'ert' could mean "having causal power"; see the *OED*'s entry on 'inert' [http://www.oed.com/view/Entry/94999].
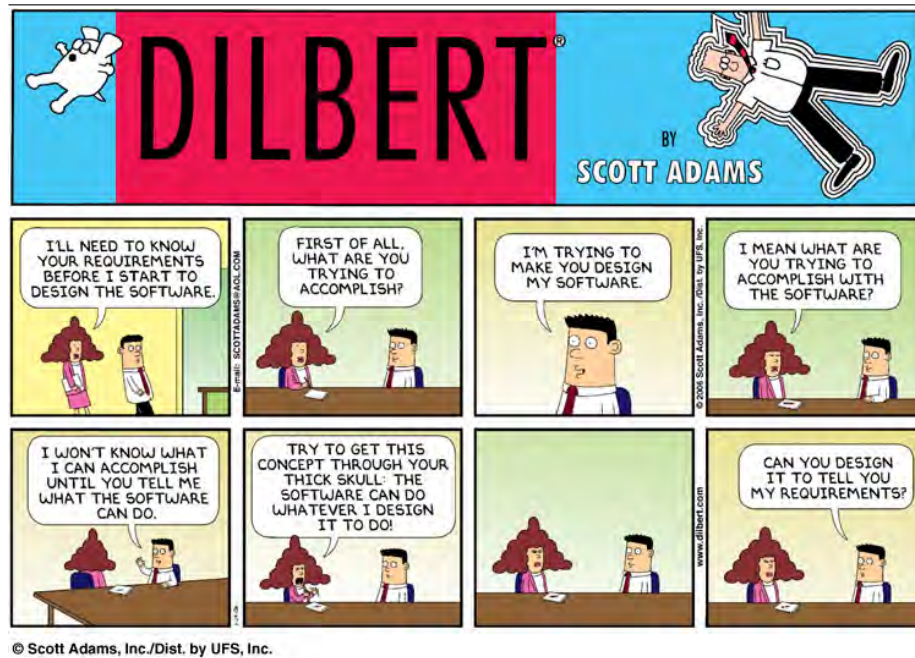
Figure 10.3: ©2006, Universal Uclick

verification or to regulate it.[13] So, to implement a plan is to copy an abstract design into reality, i.e., to control it.

There is a "mental design" of the artifact in someone's mind. This mental design "specifies all the relevant features of the" artifact (the "copy") (p. 30), "along with a set of instructions for construction" (p. 39). Then the "actual construction" of the artifact (that is, the copying of the mental design) "is a process that faithfully follows the instructions of the construction plan, and by so doing reproduces in a material medium the features of the product specified in the design. This faithful copying relationship between the design and construction phases of production is the *control* aspect of the model" (pp. 30–31). Compare the way in which a program controls the operations and output of a computer.

But there is a problem: a "faithful copy" "requires that *all* relevant features" of the artifact be specified in the design. "In other words, the design is ideally supposed to be an algorithm (effective procedure) for realizing both the construction process and the product" (p. 39). Recipes, however, show that this ideal model doesn't describe reality, according to Preston.

So, recipes differ from algorithms in several ways:

---

[13]See the *OED* definitions of 'control' as a noun (http://www.oed.com/view/Entry/40562) and as a verb (http://www.oed.com/view/Entry/40563).

1. Recipes leave details open (e.g., of ingredients, which play the same role in recipes that data structures do in programs):

    (a) Recipes provide for *alternatives*: "use either yogurt or sour cream".

    - But couldn't a recipe, or a program for that matter, simply call for a data-analogue of a typed variable or a subroutine here? The recipe might call, not for yogurt or else sour cream, but, instead, for a "white, sour, milk-based substance".

    - And what about non-deterministic algorithms? Here is an example of a non-deterministic procedure for computing the absolute value of x using a "guarded if" control structure:[14]

        if $x \geq 0$ **then** return x;
        if $x \leq 0$ **then** return $-x$;

        Here, if $x = 0$, it does not matter which line of the program is executed. In such procedures, a detail is left open, yet we still have an *algorithm*.

    - Or consider a program that simply says to input an integer, without specifying anything else about the integer. (As a magician might say, "Think of a number, any number...). This could still be an algorithm.

    (b) Recipes specify some ingredients *generically*: "use frosting", without specifying what kind of frosting.

    - Here, again, compare typed variables or named subroutines. It does not matter *how* the subroutine works; all that matters is its input-output behavior (*what* it does). And it doesn't matter *what* value the variable takes, as long as it is of the correct type. Typed variables and named subroutines are "generic", yet they appear in algorithms.

    (c) Recipes provide for *optional* ingredients ("use chopped nuts if desired").

    - But compare any conditional statement in a program that is based on user input. (On the other hand, user input *may* raise issues for interactive computing; see Ch. 11.)

2. Recipes leave construction steps (= control structure?) open: For instance, the order of steps is not necessarily specified (at best, a *partial order* is given): "cream the butter and sugar, then add the rest of the ingredients" (where no order for adding is given for "the rest of the ingredients").

    - Again, compare non-deterministic statements, such as the guarded-if command in the above example, or programs written in languages like Lisp, where the order of the functions in the program is not related in any way to the order in which they are evaluated when the program is executed. (A Lisp program is an (unordered) *set* of functions, not a(n ordered) *sequence* of instructions.)

---

[14]In an ordinary "if" statement, when more than one Boolean condition is satsified, the first one is executed. In a "guarded if" statement, it doesn't matter which one is executed. See [Gries, 1981, Ch. 10] or http://en.wikipedia.org/wiki/Guarded_Command_Language#Selection:_if

3. In recipes, some necessary steps can be *omitted* (that is, go unmentioned): "put cookies on baking sheet". But should the baking sheet be greased? A knowledgeable chef would know that it has to be, so a recipe written for such a chef need not mention the obvious.

   • But the same kind of thing can occur in a program, with preconditions that are assumed but not spelled out, or details hidden in a subroutine. (Perhaps "put cookies on baking sheet" is a call to a subroutine of the form: "grease baking sheet; put cookies on it".)

4. Recipes can provide *alternative ways* to do something: "roll in powdered sugar or shake in bag with powdered sugar"

   • Again, non-determinism is similar, as are subroutines: To say "multiply *x* and *y*" is not to specify how; to say "coat in powdered sugar" is not to specify whether this should be done by rolling or shaking.

Preston claims that the cook (i.e., the CPU) is expected to do some of the design work, to supply the missing parts. So, not everything is in the design. She claims that cooks don't *faithfully follow* recipes; instead, they *improvise*. They can even change other ("fixed") parts of the recipe, because of their background knowledge, based on experience. For example, they can substitute walnuts for pecans in a recipe for pecan pie. Therefore, the constructor/executor is typically *intelligent*, in contrast to an *un*intelligent CPU. Here, however, compare an intelligent student hand-tracing a program. DISCUSS "DARWIN'S DANGEROUS INVERSION"? [Dennett, 2009]. Preston notes that the control ideal assumes that the world is stable over time and homogeneous over agents.

But here is a different interpretation of her analysis: Her centralized control model might describe an algorithm together with a CPU producing a process (i.e., an algorithm being executed). But her theory of collaborative improvisation might describe a specification together with a programmer producing an algorithm. That is, the development of an algorithm by a programmer from a specification is improvisatory and collaborative, but the output of an algorithm by a computer from executing the algorithm is centralized control.

So, recipes are more like design specifications *for* computer programs than they are like *algorithms*. In fact, my counterexamples to differentiate between algorithms and recipes just show that *either* recipes are high-level programs requiring implementation in a lower level, *or* that recipes are specifications.

10.5.3   Discussion

The Church-Turing Computability Thesis, in any of its various forms, states that the *informal* notion of effective computation (or algorithm, or whatever) is equivalent to the *formal* notion of a Turing-machine program (or a λ-definable function, or a recursive function, or whatever). The arguments in favor of the Church-Turing Computability Thesis are generally of the form that (1) all known informal algorithms are Turing-

machine-computable[15] and (2) all of the formal, mathematical versions of computation are logically equivalent to each other (see §7.6.4, above).

It has also been argued that the Thesis cannot be formally proved because one "side" of it is informal, hence not capable of being part of a formal proof. Some have suggested (Gandy CITE, Sieg CITE, and [Dershowitz and Gurevich, 2008]) that the thesis *is* capable of being proved by providing a set of formal axioms for the informal notion, and then proving that Turing machines satisfy those axioms. Although this is an interesting exercise, it is not obvious that this proves the Church-Turing Computability Thesis. Rather, it seems to replace that Thesis with a new one, namely, that the informal notion is indeed captured by the formal axioms. But *that* thesis likewise cannot be proved for the same reason that the Church-Turing Computability Thesis cannot: To prove it would require using an informal notion that cannot be part of a formal proof.

Others have argued that neither (1) nor (2) are even non-deductively good arguments for the Church-Turing Computability Thesis. Against (1), it can be argued that, just because all known informal algorithms are Turing-machine-computable, it does not follow that all informal algorithms are. After all, just because Aristotle's theory of physics lasted for some 2000 years until Newton came along, it did not follow that Aristotle's physics was correct, and just because Newton's theory lasted for some 200 years until Einstein came along, it did not follow that Newton's theory was correct. So, there is no inductive reason to think that the Church-Turing Computability Thesis is correct any more than there is to think that Einstein's theory is.[16]

Against (2), it can be argued that, even though the common theme underlying the equivalence of Turing machines, λ-definability, recursive functions, etc., is "robust" and of great mathematical interest, that is not reason enough to think that there might not be any *other* theory of effective computation. (And, indeed, there are alternatives; see Ch. 11.)

So, are there good reasons for seriously doubting the Church-Turing Computability Thesis? We have just seen two candidates: Cleland argues that certain "mundane procedures" are effectively computable but not Turing-machine-computable, and Preston suggests that certain recipe-like procedures of the sort typically cited as examples of effective procedures are not really algorithmic.

But against Cleland's example, we have seen that there may be a problem in how one determines what the proper output of an algorithm is, or, to put it another way, what the problem that an algorithm is supposed to solve is. Consider a recipe for hollandaise sauce that, when correctly executed on Earth, produces hollandaise sauce but, when correctly executed on the moon, does not. Is that recipe therefore not Turing-machine-computable? It would seem to be Turing-machine-computable on earth, but not on the moon. Or is it Turing-machine-computable *simpliciter* (for example, no matter where it is executed), but conditions having nothing to do with the algorithm or recipe *itself* conspire to make it unsuccessful *as a recipe for hollandaise sauce* on the moon? Is the algorithm or recipe itself any different?

---

[15]This puts it positively. To put it negatively, no one has yet found a universally convincing example of an informally computable function that is not also Turing-machine computable

[16]On the question of whether *any* scientific theory is correct, on the grounds that they are all only falsifiable, see CITE.

And against Preston's example, we have seen that recipes are, perhaps, more like specifications for algorithms than they are like algorithms.

In the next chapter, we will look at other potential counterexamples to the Church-Turing Computability Thesis.

10.6 NOTES FOR NEXT DRAFT

1. Shapiro's more general notion of "procedure" (Farkas 1999 may be relevant here, and some of this may more properly belong in Ch. 3):

   [Shapiro, 2001] characterizes " 'procedure' as the most general term for the way 'of going about the accomplishment of something' ", citing the Merriam-Webster *Third New International Dictionary* (see http://www.merriam-webster.com/dictionary/procedure). This includes serial algorithms, parallel algorithms (which are not "step by step", or serial), operating systems (which don't halt), heuristics (which "are not guaranteed to produce the correct answer"), musical scores, and recipes. Thus, Turing machines (or Turing-machine programs), that is, algorithms as analyzed in §7.5, are a special case of procedures. Given this definition, he claims that procedures are "natural phenomena that may be, and are, objectively measured, principally in terms of the amount of time they take...and in terms of the amount of resources they require." He gives no argument for the claim that they are natural phenomena, but this seems reasonable. First, they don't seem to be "social" phenomena in the sense in which institutions such as money or governments are (cite Searle). Some of them might be, but the concept of a procedure in general surely seems to be logically prior to any specific procedure (such as the procedure for doing long division or for baking a cake). Second, there are surely some procedures "in nature", such as a bird's procedure for building a nest. Insofar as a natural science is one that deals with "objectively measurable phenomena" (again citing Webster; http://www.merriam-webster.com/dictionary/natural%20science), it follows trivially that computer science is the natural science of procedures.

   In this chapter, we are focusing on this more general notion of 'procedure'

2. On the relation between an algorithm and its output:

   Fregean senses don't *determine* reference (because of the possibility of multiple realization, or "slippage" between sense and reference [Putnam, 1975]). But senses do tell you how to *find* a referent: They are *algorithms*. The referent is like the purpose or task of the algorithm. But there is always the possibility of "slippage": An algorithm can be intended for one purpose (say, playing chess), but be used for another (say, simulating a Civil War battle [Fodor, 1978, 232]) by changing the interpretation of the inputs and outputs. (But discuss Rescorla!; see also [Pylyshyn, 1992].)

   Here is another relevant quotation:

   > [John von Neumann's]...basic insight was that the geometry of the vectors in a Hilbert space has the same formal properties as the structure of the states of a quantum-mechanical system. Once that is accepted, the difference between a quantum physicist and a mathematical operator-theorist becomes one of language and emphasis only. [Halmos, 1973, 384]

   First, this is a nice statement of mathematical structuralism. Second, this is also a nice statement of the kind of phenomenon illustrated by Fodor's Civil

War-chess example. A computer program designed to operate on Hilbert-space vectors could be used or adapted to operate on quantum-mechanical states with no change, except possible for input-output coding. It also illustrates where the intensionality of computation might be relevant: A Civil War or Hilbert-space program might be intensionally distinct from a chess or quantum-mechanics program, yet they could be extensionally equivalent.

3. Re: Cleland:
   [Sloman, 2002, §3.2] makes a possibly useful distinction between "internal" and "external" processes: The former "include manipulation of cogs, levers, pulleys, strings, etc." The latter "include movements or rearrangements of various kinds of physical objects". So, a computer on earth that is programmed to make hollandaise sauce and one on the moon that is identically programmed will have the same internal processes, but different external ones (because of differences in the external environment).

   A related distinction is that between "competence" and "performance": A computer might be competent to make hollandaise because of its internal processes, yet fail to do so because of performance limitations due to external-environmental limitations.

4. Does the ability of a machine to do something that is non-TM-computable mean that it can compute something that is non-TM-computable? What does physical performance have to do with computation? Surely we want to say that whatever a robot can do is computable, even if that includes cooking. But surely that's because of the "internal" processes, not the "external" ones.

5.   . . . Turing machines are not so relevant [to AI] intrinsically as machines that are designed from the start to have interfaces to external sensors and motors with which they can interact online, unlike Turing machines which at least in their main form are totally self contained, and are designed primarily to run in ballistic mode once set up with an intial machine table and tape configuration. [Sloman, 2002, §4.2]

   This seems to be a distinction between abstract TMs and robots. And Cleland's arguments seem more relevant to robots than to TMs, hence have nothing really to say about the Church-Turing Computability Thesis (which only concerns TMs).

6. There may be a connection between Cleland's use of hollandaise recipes to counter the Church-Turing Computability Thesis, Preston's use of recipes as (bad?) examples of algorithms, and Wegner's emphasis on interactive computing as a counterexample to CTCT:

   One example of a problem that is not algorithmic is the following instruction from a recipe [quote Knuth, 1968]: 'toss lightly until the mixture is crumbly.' This problem is not algorithmic because it is impossible for a computer to know how long to mix: this may depend on conditions such as humidity that cannot be predicted with

certainty ahead of time. In the function-based mathematical world-view, all inputs must be specified at the start of the computation, preventing the kind of feedback that would be necessary to determine when it's time to stop mixing. [Goldin and Wegner, 2005], as cited in [Cooper, 2012, 78].

7. On "Following" Rules THIS PARA NEEDS TO BE CLARIFIED:
What does it mean to "follow" a rule? Consider three examples of following rules: intelligently hand-tracing a program, crossing at a green light, and obeying the law of gravity (as a cartoon once said, it's not just a good idea; it's the law!).

In the first example, you are literally *following* the program's rules—the program tells you to do *A*, so you do *A* (and if it tells you *how to* do *A*, then you do *A in that way*. Compare how a universal Turing machine executes a program stored on its tape.

In the second example, you know that there is a rule (a legal law) that says that you are only supposed to cross the street when you have a green light. You could choose to jaywalk, but you decide to obey the law (to follow the rule). But this is different from "obeying" the (descriptive) law of gravity; with that law, you have no choice. You simply behave in accordance with the "law" of gravity.

Compare describing these as "following a rule" with describing them as "behaving in accordance with a rule"—a rule that *describes independent* behavior—and with "executing" a rule in the way that a Turing machine does. On the latter, compare executing a rule the way that a Turing machine does with the following rule (which can, but need not, be described in advance as having the goal of adding two numbers in a spreadsheet): "enter a number in cell 1; enter a number in cell 2; enter the string '=sum(cell1,cell2)' in cell 3" (or, even better?: "enter a number in cell1; enter a number in cell2; enter the string '=⟨click on cell1⟩;⟨click on cell2⟩' in cell3"), and compare these with intelligently adding by following the normal addition algorithm. The former is Turing's "strange inversion" [Dennett, 2009].[17]

8. The following quote can go either here or in an earlier chapter on the CTCT. It is Turing's statement of one of the principle justifications for the thesis.

> Several definitions have been given to express an exact meaning corresponding to the intuitive idea of 'effective calculability' as applied for instance to functions of positive integers. The purpose of the present paper is to show that the computable functions introduced by the author are identical with the λ-definable functions of Church and the general recursive functions due to Herbrand and Gödel and

---

[17]This is based on a real-life example. One of the TAs for a "Great Ideas in Computer Science" course that I taught (aimed at students who were computer-phobic) was trying to show me how a spreadsheet worked (at the time, I had never used one, so, with respect to this, I was as naive as any of the students in the course!). She gave me those instructions, but I had no idea why I was supposed to do those things. Yet I could faithfully follow her instructions without understanding them (that is, without understanding their purpose); cf. [Dennett, 2009].

> developed by Kleene. It is shown that every λ-definable function is
> computable and that every computable function is general recursive.
> ...If these results are taken in conjunction with an already available
> proof that every general recursive function is λ-definable we shall
> have the required equivalence of computability with λ-definability....
>
> ...
>
>   The identification of 'effectively calculable' functions with
> computable functions is possibly more convincing than an
> identification with the λ-definable or general recursive functions. For
> those who take this view the formal proof of equivalence provides a
> justification for Church's calculus, and allows the 'machines' which
> generate computable functions to be replaced by the more convenient
> λ-definitions. [Turing, 1937, 153]

Turing cites [Church, 1936b] for the definition of λ-definability, [Kleene, 1936a]
for the definition of general recursiveness, and [Kleene, 1936b] for the proof
of their equivalence. Gödel was among those who found Turing's computable
functions to be "more convincing than" the other formalisms.

9. CTCT can be viewed as an "explication" of the informal notion of "effective
   calculability" or "algorithm" [Montague, 1960, 430]. An "explication" is the
   replacement of an informal or vague term with a formal and more precise
   one.[18] [Mendelson, 1990, 229] calls it a "rational reconstruction" (a term also
   due to Carnap): "a precise, scientific concept that is offered as an equivalent
   of a prescientific, intuitive, imprecise notion." Mendelson goes on to argue
   that CTCT has the same status as the definition of a function as a certain set
   of ordered pairs (see §7.4.1, above) and other mathematical notions (logical
   validity, Tarski's definition of truth, the δ-ε definition of limits, etc.). Mendelson
   then claims that "it is completely unwarranted to say that CT is unprovable
   just because it states an equivalence between a vague, imprecise notion...and
   a precise mathematical notion" [Mendelson, 1990, 232]. One reason he gives
   is that both sides of the equivalence are equally vague! He points out that "the
   concept of set is no clearer than that of function". Another is that the argument
   that all Turing-machine programs are (informally) computable is considered to
   be a proof, yet it involves a vague, informal concept. (Note that it is the claim
   that all informally computable functions are Turing-machine computable that is
   usually considered incapable of proof on these grounds.)

   By the way, Mendelson's paper has one of the clearest discussions of the nature
   of CTCT.

   For responses to Mendelson, see Bringsjord 1993, [Shapiro, 1993].

10. Not sure where this belongs (this chapter? chapter on hypercomputation?):
    [Montague, 1960, 433] suggests that—for a more general notion of computation
    than a mere Turing machine (one that would apply to both digital and analog
    computation)—a computer needs an output signal that indicates when the

---

[18]The concept is due to the philosopher Rudolf Carnap; see, for example, [Carnap, 1956, 7–8].

computation is finished. In Turing's theory of computation, the machine simply halts. But, of course, how do you know that a machine has halted rather than merely being in an infinite loop?

11. On the notion of a "procedure":
[Simon, 1962, 479] offers two kinds of descriptions of phenomena in the world: state descriptions and process descriptions. "The former characterize the world as sensed; they provide the criteria for identifying objects. . . . The latter characterize the world as acted upon; they provide the means for producing or generating objects having the desired characteristics" given by a state description. His example of a state description is "A circle is the locus of all points equidistant from a given point"; his example of a process description is "To construct a circle, rotate a compass with one arm fixed until the other arm has returned to its staring point". Process descriptions describe procedures. (State descriptions seem to be part of "science", whereas process descriptions seem to be part of "engineering".)

12. Another argument against CTCT (which I won't elaborate on):
[Bowie, 1973] argues that Church's thesis is false on the grounds that the informal notion of computability is intensional but the notion of recursive functions is extensional (see p. 67). (On the difference between "intensional" and "extensional", see [Rapaport, 2009].) He also argues against it on what would now be considered "hypercomputational" grounds, roughly, that, if you give a computable function a non-computable input (intensionally represented), it will compute a non-computable output (see p. 75).

[Ross, 1974] is a reply to [Bowie, 1973]: He points out that Bowie's definition of 'computable' is non-standard, and argues that the standard definition (essentially the one given in Ch. 7) is *not* intensional.

13. Gandy's statement of Church's thesis:
"What is effectively calculable ["by an abstract human being using some mechanical aids (such as paper and pencil)"] is computable", where " 'computable'. . . mean[s] 'computable by a Turing machine' ", " 'abstract' indicates that the argument makes no appeal to the existence of practical limits on time and space [and]. . . 'effective' in the thesis serves to emphasize that the process of calculation is deterministic—not dependent on guesswork—and that it must terminate in a finite time." [Gandy, 1980, 123–124]

Gandy goes on to be concerned with a mechanical version of CTCT, whether "What can be calculated by a machine is computable" [Gandy, 1980, 124], where by 'machine' he says that he us "using the term with its nineteenth century meaning; the reader may like to imagine some glorious contraption of gleaming brass and polished mahogany, or he [sic] may choose to inspect the parts of Babbage's 'Analytical Engine' which are preserved in the Science Museum at South Kensington." [Gandy, 1980, 125]. One difference between a human (even an "abstract" one) and a machine is that the latter can easily perform paralle operations such as printing "an arbitrary number of symbols simultaneously"

[Gandy, 1980, 125]. We will not explore this issue here, but some references are listed in the "Further Sources" (also see [Deutsch, 1985], who discusses CTCT in the context of quantum computation).

14. Gödel's objection to the CTCT:

> Gödel has objected, against Turing's arguments, that the human mind may, by its grasp of abstract objects [by "insight and imaginative grasp"], be able to transcend mechanism. [Gandy, 1980, 124]

As Gandy notes, we need to know much more about the nature and limits of the human mind than we do in order to adequately evaluate this objection. After all, what *seems* to be "non-mechanical intelligence" need not be. A more detailed reply to Gödel is given in [Kleene, 1987, 492–494], the bottom line of which is this:

> The notion of an "effective calculation procedure" or "algorithm" (for which I believe Church's thesis) involves its being possible to convey a complete description of the effective procedure or algorithm by a finite communication, in advance of performing computations in accordance with it. (p. 493.)

And, according to Kleene, Gödel's objection fails to satisfy that criterion (as does the objection raised in [Kalmár, 1959]).

15. On the importance of a physical CTCT:
[Newell, 1980, 148] points out that physical systems are limited in speed, space, and reliability. Hence, a physical device that implements a Turing machine will inevitably be more limited than that abstract, ideal "machine". So, some functions that are theoretically computable on a Turing machine might *not* be physically computable on a real computer. Hence, a physical version of the CTCT is going to differ in plausibility from the abstract CTCT. The abstract version can be made slightly more realistic, perhaps, by redefining, or placing limits on, some of the terms:

> [R]ather than talk about memory being actually unbounded, we will talk about it being *open*, which is to say available up to some point, which then bounds the performance, both qualitatively and quantitatively. *Limited*, in opposition to open, will imply that the limit is not only finite, but small enough to force concern. Correspondingly, *universal* can be taken to require only sufficiently open memory, not unbounded memory. [Newell, 1980, 161]

16. According to many, a computer simulation of a hurricane is not a hurricane, because it does not get people wet CITE. But it could get simulated people simulatedly wet. The difference is between a real hurricane and a simulated one has to do, in part, with the nature of the inputs and outputs. But, as [Carleton, 1984, 222–223] notes, "The input to a fire simulation is not oxygen, tinder, and catalyst. That is, it is not the same input as goes into a natural

system which produces fire. ... [I]t is by virtue of dealing in the right kinds of input and output that one system can play the role in a situation that is played by a system we acknowledge to literally undergo the [activity]... our system simulates." Cleland's hollandaise-sauce-making program may differ in output when executed on the moon than on earth; it has the wrong output. But a hurricane-simulator, a fire-simulator, and a hollandaise-sauce program each exhibit their relevant behaviors if you ignore the input and the output. So, the question becomes how central to what it is that a computer program is (supposed to be) doing is the nature of the inputs and outputs (in general, of the environment in which it is being executed)? This is relevant to the issues concerning embodied cognition as well as to Rescorla's emphasis on semantics in computation.

How important are the input and output? Here is a definition from [Deutsch, 1985, 2]:

> Two classical deterministic computing machines are '*computationally equivalent*' under given labellings of their input and output states if they compute the same function under those labellings.

But wouldn't they be computationally equivalent (in a less-restricted sense) if those labellings were ignored? On this definition, Fodor's Civil War and chess computers are not computationally equivalent. Rescorla, on the other hand, might favor this definition.

17. [Goodman, 1987, 482] raises the question of whether the CTCT is a reduction "of the informal mathematical notion of algorithm... [to] the formal set-theoretic notion of a Turing machine program." The former is intensional in nature, because "An algorithm, in the informal mathematical sense, is a specific procedure *for solving a particular kind of mathematical problem*" (my italics; see also p. 483). But the latter is extensional, because "the Turing machine program does not tell you what the program is for. ... [Only the] documentation contains the intensional content which is missing from the bare machine code... and brings the program closer to the algorithm which it is intended to implement" (p. 483). This seems to nicely capture Fodor's Civil War-chess puzzle: That a particular program is a computational model of a Civil War battle *rather than* of a chess match is a matter of its intensional description; extensionally, they are the same program (modulo input-output encoding). Ah, but perhaps *that* is where the intensionality can be put back in the Turing machine: in the input-output encoding. But that is not really part of the machine; it is part of a user's interaction with the machine or how a user treats or interprets the machine's output, how a user interprets what the machine is doing. Here, 'what the machine is doing' is potentially ambiguous between, say, "modeling a Civil War battle" (a description from Dennett's intentional stance) and some design-stance description that would be neutral between "modeling a Civil War battle" and "modeling a chess match" (see [Goodman, 1987, 483] for a very similar discussion).

However, Goodman does not believe that the CTCT is

an analysis of the informal concept of algorithm. It at most provides a necessary condition for the existence of an algorithm. That is, a problem which no Turing machine can solve cannot be solved algorithmically. However, a Turing machine program without additional explanation is not an algorithm, and an algorithm is not as it stands a Turing machine program. ...My contention is rather that not all of the content of our informal intensional talk about algorithms is captured by extensional talk about Turing machine programs.

From this point of view, then, there is nothing reductionistic about Church's thesis, since it does not assert that any previously known objects can be eliminated in favor of any new objects.

18.   "Church's superthesis", an assertion that for any algorithm whatsoever, there is a Turing machine that computes the same function *the same way*" [Shapiro, 1993, 65], citing Kreisel, Georg (1969), "Some Reasons for Generalizing Recursion Theory", *Logic Colloquium '69* (Amsterdam: North-Holland): 138–198.

19. On Rescorla's emphasis on semantics in computation, and on Cleland's emphasis on the stated goal of a computation, compare this passage from Piccinini:

[Stored-program computers have the ability to respond to (non-semantically individuated) strings of tokens stored in their memory by executing sequences of primitive operations, which in turn generate new strings of tokens that get stored in memory. [Note that this is basically a description of how computers work, or of what computation is—note added by WJR.] Different bits and pieces [that is, substrings—WJR] of these strings of tokens have different effects on the machine. ...An accurate description of how tokens can be compounded into sub-strings, and sub-strings can be compounded into strings, which does not presuppose that the strings of tokens have any content, may be called the syntax of the system of strings manipulated by the computer. [Compare my description of syntax in §§14.4 and 19.5.3.2—WJR.] ...[T]he effect of a string on the computer is assigned to it [that is, the string—WJR] as its content. This assignment constitutes an *internal* semantics of a computer. An internal semantics assigns as contents to a system its own internal components and activities, whereas an ordinary (external) semantics assigns as contents to a system objects and properies in the system's environment. ...None of this entails that computer languages have any external semantics, i.e. any content..., although it is compatible with their having one. ...

[In order to understand computing mechanisms and how they work (as opposed to why they are built and how they are used), there is no need to invoke content.... [Piccinini, 2004b, 401–402, 404] (see also [Piccinini, 2006a, §2]

On this view, compatible with my own, it is the "internal" workings of the computer that count, not the external interpretation of its inputs and outputs (or even the external interpretation of its internal mechanisms or symbol manipulations). This is the sense in which a Civil War computer and a chess computer are computing "the same thing".

Note the hedge in the last sentence of Piccinini's quote: Cleland and Rescorla might be quite right in terms of their emphasis on why or how a particular computer or program is being used. That's an intentional/intensional aspect of computation, but doesn't necessarily vitiate the CTCT.

20. Related to the above point is Sloman's claim that one of the "primary features" of computers (and of brains) is "Coupling to environment via physical transducers" [Sloman, 2002, §5, #F6, pp. 17–18]. This allows for "perceptual processes that control or modify actions" and "is how internal information manipulation often leads to external behaviour". The former (perceptual input) would seem to be relevant to interactive computing (see Ch. 11). The latter—something also emphasized by [Shapiro, 2001]—is related to the issues that Cleland discusses. A computer without one or the other of these would be solipsistic. But computation can be kept separate from interaction with the world. It is not that the latter is unimportant or secondary, but that it is a separate thing. (So, insofar as Cleland-like arguments work against the CTCT, perhaps there are two versions of the CTCT: an "internal" or purely "syntactic" one, which is true, and an "external" or "semantic" one, that might be false.

In fact, [Sloman, 2002, §3.2, p. 9] says as much:

> In all these machines [that is, precursors to computers—WJR] we can, to a first approximation, divide the processes produced by the machine into two main categories: *internal* and *external*.

The former correspond to what [Piccinini, 2004b] and I would call purely syntactic computation; the latter correspond to the kinds of things that Cleland is more concerned with.

And [Sloman, 2002, §4.2, p. 12] believes that Turing machines are, therefore, less relevant to AI than "machines that are designed from the start to have interfaces to external sensors and motors. . . , unlike Turing machines which at least in their main forma re totally self contained, and are designed primarily to run in ballistic mode once set up with an initial machine table and tape configuration".

21. On Cleland's problem:
According to [Piccinini, 2006a, §2],

> In the mathematical theory of computation, abstract computing mechanisms are individuated by means of formal descriptions, some of which are called programs. Programs and other formal descriptions of computing mechanisms specify which inputs may enter the mechanism, how the inputs affect the internal states of the

> mechanism, and which outputs come out of the mechanism under
> which conditions. Inputs and outputs are strings of letters from a
> finite alphabet, often called symbols.
>
> In computability theory, symbols are typically marks on paper
> individuated by their geometrical shape (as opposed to their semantic
> properties). Symbols and strings of symbols may or may not be
> assigned an interpretation; if they are interpreted, the same string may
> be interpreted differently.... In these computational descriptions,
> the identity of the computing mechanism does not hinge on how the
> strings are interpreted.

By 'individuated', Piccinini is talking about how one decides whether what
appear to be two programs (say, one for a Civil War battle and one for a chess
match) are, in fact, two distinct programs or really just one program (perhaps
being described differently). Here, he suggests that it is not how the inputs
and outputs are interpreted (their semantics) that matters, but what the inputs
and outputs look like (their syntax). So, for Piccinini, the Civil War and chess
programs are the same. For Cleland, they would be different. For Piccinini, the
hollandaise-sauce program running on the Moon works just as well as the one
running on Earth; for Cleland, only the latter does what it is supposed to do.

So, the question "Which Turing machine is this?" has only one answer, given in
terms of its syntax ("determined by [its] instructions, not by [its] interpretations"
[Piccinini, 2006a, §2]. But the question "What does this Turing machine do?"
has $n + 1$ answers: one syntactic answer and n semantic answers (one for each
of n different interpretations).

One is tempted to ask which of these is the "right" way to look at it. But it is
important to see that both are right in different contexts. If one is concerned only
with the internal mechanisms, then Piccinini's way of looking at things is more
appropriate. If one wants to buy a chess program (but not a Civil War program)
or a hollandaise-sauce maker, then Cleland's way is more appropriate.

22. On [Rescorla, 2007]:

    (a)   A Turing machine manipulates syntactic entities: strings
           consisting of strokes and blanks. ...Our main interest is
           not string-theoretic functions but number-theoretic functions.
           We want to investigate computable functions from the natural
           numbers to the natural numbers. To do so, we must correlate
           strings of strokes with numbers. (p. 253.)

    This focuses the issues very clearly. Which are we really interested
    in: syntactic/symbolic/numeral computation or semantic/"ontological"/-
    number computation? Piccinini is interested in the former, Cleland in the
    latter. Hilbert, I think, was interested in the former, for, after all, we humans
    can only do the latter via the former. Is that a limitation? Perhaps, but
    it also gives us a freedom, because symbols can represent anything, not

just numbers. Can numbers represent things? Yes, but only insofar as we humans who use the numbers to represent things are thereby using them as if they were symbols.

Is computation just about numbers? Functions are not limited to numbers, so why should computation be thus limited? The broader notion of computation (not limited to numbers) includes number computation.

(b) Let's grant Rescorla's point that (semantic) computation of number-theoretic functions is distinct from (syntactic) computation of numeral/symbolic-theoretic functions in the Turing-machine sense. After all, Turing machines manipulate marks on paper, not numbers (at least, not directly). But now let's imagine a Platonic, number-theoretic, Turing-like machine as follows: The machine contains an infinite supply of "boxes", each of which can contain a natural number. (If you prefer, each box can only contain the number 1, and other natural numbers would be represented by sequences of boxes containing 1s.) The machine can look into a box to see what number (if any) is in it, and it can place any number in any box (putting a number in a box that already has a number in it just replaces the previous number). It can move one box to the left or one box to the right of its current location. In other words, it can do exactly what a normal Turing machine can do, except that, instead of dealing with marks on paper, it deals with numbers in boxes. If we consider an abstract Turing machine to be a (perhaps set-theoretically defined) mathematical structure, then both numeral-based and number-based Turing machines are (abstract) Turing machines. The CTCT can be interpreted to be saying that informal computation is Turing-machine computation in any of these three senses of Turing machine. Is there anything a number-theoretic Turing machine can or can't do that a numerical-based Turing machine can or can't do? If they are equally "powerful" (which I believe that they are), then I don't think that Rescorla's argument here is significant.

On second thought, maybe not. The version of a number-theoretic Turing machine that allows any real number to be placed in a box may be too powerful: How would you add two numbers? I don't see how you could unless you had addition as a primitive operation (which it isn't). On the other hand, if you only allow the number 1 to be placed in a box, and you represent other numbers by sequences of boxes of 1s, then you just have a version of the numeral-theoretic Turing machine with the number 1 in a box representing a mark on paper (or representing the numeral '1'). In that case, the number-theoretic Turing machine is just a very odd kind of numeral-theoretic Turing machine.

(c) A nice example of the "which Turing machine is this?/what does this Turing machine do?" contrast is this example of Rescorla's. (Note: The symbol '$\underline{x}$' represents a sequence of x strokes, where x is a natural number.)

> Different textbooks emply different correlations between Turing machines syntax and the natural numbers. The following three

correlations are among the most popular:

$$d_1(\underline{n}) = n.$$

$$d_2(\underline{n+1}) = n.$$

$$d_3(\underline{n+1}) = n, \text{as an input.}$$

$$d_3(\underline{n}) = n, \text{as an output.}$$

A machine that doubles the number of strokes computes $f(n) = 2n$ under d_1, $g(n) = 2n+1$ under d_2, and $h(n) = 2n+2$ under d_3. Thus, the same Turing machine computes different numerical functions relative to different correlations between symbols and numbers. [Rescorla, 2007, 254].

Do we really have one machine that does three different things? Piccinini would agree that there is only one machine. What it does (in one sense of that phrase) depends on how its input and output are interpreted, that is, on the environment in which it is working. In different environments, it does different things, or so Cleland would say.

(d)   . . . Turing's analysis concerns only *human* mechanical activity,
         not *general* mechanical activity. . . . For Turing, "computable"
         means "computable by a human," not "computable by some
         possible machine." [Rescorla, 2007, 257].

Exactly! This is what Hilbert was concerned with: How can we humans compute answers to certain problems? We can only directly compute on strings, not numbers (we can only indirectly compute on numbers via direct computation on strings).

(e) [Rescorla, 2007, 272–274] argues against exactly this point:

         One argument runs as follows: humans and computers directly
         manipulate symbols, not numbers; thus, what humans and
         computers *really* compute are string-theoretic functions, not
         number-theoretic functions.   . . . The argument is fallacious.
         . . . At best, the premise establishes that our computations of
         number-theoretic functions are *mediated* by our computations
         of string-theoretic functions. It does not follow that all we
         really. . . compute are string-theoretic functions. To conclude this
         would be analogous to the inference sometimes drawn by the
         British empiricists that, since our ideas mediate our perception
         of the external world, all we really perceive are our ideas.

Now, I happen to agree with the British empiricists, so I disagree with Rescorla. But Rescorla's broader point is that, to the extent that we want to compute over numbers, not numerals, we need a (computable) interpretation function between the numbers and the numerals.

(f) Against the Hilbertian point, Rescorla is less convincing. "It strkes me as rather slanted" (p. 273) is the best he can do. Essentially, his point comes down to this: Even if Hilbert, Turing, et al., really were only interested in syntactic computation, that doesn't mean that *we* shouldn't also be interested in semantic computation.

ADD RESCORLA TO RECOMMENDED READING FOR THIS CHAPTER? AND PICCININI?

(g)   . . . Turing's analysis concerns human *mechanical* activity, not human cognition in general. [Rescorla, 2007, 257].

To a certain extent, this is correct. The extent to which is might be misleading is this: It remains an open question whether human cognition in general can be completely captured by human mechanical activity. This is the topic of computational cognitive science, which we'll explore in Ch. 19. Rescorla continues:

> [Turing's] constraints. . . do not purport to goven all possible processes for determining some function's value. Imagine a computor [that is, a human clerk who is performing a computation] who possesses a mystgerious cognitive faculty, which enables him [sic] to determine some uncomputable function's value upon any input. . . . By employing the faculty, the computor introduces an essentially *nonmechanical* element into his mathematical activity. He implements a nonalgorithmic cognitive strategy. . . . But we can expunge them from our account of computation.

This faculty is precisely the kind of oracle that Turing introduced in his doctoral thesis. But computation relative to an oracle can still be considered to be a kind of computation, only one that adds to the set of primitive operations such an extra faculty. We'll return to this in Ch. 11.

(h) [Copeland and Proudfoot, 2010] is a reply to [Rescorla, 2007], and is a good paper in general on the problem of Turing machines being syntactic. They argue that " 'deviant encodings'. . . that appear to enable Turing machines to perform 'impossible' tasks, such as solving the halting problem" (p. 247, col. 1) can be avoided by respecting Turing's restriction that the input tape must be empty [Turing, 1936, 232]:

> In effect, the blank-tape restriction formalizes the requirement. . . that the computer be allowed no access to numbers or statements produced by an oracle or by a human mathematician working creatively. The blank-tape restriction provides a simple and elegant solution to the problem of deviant I[nput]-encodings. *All* I-encodings are precluded, deviant or otherwise. Prior tape-markings—inputs of any form—are simply banned where the objective is to provide a formal analysis of computability. (p. 250, col. 1)

Note that this blocks a Turing machine from being a hypercomputer (see Ch. 11.

23. Here is another nice example for the Cleland issue:

    a loom programmed to weave a certain pattern will weave that pattern regardless of what kinds of thread it is weaving. The properties of the threads make no difference to the pattern being weaved. In other words, the weaving process is insensitive to the properties of the input. [Piccinini, 2008, 39]

    As Piccinini goes on to point out, the output might have different colors depending on the colors of the input threads, but the *pattern* will remain the same. The pattern is internal; the colors are external, to use the terminology above. If you want to weave an American flag, you had better use red, white, and blue threads in the appropriate ways. But even if you use puce, purple, and plum threads, you will weave an American-flag *pattern*. Which is more important: the pattern or the colors? That's probably not exactly the right question. Rather, the issue is this: If you want a certain pattern, this program will give it to you; if you want a certain pattern with certain colors, then you need to have the right inputs (you need to use the program in the right environment). (This is related to issues in the philosophy of mathematics concerning "structuralism": Is the pattern, or structure, of the natural numbers all that matters? Or does it also matter what the natural numbers in the pattern "really" are? CITE)

24. This could go either in this chapter or earlier (Ch. 7?):
    It's worth comparing the explication of the informal (or "folk"?) notion of algorithm as a Turing machine (or recursive functions, etc.), with the attempt to define life in scientific terms.

    (a) As with any attempt at a formal explication of an informal concept (as we discussed in Chs. 3 and 9), there is never any guarantee that the formal explanation will satisfactorily capture the informal notion (usually because the informal notion is informal or fuzzy). The formal explication might include some things that are, pre-theoretically at least, not obviously included in the informal concept, and it might exclude some things that are, pre-theoretically, included. Many of the attempts to show that there is something wrong with the CTCT fall along these lines. Similarly, in the case of life, we have alleged counterexamples to various scientific definitions such as Artificial Life, etc.

        One difference between the two cases is this: There are many non-equivalent scientific definitions of life. But in the case of algorithm, there are many equivalent formalizations: Turing machines, recursive functions, lambda-calculations, etc. What might have happened to the informal notion if these had not turned out to be equivalent?

        For discussion of some of these issues in the case of life, see [Machery, 2012].

(b) This issue is also related to Darwin's and Turing's strange inversion. Compare this quote:

> The possibility of the deliberate creation of living organisms from elementary materials that are not themselves alive has engaged the human imagination for a very long time. [Lewontin, 2014, 22]

to this paraphrase:

> The possibility of the deliberate creation of intelligent behavior from elementary operations that are not themselves intelligent has engaged the human imagination for a very long time.

25. On the Cleland issue of the relevance of the environment/external world:

> Since we can arbitrarily vary inherited meanings relative to syntactic machinations, inherited meanings do not *make a difference* to those machinations. They are imposed upon an underlying causal structure. [Rescorla, 2014a, 181]

On this view, the hollandaise-sauce-making computer does its thing whether it's on the earth or the moon (whether its output is hollandaise sauce or not). (Perhaps its output is some kind of generalized, abstract hollandaise-sauce type, whose implementations/instantiations/tokens on the moon are some sort of goop, but whose implementations/instantiations/tokens on earth are what are normally considered to be (successful) hollandaise sauce.) And the chess/Civil War computers are doing the same thing.

26. How do you know when a thing "just begins to boil"? How can you be sure that the milk has scorched but not burned? Or touch something too hot to touch, or tell firm peaks from stiff peaks? How do you define "chopped"? [Gopnik, 2009b, 106]

27. Hopcroft & Ullman (CITE) distinguish a "procedure", which they vaguely define (their terminology!) as "a finite sequence of instructions that can be mechanically carried out, such as a computer program" (to be formally defined in their Ch. 6), from an "algorithm", which they define as "A procedure which always terminates". (See pp. 2–3 of their text.)

28. [Copeland and Sylvan, 1999, 46] (see also [Copeland, 1997]) distinguish between two interpretations of the CTCT. The one that they claim was actually stated by Church and by Turing "concerned the functions taht are in principle computable by an *idealised human being unaided by machinery*". This one, they claim, is correct. The other interpretation is "that the class of well-defined computations is exhausted by the computations that can be carried out by Turing machines".

So, one possible objection to Cleland is that cooking (for example) is not something that can be carried out by "an idealized human being unaided by

machinery", hence the failure of a hollandaise sauce recipe on the moon is irrelevant to the correct interpretation of the CTCT.

The point about the other interpretation is what leads to hypercomputation theories (of which the most plausible are interaction machines and trial-and-error machines).

29. [Church, 1936a, 40, my italics] calls his 1936 statement of what we now name "Church's thesis" "a *definition* of the commonly used term 'effectively calculable' ".

30. On improvisation and interactive computing:

> Music is indeed a useful analogy [to computer programs], because it's a flexible form that moves through time. But certainly computer porgrams are much richer in this respect, becasue programs not only move through time, they also interact with people and they can even modify themselves. [Knuth, 2001, 167]

Of course, improvisational music (for example, certain kinds of jazz) also have tht kind of interactive nature.

31. Modifications of the CTCT, such as the physical CTCT, really aren't concerned with the kind of computation that Hilbert cared about, any more than Euclid would have cared about geometric constructions that went beyond compass and straightedge.

32. An interesting argument to the effect that Church's Thesis should be distinguished from Turing's Thesis can be found in [Rescorla, 2007]: Church's Thesis asserts that intuitively computable *number-theoretic* functions are recursive. Turing's Thesis asserts that intuitively computable *string-theoretic* functions are Turing-computable. We can only combine these into a Church-Turing Computability Thesis by adding a requirement that there be a computable semantic interpretation function between strings and numbers. (And this raises the further question whether that semantic interpretation function is intuitively computable or recursive.)

33. [Shapiro, 2001] argues that computer science is the natural science of procedures. In support of this claim, we might consider this related claim of [Rescorla, 2014b, §2, p. 1279]:

> To formulate. . . [Euclid's GCD algorithm], Knuth uses natural language augmented with some mathematical symbols. For most of human history, this was basically the only way to formulate mechanical instructions. The computer revolution marked the advent of rigorous *computational formalisms*, which allow one to state mechanical instructions in a precise, unambiguous, canonical way.

In other words, computer science developed formal methods for making the notion and expression of procedures mathematically precise. That's what makes is a science of procedures.

34.   ...if we represent the natural number n by a string of n consecutive 1s, and start the program with the read-write head scanning the leftmost 1 of the string, then the program,

q_0 1 1 R q_0
q_0 0 1 R q_1

will scroll the head to the right across the input string, then add a single '1' to the end. It can, therefore, be taken to compute the successor function. [Aizawa, 2010, 229] But if the environment (the tape) is not a string of *n* '1's followed by a '0', then this does *not* compute the successor function. Compare this to Cleland's Hollandaise sauce recipe being executed on the moon.

35. [Egan, 2010] "argue[s] that representational content is to be understood as a *gloss* on the computational characterization of a cognitive process" (p. 253, Abstract). She cites David Marr's [Marr, 1982] work on a computational theory of visual perception. According to Egan's interpretation of Marr, "a component of the early visual processing" sytem filters "the retinal image"; "the theoretically important characterization, from a computational point of view, is a *mathematical* description: the device computes the Laplacean convolved with the Gaussian.... *As it happens*, it takes as input light intensity values at points in the retinal image, and calculates the rate of change of intensity over the image" [Egan, 2010, 255, col. 1, my italics]. But, considered solely as a "computational device, it does not matter that input values represent *light intensities* and output values the rate of change of *light intensity*. The computational theory characterizes the visual filter as a member of a well understood class of mathematical devices that have nothing essentially to do with the transduction of light" [Egan, 2010, 255, cols. 1–2, original italics].

Compare this to the chess-Civil War example: To paraphrase, the theoretically important characterization from a computational point of view is a mathematical description: the device computes some mathematical function that, as it happens, can be interpreted as a chess match or else as a Civil War battle; but, considered solely as a computational device, it does not matter that input values represent (say) chess moves or battle positions—the computational theory characterizes the device as a member of a well understood class of mathematical devices that have nohting essentially to do with chess or the Civil War. It is "environment neutral" [Egan, 2010, 256, col. 1].

[Egan, 2014, 122] points out, with respect to [Shadmehr and Wise, 2005]'s "characterization of the motor-control mechanism [for moving our hand to grasp an object—see comments below] allows us to see that a mariner who knew the distance and bearing from his home port to his present location and the

distance and bearing from his home port to a buried treasure could perform the *same* computation—vector subtraction—to compute the course from his present location to the treasure." Similarly, a Civil War buff might be able to use the same computer program that describes a particular battle to help her win a chess game:

> A crucial feature of the function-theoretic characterization [that is, the characterization that focuses solely on the mathematical function being computed and not on the purpose or external environment] is that it is 'environment neutral': the task is characterized in terms that prescind from the environment in which the mechanism is normally deployed. The mechanism described by Marr would compute the Laplacean of a Gaussian even if it were to appear (*per mirabile*) in an environment where light behaves very differently that it does on earth, or as part of an envatted brain. . . . [Egan, 2014, 122]

This relates to Cleland's hollandaise sauce example: Egan says that the visual filter "would compute the same mathematical function in any environment, *but only in some environments would its doing so enable the organism to see*" [Egan, 2010, 256, col. 2, my italics]. Similarly, Cleland's recipe would compute the same (culinary?) function in any environment, but only on Earth (and not on the Moon) would it doing so result in Hollandaise sauce.

> When the computational characterization is accompanied by an appropriate cognitive interpretation, in terms of distal objects and properties, we can see how the mechanism that computes a certain mathematical function can, in a particular context, subserve a cognitive function such as vision. . . . [Egan, 2010, 256, col. 2] (see also p. 257, cols. 1-2)

That is, we can see what it implements by looking at its semantic interpretation. (We will discuss this further in §14.4.)

If the program is put in a chess environment, it plays chess; if it is put in a Civil War environment, it simulates a battle. This issue is also related to Putnam's Twin Earth puzzle (MAYBE PUT THIS IN A FOOTNOTE IF IT IS NOT DISCUSSED ANYWHERE ELSE): If I use the word 'water' on Earth, it denotes H_2O; if I (or twin-I) use it on Twin Earth, it denotes XYZ.

Does this mean that algorithms are not intentional, as Hill argues when she says that all algorithms must have the form: To accomplish task X, do this.

[Egan, 2014] takes up these themes again, in more detail. She gives another example, due to [Shadmehr and Wise, 2005], concerning how we are able to move our hand to grasp an object. Note that our brain presumably actually

computes such mathematical functions as those in this example, and *thereby* our hand moves correctly; there is a causal connection between the brain's computation and our hand's motion. Recall our earlier discussion of whether the solar system computes Kepler's laws of motion (§9.7.2). In what ways are these two cases—the brain computing various functions that cause (or, at least, result in) our hand to move in a certain way, and the solar system (possibly) computing various functions that (possibly) result in (if not cause) the planets to move as they do. Is it possible that our brain causes our hands to move correctly but doesn't really compute those functions (in the same way that the solar system doesn't really compute Kepler's laws)? It seems reasonable to say, in the brain case, that the brain really does compute, but not so reasonable to say, in the solar system case, that the solar system computes. We seem to be willing to be scientific realists in the one case, but not the other. What, if anything, is the difference between the two cases?

36. On the conjunction-disjunction issue:

    (a) Be sure to cite Shagrir.

    (b) [Sprevak, 2010, §3.3, pp. 268–269] has a similar example.

10.7   Further Sources of Information

1. Books and Essays:

   (a) Abramson, Darren (2011), "Philosophy of Mind Is (in Part) Philosophy of Computer Science", *Minds and Machines* 21: 203–219.

       • Argues that the CTCT is relevant to questions about the Turing test for AI.

   (b) Blass, Andreas; & Gurevich, Yuri (2003), "Algorithms: A Quest for Absolute Definitions", *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, No. 81 (October): 195–225.

       • Pre-print accessed 22 April 2014 from:
         http://research.microsoft.com/en-us/um/people/gurevich/opera/164.pdf
       • Reprinted in Olszewksi et al. 2006 (see below), pp. 24–57.
       • "The [Church-Turing] thesis was a great step toward understanding algorithms, but it did not solve the problem what an algorithm is."

   (c) Bringsjord, Selmer (1993), "The Narrational Case against Church's Thesis", accessed 17 April 2014 from:
       http://homepages.rpi.edu/~brings/SELPAP/CT/ct/

   (d) **Cleland, Carol E. (2004), "The Concept of Computability",** *Theoretical Computer Science* **317: 209–225.**

   (e) Daly, Ian (2010), "Just Like Mombot Used to Make", *New York Times*, Dining section (24 February): D1, D5.

       • Accessed 8 November 2012 from:
         http://www.nytimes.com/2010/02/24/dining/24robots.html
       • On robots that not only follow recipes, but actually cook the meals: "Fear not, humans, these robots are here to be our friends. To prove it, they serve you food."
       • Interesting reading in connection with what Cleland 1993 and Preston 2006 have to say about the computability of recipes.

   (f) **Davis, Martin (2004), "The Myth of Hypercomputation", in C. Teuscher (ed.),** *Alan Turing: The Life and Legacy of a Great Thinker* **(Berlin: Springer): 195–212.**

       • Accessed 15 May 2014 from:
         http://www1.maths.leeds.ac.uk/~pmt6sbc/docs/davis.myth.pdf
       • Contains a good discussion of the Church-Turing Thesis.

   (g) Folina, Janet (1998), "Church's Thesis: Prelude to a Proof", *Philosophia Mathematica* 6(3): 302–323.

       • Argues against [Gandy, 1988, Mendelson, 1990, Shapiro, 1993, Sieg, 1994] (and others) that the thesis is unprovable but true.

   (h) Kearns, John (1997), "Thinking Machines: Some Fundamental Confusions", *Minds and Machines* 7(2): 269–287.

       • A discussion of the Church-Turing Computability Thesis, arguing that it "has not interesting consequences for human cognition" because "carrying out a procedure is a purposive, intentional activity. No actual machine does, or can do, as much."

(i) Kripke, Saul A. (2013), "The Church-Turing 'Thesis' as a Special Corollary of Gödel's Completeness Theorem", in B. Jack Copeland, Carl J. Posy, & Oron Shagrir (eds.), *Computability: Turing, Gödel, Church, and Beyond* (Cambridge, MA: MIT Press): 77–104.

- An argument by one of the great logicians that the CTCT can be proved. Also contains a lot of useful historical remarks.

(j) Kroes, Peter (2010), "Engineering and the Dual Nature of Technical Artefacts", *Cambridge Journal of Economics* 34: 51–62.

- Argues that "technical artefacts are physical structures with functional properties"

(k) Olszewski, Adam; Woleński, Jan; & Janusz, Robert (eds.) (2006), *Church's Thesis after 70 Years* (Frankfurt: Ontos Verlag).

- Includes essays by Selmer Bringsjord, Carol Cleland, B. Jack Copeland, Janet Folina, Yuri Gurevich, Andrew Hodges, Charles McCarty, Elliott Mendelson, Oron Shagrir, Stewart Shapiro, and Wilfrid Sieg, among many others.

(l) Petroski, Henry (2010), "Occasional Design", *American Scientist* 98(1) (January-February): 16–19.

- "An everyday challenge provides lessons in the processes of planning and execution"
- A nice follow-up to Preston 2006.

(m) Rey, Georges (2012), "The Turing Thesis vs. the Turing Test", *The Philosopher's Magazine* 57 (2nd Quarter): 84–89.

(n) Shapiro, Stewart (2013), "The Open Texture of Computability", in B. Jack Copeland, Carl J. Posy, & Oron Shagrir (eds.), *Computability: Turing, Gödel, Church, and Beyond* (Cambridge, MA: MIT Press): 153–181.

- A very readable discussion of the provability of the CTCT; of the nature of the "informality", "vagueness", or "open texture" of the notion of computability; and of the difference between human, mechanical, and mathematical computability, with observations on many of the other readings discussed or mentioned in this chapter.

(o) **Shapiro, Stuart C. (2001), "Computer Science: The Study of Procedures"**

- Accessed 8 November 2012 from:
  http://www.cse.buffalo.edu/~shapiro/Papers/whatiscs.pdf

(p) Sieg, Wilfried (1994), "Mechanical Procedures and Mathematical Experience", in George Alexander (ed.), *Mathematics and Mind* (New York: Oxford University Press): 71–117.

- Accessed 10 December 2013 from:
  http://repository.cmu.edu/cgi/viewcontent.cgi?article=1248&context=philosophy
- §§2–3 are especially good on the history of the Church-Turing Computability Thesis.

(q) Sieg, Wilfried (2000), "Calculations by Man and Machine: Conceptual Analysis", *Technical Report CMU-PHIL-104* (Pittsburgh, PA: Carnegie-Mellon University Department of Philosophy); http://repository.cmu.edu/philosophy/178.

- The (informal) "calculability of number-theoretic functions" is first analyzed into calculability by humans "satisfying boundedness and locality conditions"; that is analyzed into "computability by string machine"; and the latter is analyzed into computability by a Turing machine. Sieg identifies "Turing's thesis" as the analysis of the first of these by the last.

(r) **Soare, Robert I. (2009), "Turing Oracle Machines, Online Computing, and Three Displacements in Computability Theory",** *Annals of Pure and Applied Logic* **160: 368–399.**

  - Preprint accessed 2 July 2014 from:
    http://www.people.cs.uchicago.edu/ soare/History/turing.pdf
    and published version accessed 2 July 2014 from:
    http://ac.els-cdn.com/S0168007209000128/1-s2.0-S0168007209000128-
    main.pdf?_tid=8258a7e2-01ef-11e4-9636-
    00000aab0f6b&acdnat=1404309072_f745d1632bb6fdd95f711397fda63ee2

  - §§2, 3, and 12 ("Origins of Computabilty and Incomputablity", "Turing Breaks the Stalemate", and "Renaming It the 'Computability Thesis' ") contain a good summary of the history of the CTCT.

(s) Special Issue on Church's Thesis (1987), *Notre Dame Journal of Formal Logic* 28(4) (October).

  - Accessed 9 April 2014 from:
    http://projecteuclid.org/euclid.ndjfl/1093637642

(t) Turner, Raymond (2011), "Specification", *Minds and Machines* 21(2): 135–152.

2. Websites:

  - Does a function in mathematics *change* anything?
    Accessed 8 November 2012 from http://www.askphilosophers.org/question/1877

  - Humorous Algorithms
    Accessed 8 November 2012 from:
    http://www.cse.buffalo.edu/∼rapaport/510/alg-cartoons.html

Chapter 11

The Church-Turing Computability Thesis: II. What Is Hypercomputation?

Speculation that there may be physical processes—and so, potentially, machine-operations—whose behaviour cannot be simulated by the universal Turing machine of 1936 stretches back over a number of decades. Could a machine, or natural system, deserving the name 'hypercomputer' really exist? Indeed, is the mind—or the brain—some form of hypercomputer? [Copeland, 2002, 462][1]

We now know both that hypercomputation (or super-recursive computation) is mathematically well-understood, and that it provides a theory that according to some accounts for some real-life computation...[is] better than the standard theory of computation at and below the "Turing Limit." ...[S]ince it's mathematically *possible* that human minds are hypercomputers, such minds *are* in fact hypercomputers. [Bringsjord and Arkoudas, 2004, 167]

The editors have kindly invited me to write an introduction to this special issue devoted to "hypercomputation" despite their perfect awareness of my belief that there is no such subject. [Davis, 2006c, 4]

According to a 1992 paper, a computer operating in a Malament-Hogarth spacetime or in orbit around a rotating black hole could theoretically perform non-Turing computations.
—http://en.wikipedia.org/wiki/Hypercomputation
(accessed 26 May 2014). [At least, "it's something", as someone once posted in reply.]

---

[1]Recall our discussion of the mind-body problem, §2.8.

11.1   Recommended Readings:

1. Required:

   (a) Copeland, B. Jack (2002), "Hypercomputation", *Minds and Machines* 12(4) (November): 461–502.

      - A good overview and survey.
      - Read pp. 461–465.
      - Read §1.5, on Putnam & Gold
      - Read §1.7, on Boolos & Jeffrey
      - Read §1.13, on Kugel
      - Read §1.18–§1.18.1, on Penrose, Turing, & Gödel
      - Read §1.24, on Cleland
      - Read all of §§2–3

   (b) Wegner, Peter (1997), "Why Interaction Is More Powerful than Algorithms", *Communications of the ACM* 40(5) (May): 80–91.

      - Many of Wegner's papers are online in various formats at his homepage:
        http://www.cs.brown.edu/people/pw/home.html
        (accessed 15 May 2014).

   (c) Kugel, Peter (2002), "Computing Machines Can't Be Intelligent (. . . and Turing Said So)", *Minds and Machines* 12(4) (November): 563–579.

   (d) Davis, Martin (2004), "The Myth of Hypercomputation", in C. Teuscher (ed.), *Alan Turing: The Life and Legacy of a Great Thinker* (Berlin: Springer): 195–212.

      - Accessed 15 May 2014 from:
        http://www1.maths.leeds.ac.uk/~pmt6sbc/docs/davis.myth.pdf

2. Very Strongly Recommended:

   (a) Eberbach, Eugene; & Wegner, Peter (2003), "Beyond Turing Machines", *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, No. 81 (October): 279–304

      - Accessed 8 November 2012 from:
        http://www.eatcs.org/images/bulletin/beatcs81.pdf
      - Another useful survey, but it opens with some seriously misleading comments about the history of computation and the nature of logic.[2]

   (b) On Putnam-Gold machines:

      - John Case's COLT Page
        accessed 8 November 2012 from: http://www.eecis.udel.edu/~case/colt.html

---

[2] For example, contrary to what the authors say, (1) [Turing, 1936] was *not* "primarily a mathematical demonstration that computers could not model mathematical problem solving" (p. 279) (although it did include a mathematical demonstration that there were *some* mathematical problems that a computer could not solve); and (2) Hilbert did *not* take the *Entscheidungsproblem* "as a basis that all mathematical problems could be proved true or false by logic" (p. 279), simply because mathematical problems cannot be "proved true or false" (rather, mathematical propositions might be provable or not, and they might be true or false, but truth and falsity are semantic notions incapable of proof, whereas provability is a syntactic notion).

- Nowak, Martin A.; Komarova, Natalia L.; & Niyogi, Partha (2002), "Computational and Evolutionary Aspects of Language", *Nature* 417 (6 June): 611–617.

  – Accessed 4 June 2014 from:
    http://people.cs.uchicago.edu/∼niyogi/papersps/NKNnature.pdf

11.2   Introduction

> [A] computer linked with a human mind is a more powerful tool than
> an unassisted human mind. One hope of many computer scientists is to
> demonstrate…that the computer/human team can actually accomplish far
> more than a human alone. [Forsythe, 1967a, 3].

One might also ask whether that "team" could accomplish far more than a *computer*
alone, say by interacting with the computer while it is computing [Lohr, 2013,
Steed, 2013].

We have seen that it can be argued that computer science is the systematic study of
computing, that computing is the mathematical study of algorithms, and that algorithms
are best understood mathematically in terms of Turing machines (or anything logically
equivalent to Turing machines).

But *are* they best understood that way? We saw in the previous chapter that it can
be argued that there might be other kinds of algorithms that are not computable by a
Turing machine. In this chapter, we will look at some of the arguments for this point
of view and at some of the kinds of algorithms that are allegedly not computable by a
Turing machine.

11.3   Copeland's Theory of Hypercomputation

In a series of papers, the philosopher Jack Copeland has contrasted Turing-
machine computation with what he has called "hypercomputation" (see especially
[Copeland, 2002]). He describes a Turing-machine computation as a computation of
"functions or numbers…with paper and pencil [such that each digit of the numerical
representation of the number or each input-output pair of the function is produced]
in a finite number of steps by a human clerk working effectively". And he defines
'hypercomputation' as "the computation of functions or numbers that cannot be"
computed by a Turing machine [Copeland, 2002, 461].

So, if hypercomputable functions or numbers cannot be computed by a Turing
machine, how can they be computed? He cites the following possibilities:

- The primitive operations of the computation might not be executable by a human
  working alone, in the way that Turing's 1936 paper described.

  MOVE THIS TO A SECTION ON ORACLE MACHINES:
  For example, in Turing's Ph.D. dissertation at Princeton (which he began *after*
  his classic 1936 paper was published, and which was completed in 1938; Church
  was his dissertation advisor), Turing described what he called "oracle" machines,
  or *o*-machines. These were an extension of Turing machines that allowed
  additional primitive operations that are not computable by an ordinary Turing
  machine (that is, by what Turing called an *a*-machine). An oracle machine
  consists of a Turing machine plus a "black box" that outputs the value of
  a function that is not computable by a Turing machine. If a function *g* is
  computable by an oracle machine with an oracle that outputs the value of a

(non-Turing-machine-computable) function f, then it is said that g is computable *relative to f*.

- Some data could be supplied during the computation, rather than being pre-stored on the Turing machine tape. This is what happens during "interactive" computing.

MOVE THIS TO A SECTION ON INTERACTIVE COMPUTING:
Peter Wegner [Wegner, 1997] has argued that "interaction machines" (or "coupled" Turing machines) are strictly more powerful than Turing machines. Turing machines accept no new input while computing. But interaction machines are "coupled" to a finite number of streams of input symbols (and can have a finite number of output channels). If an interaction machine halts, then it can be simulated by a universal Turing machine (by the Substitution Property discussed below). But if it does not halt (as is the case, for example, for an ATM;[3] ATMs only halt when they are broken or being repaired), then, if the unending input stream is a number computable by a universal Turing machine (e.g., π), then the interaction machine can also be simulated by a universal Turing machine, but, if the unending input stream is *not* computable by a universal Turing machine, then the interaction machine is a hypercomputer.

But is it? Or is it just an oracle machine? We will see in a moment why it is not obvious that oracle-machine computation is "hyper" in any interesting sense.

Why might such a non-halting, non-computable, interaction machine be a hypercomputer? Its input stream might be random, in which case, it would not be computable by a Turing machine (this was proved by Church) (but Davis objects to this; see below WHERE?). Or its input stream might not be random, but just non-computable.

Here is a potential counterexample to the idea that interactive computation is not computable by a Turing machine. In the theory of (Turing machine) computation, there is a theorem called the Substitution Property:

$$(\exists \text{ recursive function } s)(\forall x, y, z \in \mathbb{N})[\phi_x(y, z) = \phi_{s(x,y)}(z)]$$

In this statement, '$\phi_a(b)$' represents the ath Turing machine (in some numbering scheme for Turing machines), and b is its input. (How do you number Turing machines? One way is to take the numbering that Turing used and that we discussed in §8.12.)

So, the Substitution Property says that there exists a recursive function s (i.e., a function s that is computable by a Turing machine) that has the following property: For any three natural numbers x, y, z, the following is true: The xth algorithm (the xth Turing machine), when given both y and z as inputs produces the same output that the $s(x, y)$th algorithm (that is, the $s(x, y)$th Turing machine) does when given z as input.

Here is another way to say this: First, enumerate all of the partial recursive functions. Second, let ϕ_x be the xth partial recursive function. Suppose that it

---

[3]That is, an automated teller machine.

takes two inputs: x and y (another way to say this is that its *single* input is the ordered pair (y,z)). Then there exists *another* partial recursive function $\phi_{s(x,y)}$— that is, we can find another partial recursive function that depends on ϕ's (two) inputs (and the dependence is itself a recursive function, s)—such that $\phi_{s(x,y)}$ is input-output–equivalent to ϕ_x when y is fixed, and which is such that $\phi_{s(x,y)}$ is a function with y (that is, with part of ϕ_x's input) *stored internally as data.*

Here is an example of these two kinds of Turing machines (with a program for $\phi_x(y,z)$ on the left and a program for $\phi_{s(x,y)}(z)$ on the right:

begin	**begin**
input(y,z);	*constant* $:= y$;
output $:=$ process(y,z);	input(z);
print(*output*)	*output* $:=$ process(*constant*, z);
end.	print(*output*);
	end.

In other words, data can be stored effectively (that is, algorithmically) in programs; the data need not be input from the external world.

In this way, any interactive program could, in principle, be shown to be logically equivalent to a non-interactive program. That is, any interactive program can be simulated by an "ordinary" Turing machine by pre-storing the external input.

The catch here is whether you need to know "in advance" what the external input is going to be. But the Substitution Property theorem does seem to say that, once you know what that input is, you need only an ordinary Turing machine, not an interactive hypercomputer.

- The constraint of data as symbols on paper could be relaxed.

  MOVE THIS TO A SECTION ON CLELAND?:
  For example, Cleland's "mundane" hollandaise-sauce recipe does not take such symbols as either input or output; instead, the input are certain food ingredients and the output is a certain food preparation.

- The computer's rules could be allowed to change during the computation.

  MOVE THIS TO A SECTION THAT ELABORATES ON IT?:
  But what about self-modifying programs? Surely, they are computable by a Turing machine, yet they can change their own programs. More precisely, a universal Turing machine with a program P stored on its tape could, as part of executing P, change P, yet continue executing. This is a consequence of the idea that program and data are interchangeable.

- The constraints about a finite number of steps or a finite amount of time can be relaxed.

  MOVE THIS TO A SECTION THAT ELABORATES ON IT?:
  This would be the case with what has been called a "Zeus" machine or an accelerating Turing machine [Boolos and Jeffrey, 1974]. Turing placed

"temporal patterning" restrictions on his model because he wanted to model *human* computation. If we relax those restrictions, then we could devise a machine that would calculate each digit of a real number's decimal expansion in half of the time of the previous digit's calculation. Thus, an infinite calculation, including the Halting Problem, could be computed in a finite amount of time. (As Bertrand Russell observed of a very similar example [Russell, 1936, 143], this is not *logically* impossible, just "medically" impossible!)

There are several basic questions that need to be considered:

- *Are* these really alternative models of *humanly* effective procedures? (And does 'effective' need to mean "*humanly* effective"?)

- *Are* they *counterexamples* to the Church-Turing Computability Thesis?
  Or are they just *other models* of computation (perhaps of some more general notion of computation)?

- How realistic are they? Can such hypercomputers really exist?

- Is the mind or brain a hypercomputer (instead of a Turing-machine computer)?

Martin Davis thinks that most of these hypercomputers are either wrong or just plain silly [Davis, 2004], [Davis, 2006c]: Consider machines that allow for non-computable input: Of course, if you allow for non-computable input, you should expect to be able to get non-computable output. (Compare the "garbage in/garbage out" principle: If you allow for incorrect input, you should expect to be able to get incorrect output.)

But might there be some intuitively effective, yet not Turing-machine–effective, computations that don't take non-computable input? One potential example of this are so-called trial-and-error machines, to which we now turn.

11.4 Trial-and-Error Machines

One example of such a machine is what are called "trial-and-error" machines [Putnam, 1965], [Gold, 1967], [Hintikka and Mutanen, 1997]. Here, no constraints need to be relaxed; rather, we change our interpretation of what counts as the output of the computation.

Recall our discussion of the Halting Problem: We contrasted two algorithms:

```
algorithm H₁(C, i):
begin
    if C(i) halts
        then output 'halts'
        else output 'loops'
end.
```

This version of the algorithm *can* be converted to the self-referential $H*$ (see §7.8) and thereby used in order to show that the Halting Problem is not Turing-machine computable.

But there is another version of this algorithm, which we saw could *not* be so converted:

> **algorithm** $H_2(C,i)$:
> **begin**
>   output 'loops'; {i.e., make an initial guess that C loops}
>   **if** $C(i)$ halts
>   **then** output 'halts'; {i.e., revise your guess}
> **end.**

H_2 is an example of a "trial-and-error" machine: You make an initial guess about the desired output, and then keep running the program on a number of "trials". If the trials produce "errors" or don't come up with a desired response, then continue to run more trials.

In general, a trial-and-error machine is a Turing machine with input i that outputs a sequence of 'yes'/'no' responses such that it is the *last* output that is "the" desired output of the machine (rather than the first, or only, output). But you don't allow any way to tell effectively if you've actually achieved the desired output, that is, if the machine has really halted.

As Hilary Putnam says,

> [I]if the machine has most recently printed "yes", then we know that the... [input has been accepted] *unless the machine is going to change its mind*; but we have no procedure for telling whether it will change its mind or not. The... [problems] for which there exist decision procedures in this widened sense are decidable by "empirical" means—for, if we always "posit" that the most recently generated answer is correct, we will make a finite number of mistakes, but we will eventually get the correct answer. (Note, however, that even if we have gotten to the correct answer (the end of the finite sequence) we are never *sure* that we have the correct answer.) [Putnam, 1965, 49, italics in original]

Such a machine, as we've seen, can "decide"—or "compute"—the Halting Problem!

11.4.1   Does Intelligence Require Trial-and-Error Machines?

NOTE: THIS SECTION NEEDS TO BE REVISED AFTER A CAREFUL RE-READING OF KUGEL 2002.

Peter Kugel [Kugel, 2002] has argued that trial-and-error machines can give a computer "initiative", allowing them to achieve real AI. If so, then this would show, *not* that AI is not computable, but that it is not a computable *function*. There are certain aspects of Kugel's argument that make his version of hypercomputation more plausible than some of the others.

Kugel argues that AI will be possible, using digital computers (and not requiring fancy, quantum computers or other kinds of non-digital computers), but by using those digital computers in a non-computational way. He begins his argument by observing

that intelligence in general, and artifical intelligence in particular, requires "initiative", which he roughly identifies with the absences of "discipline", defined, in turn, as the ability to follow orders. Thus, perhaps, intelligence and AI require the ability to break rules! Computation, on the other hand, requires this kind of discipline (after all, as we have seen, computation certainly includes the ability to follow orders, or, at least, to behave in accordance with orders).

Computers compute. But, Kugel points out, they can do other things, too. For example, they can "change their mind". Turing 1947 (CITE) claimed that infallible entities could not be intelligent, but that fallibility allows for intelligence. And, according to Kugel, [Turing, 1950] said that computers could probably be programmed to *fake* intelligence, but probably could *not* be programmed to *be* intelligent. We will see if this is a reasonable interpretation of [Turing, 1950] when we look at the philosophy of AI in Chapter 19, but I will note here that most interpretations of [Turing, 1950] take it that a computer might be able to fool a human interrogator into confusing a human with a computer *because* both of them would be intelligent!

The core of Kugel's argument is the observation that a distinction can be made between a Turing machine and Turing *machinery*. Turing machinery with a finite memory (either a finite tape or even no tape at all) is not a Turing machine but a "finite automaton", a mathematical structure with less than the power of a Turing machine.

SAY MORE ABOUT FINITE AUTOMATA.

In a Turing machine, the first output is the result of its computation. But there is nothing preventing the use of Turing *machinery* (such as a finite automaton) and taking the *last* output of its operation as its result. (You can't say that the operation of such a finite automaton is "computation", if you accept the CTCT, which identifies computation with the operation of a Turing machine.) This, as we have seen, is a Putnam-Gold trial-and-error machine.

Kugel next argues that Turing-machine computation does not suffice for intelligence, on the grounds that, if it did, it would not be able to survive! So, suppose (by way of *reductio*) that Turing-machine computation did suffice for intelligence. And suppose that a mind is a universal Turing-machine with "instincts" (that is, with some built-in programs) and that is capable of learning (that is, capable of computing new programs). To learn (that is, to compute a new program), it could *either* compute a *total* computable program (that is, one defined on all inputs) *or else* compute a *partial* computable program (that is, one that is undefined on some inputs).

Next, Kugel defines a *total machine* to be one that computes *only* total computable functions, and a *universal machine* to be one that computes *all* total computable functions. Is a universal Turing machine "total" or "universal" in this sense? According to Kugel, it can't be both: the set of total computable functions is enumerable: $f_1, f_2, \ldots$. Let P_i be a program that computes f_i, and let P be a program (machine?) that runs each P_i. Next, let P' compute $f_n + 1$. Then P' is a total computable function, but it is not among the P_i, hence it is not computed by P. That is, if P computes *only* total functions, then it can't compute *all* of them.

According to Kugel, a universal Turing machine is a "universal" machine (so it also computes *partial* functions). If the mind is a universal Turing machine, then there are functions whose values it can't compute for some inputs. And this, says Kugel would be detrimental to survival. If the mind were total, then there would be functions that it

couldn't compute *at all* (namely, partial ones). This would be equally detrimental.

But, says Kugel, there is a third option: Let the mind be a universal Turing machine with "pre-computed" or "default" values for those undefined inputs. Such a machine is not a Turing machine, because it could solve the Halting Problem! Give it a program *C* and input *i*, and simply let it immediately output 'loops' and only *then* start determining whether *C* halts on *i*. If *C* does halt on *i*, let this machine output 'halts'. That is, its "real" output is the *last* thing that it outputs, not the *first* thing. It is a trial-and-error machine.

(It is interesting to compare such machines to heuristic programming. In Chapter 7, we saw that a heuristic program was one that gave an approximate solution to a problem. It would seem that trial-and-error machines do just that: Each output is an approximation to the last output. Ideally, each output is a closer approximation than each previous output.)

Again, Putnam-Gold machines (NOTE: DECIDE WHETHER TO CALL THEM PGM OR T&EM) are such that, for each input, they allow an unlimited, but finite, number of outputs ("trials"). The last output (not any of the intermediate "guesses") is considered to be the computed value of the input. (If there is no first output, or no last output, then the result is technically undefined.) Both Putnam-Gold machines and Turing machines use the same "machinery", but they use it in different ways.

Is there an aspect of intelligence that is not Turing-machine–computable, but that *is* Putnam-Gold–computable? Yes: learning a function from its values. In Gold's version, you are given the initial outputs of a function *f*: $f(1), f(2),\ldots$, and you are asked if you can infer, or guess, or compute what function *f* is. This is called "computational learning theory" or "inductive inference". It is an abstract way of describing the problem that a child faces when learning its native language: $f(i)$ is what the child hears at time $t = i$, and *f* is the grammar of the language. Putnam-Gold machines are precisely defined to model this. ELABORATE ON GOLD'S THEORY

11.5   Summary

Martin Davis seems to have a good refutation of *some* hypercomputation theories, namely, of course you should expect to get non-computable output if you allow non-computable input.

Only two theories of hypercomputation seem useful: Interaction machines and trial-and-error machines.   But both of these seem to be only minor extensions of the Turing analysis of computation. Interaction machines give rise to relative computation, which still seems to be a version of Turing's original analysis. The only case in which an interaction machine differs significantly from a Turing machine is when the interactive machine doesn't halt *and* its input stream allows for an oracle-supplied, non-computable number. But this seems to fall under Davis's objection to hypercomputation. Notably (as we will see in Ch. 19), the Turing test is interactive; insofar as interaction is not modeled by a Turing machine, this would seem to be a trivial gap.

Trial-and-error machines, on the other hand, seem to have potential, but, importantly—despite being technically not the computation of a result—they abide by

all of Turing's constraints on what counts as computing, with the possible exception of knowing whether it has halted. So, perhaps intelligence is not a computable function, not because it is not computable, but because it is not a function![4]

---

[4]Some of the observations in this summary are based on a personal conversation with Jonathan Buss, 6 March 2007.

11.6   NOTES FOR NEXT DRAFT

1. Possibly organize this chapter in terms of "procedures" that relax different conditions on the notion of algorithm.

   For example, [Hopcroft and Ullman, 1969, 2] characterize a "procedure" as "a finite sequence of instructions that can be mechanically carried out, such as a computer program" and "algorithm" as a procedure that halts (p. 91). They also characterize "Church's hypothesis" as the claim "that any process which could naturally be called a procedure can be realized by a Turing machine" (p. 80). Since procedures in their sense need not halt, neither need Turing machines.

2. Need to say something about Hilbert's original constraints (finiteness, etc.), which would seem to require "computation" to be *humanly possible* computation.

3. Say something about quantum computing.

4. Need to elaborate on the notion of interactive computing and its relation to oracle machines.

5. Wegner is a proponent of "interactive computing". There is also something called "reactive systems", which may be the same thing, but neither body of literature seems to cite the other. Their relationships may be worth exploring. See:
   http://en.wikipedia.org/wiki/Reactive_system
   accessed 8 November 2012. Check with Selman on this.

   One of the pioneers of reactive systems, Amir Pnueli, characterizes them as "systems whose role is to maintain an ongoing interaction with their environment rather than produce some final value upon termination. Typical examples of reactive systems are air traffic control system[s], programs controlling mechanical devices such as a train, a plane, or ongoing processes such as a nuclear reactor" (from the syllabus for his Fall 2002 NYU course, "Analysis of Reactive Systems"). In the lecture notes for that course, he wrote: "There are two classes of programs: Computational Programs: Run in order to produce a final result on termination. Can be modeled as a black box. Specified in terms of input/output relations. Example: The program which computes $y = 1 + 3 + \ldots + (2x - 1)$ can be specified by the requirement $y = x^2$. Reactive Programs: Programs whose role is to maintain an ongoing interaction with their environments. ... Such programs must be specified and verified in terms of their behaviors." (from Lecture 1,
   http://www.cs.nyu.edu/courses/fall02/G22.3033-004/index.htm) also at:
   http://cs.nyu.edu/courses/fall02/G22.3033-004/ accessed 18 June 2014

6. Some games that involve tossing dice might be like oracle or interactive Turing machines. Still, by the Subst Thm, any such game/Turing machine, after it has

been played, can be simulated by a deterministic Turing machine. I.e., for each such oracle Turing machine, there is some deterministic Turing machine whose i-o and algorithmic behavior is identical, except it can't be determined in advance which one it is.

7. "There are things...bees can do that humans cannot and vice versa" [Sloman, 2002, §3.3]. Does that mean that bees can do non-computable tasks? Iff 'do' means something different from 'compute', such as physical performance. Cf. Cleland

8. See [Hayes, 1990], as discussed in §11, suggesting that interactive and classical Turing computation differ only in the source of input information.

9. On Turing on fallibility, related to §11.4.1:

> [O]ne can show that however the machine [that is, a computer] is constructed there are bound to be cases where the machine fails to give an answer [to a mathematical question], but a mathematician would be able to. On the other hand, the machine has certain advantages over the mathematician. Whatever it does can be relied upon, assuming no mechanical 'breakdown', whereas the mathematician makes a certain proportion of mistakes. I believe that this danger of the mathematician making mistakes is an unavoidable corollary of his [sic] power of sometimes hitting upon an entirely new method. [Turing, 1951, 256]

This gives support Kugel's claims about fallibility. Such trade-offs are common; for example, as Gödel showed, certain formal arithmetic systems can be consistent (infallible?) or else complete (truthful?), but not both. An analogy is this: In the early days of cable TV (the late 1970s), there were typically two sources of information about what shows were on—*TV Guide* magazine and the local newspaper. The former was "consistent"/"infallible" in the sense that everything that it said was on TV was, indeed, on TV; but it was incomplete, because it did not list any cable-TV shows. The local newspaper, on the other hand, was "complete" in the sense that it included all broadcast as well as all cable-TV shows, but it was "inconsistent"/"fallible", because it also erroneously included shows that were not on TV or cable (but there was no way of knowing which was which except by being disappointed when an advertised show was not actually on).

In a later passage, Turing suggests "one feature that...should be incorporated in the machines, and that is a 'random element' " (p. 259). This turns the computer into a kind of interactive *o*-machine that "would result in the behaviour of the machine not being by any means completely determined by the experiences to which it was subjected" (p. 259), suggesting that Turing realized that it would make it a kind of hypercomputer, but, presumably, one that would be only (small) extension of a Turing machine.

(It is also of interest to note that, in this talk, Turing envisaged what has come to be known as "The Singularity": "it seems probably tht once the machine thinking method had started, it would not take long to outstrip our feeble powers. There would be no question of the machines dying, and they would be able to converse with each other to sharpen their wits. At some stage therefore we should have to expect the machines to take control, in the way that is mentioned in Samuel Butler's *Erewhon*" (pp. 259–260).)

10. "Concurrent computation" can be thought of in two ways. In the first way, a single computer (say, a (universal) Turing machine) executes two different algorithms, not one after the other ("sequentially"), but in an interleaved fashion, in much the way that a parallel computation can be simulated on a serial computer: On a serial computer (again, think of a Turing machine), only one operation can be executed at a given time; on a parallel computer, two (or more) operations can be executed simultaneously. But one can simulate parallel computation on a serial computer by interleaving the the two operations. The difference between parallel and concurrent computation is that the latter can also be thought of in the second way mentioned above: Two (or more) computers (say, non-universal Turing machines) can each be executing two different, single algorithms simultaneously in such a way that outputs from one can be used as inputs to the other.[5]

Is concurrent computation a kind of hypercomputation? It is, if "interactive computation" is the same as that second way of thinking about concurrent computation, and if interactive computation is hypercomputation. At least one important theoretician of concurrent computing has said that "concurrent computation... is in a sense the *whole* of our subject—containing sequential computing as a well-behaved special area" [Milner, 1993, 78] This is consistent with those views of hypercomputation that take it to be a generalization of (classical) computation. Where (classical) computation is based on an analysis of mathematical functions, Milner argues that concurrent computation must be based on a different mathematical model; that also suggests that it is, in some sense, "more" than mere classical computation. Interestingly, however, he also says that this different mathematical model must "match the functional calculus not by copying its constructions, but by emulating two of its attributes: It is *synthetic*—we build systems in it, because the structure of terms represents the structure of processes; **and it is *computational*—its basic semantic notion is a step of computation**" [Milner, 1993, 82; italics in original, my boldface]. Again, it appears to be a generalization of classical computation.

Unlike certain models of hypercomputation that either try to do something that is physically (or "medically") impossible or can only exist in black holes,

---

[5]See http://en.wikipedia.org/wiki/Concurrent_computing
and http://en.wikipedia.org/wiki/Concurrency_(computer_science),
both accessed 26 May 2014.

concurrent computation seems more "down to earth"—more of a model of *humanly* possible processing, merely analyzing what happens when Turing machines interact.

In fact, here's an interesting way to think about it: A single Turing machine can be thought of as a model of a human computer solving a single task. But humans can work on more than one task at a time (concurrently, if not in parallel). And two or more humans can work on a single problem at the same time. Moreover, there is no reason why one human's insights ("outputs") from work on one problem couldn't be used (as "inputs") in that same human's work on a distinct problem. And there is surely no reason why one human's insights from work on a given problem couldn't be used by another human's work on that same problem. Surely even Hilbert would have accepted a proof of some mathematical proposition that was done jointly by two mathematicians. So, insofar as concurrent computation is a kind of hypersomputation, it seems to be a benign kind.

Is it the same as Wegner's "interactive computation"? Perhaps. One possible difference concerns the *source* of intermediate input. If such input comes from an oracle that can solve non-classically-computable problems, then it is surely different.

11. [Wegner, 1995, 45] identifies interaction machines with Turing machines with oracles and with "modeling reactive processes" (citing work by Pneuli). Moreover, "Church's thesis...becomes invalid if the intuitive notion of computability is extended to include interactive computation. Church and Turing identify 'effective computability' with purely automatic computation, while interaction machines model patterns of distributed interactive computations coordinated by protocols" [Wegner, 1995, 46]. Here, I think, we can begin to see where a disagreement with Wegner over the "invalidity" of the CTCT even for someone who might agree with him about the way in which interactive computing might be "hyper": I wouldn't want to say that CTCT identifies effective computability with purely automatic computation, because I would distinguish the latter notion from the intutive notion, taking automatic computation as a synonym for λ-computation or Turing-machine–computation and then understanding the CTCT as identifying the intuitive notion with the more formal, mathematical notion of automatic/λ-/Turing-machine–computation.

But here's an interesting question: If you don't allow oracles (or physically impossible computations, or black-hole computations, etc.), can interactive computation make the Halting Problem "computable"? Put another way, the Halting Problem is not classically computable; is it interactively computable?

12. "Automatic" computation is like "batch" computing:

> The classic models of computation are analogous to minds without bodies. For Turing's machine, a calculation begins with a problem on tis tape, and ends with an answer there. . . . How the initial tape. . . is input, and how the final one is output, are questions neither asked nor answered. These theories conform to the practice of batch computing.
>
> Eventually, interactive models of computation emerged, analogous to minds in bodies. . . . A single input at initiation and a single output at termination are now superseded by multiple inputs and outputs distributed in time and space. These theories conform to the practice of interactive computing.
>
> Interaction is the mind-body problem of computing. [Wadler, 1997, 240–241]

Wadler also notes that such interaction need not be concurrent. The rest of Wadler 1997 is concerned to show how such interaction can be added to a Turing-complete programming language. (It is not clear whether to what extent this makes the language more "powerful".)

More fully:

> Today, computing scientists face their own version of the mind-body problem: how can virtual software interact with the real world? In the beginning, we merely wanted computers to extend our minds: to calculate trajectories, to sum finances, and to recall addresses. But as time passed, we also wanted computers to extend our bodies: to guide missiles, to link telephones, and to proffer menus.
>
> The classic models of computation are analogous to minds without bodies. For Turing's machine, a calculation begins with a problem on its tape, and ends with an answer there. . . .
>
> Eventually, interactive models of computation emerged, analogous to minds in bodies. . . . A single input at initiation and a single output at termination are now superseded by multiple inputs and outputs distributed in time and space.

13. As [Hintikka and Mutanen, 1997, 181] note, the Halting Problem algorithm in its trial-and-error form "is not recursive, even though it is obviously mechanically determined in a perfectly natural sense." They also note that this "perfectly natural sense" is surely part of the informal notion of computation that the CTCT asserts is identical to recursiveness, hence that the CTCT "is not valid" (p. 180). (Actually, they're a bit more cautious, claiming that the informal notion is "ambiguous": "We doubt that our pretheoretical ideas of mechanical calculability are so sharp as to allow for only one explication" (p. 180).

14. Here's another term for interactive/reactive (etc.) computing:

> Turing machines accept no input while operating. A finite amount of data may be inscribed on the tape before the computation starts, but thereafter the machine runs in isolation from its environment. A coupled Turing machine is the result of coupling a Turing machine to its environment via one or more input channels. Each channel supplies a stream of symbols to the tape as the machine operates. [Copeland and Sylvan, 1999, 51]

As with Kugel, such a system clearly uses Turing "machinery". The authors also note that "any coupled Turing machine that halts can be simulated by the universal Turing machine, since the streams of symbols supplied by the coupled machine's input channels are, in this case, finite in length, and so can be inscribed on the universal machine's tape before it commences its task", essentially by the methods I suggested in §11.3 in our discussion of the Substitution Property. "However, the universal Turing machine is not always able to simulate a coupled Turing machine that—like an idealised automatic teller machine or air traffic controller—never halts" (p. 51). And they give a simple proof (p. 52) that there is a coupled Turing machine "that cannot be simulated by the universal Turing machine". Essentially, the proof involves an oracle that supplies an non-Turing computable real number, so this falls prey to Davis's objection. It raises the following question: Can a coupled Turing machine that does not make use of a non-computable oracle and that does not halt be simulated by a classical Turing machine?

15. Of course there are kinds of *hypo*computation: Consider the Chomsky hierarchy, FSA, PDA, etc. So, if there is any serious *hyper*computation, that puts classical, Turing-machine computation in the middle. It still holds a central place, given the equivalence of Turing machines to recursive functions, etc., and given its modeling of human computing and its relation to Hilbert's problem. Hypercomputation seems to come in two "flavors": what I'll call "weird" hypercomputation and what I'll call "plausible" hypercomputation (to use "neutral" terms!). In the former category, I'll put medically impossible Zeus machines, machines that can only exist near black holes, etc. In the latter category, I'll put tae-machines and interactive machines. In a grey zone in the middle, I'll put o-machines. o-machines are clearly a plausible extension of Turing machines, as even Turing knew. What about interactive machines that don't use non-computable input?

    Related to all this is the role of non-halting programs like operating systems, ATMs, etc. Some of my colleagues say that those programs don't express algorithms, because algorithms, by definition, must halt. But even Turing's original Turing machines didn't halt: They computed infinite decimals, after all! The central idea behind the Halting Problem is to find an algorithm that distinguishes between programs that halt and those that don't. Whether halting is a Good Thing is independent of that.

16. [Copeland and Sylvan, 2000, §8.1, esp. pp. 190–191] offer an interesting

categorization of hypercomputation. They first carve off analogue computing as a challenge to classical computation. This is not normally considered to be part of hypercomputation, so I will ignore it here (but see Chs. 3 and 9 for some references). Then there are the challenges from "newer physics"; these include Zeus machines, relativistic machines, etc. (which strike me as only of theoretical interest). But then they point out, more interestingly, that "computability is relative not simply to physics, but more generally to *systems* or *frameworks*". They begin with "degree theory" or "Turing reduction", based on Turing's *o*-machines. I NEED TO SAY MORE ABOUT THIS. As we've seen, this either falls prey to Davis's objection (non-computability in/non-computability out) or else simply becomes Wegner's interaction machines (Copeland's coupled machines), which are "built" out of "Turing machinery" and which may, or may not, exceed Turing machines in computing power (the only argument I've seen to that effect runs afoul of Davis's objection).

They then identify two kinds of relativity: "resource" relativity and "logical" relativity. The former includes "relativity in procedures, methods or in the devices available for computing", which includes the analogue and physical categories discussed above, as well as *o*-machines. Different "procedures" might include different basic operations or instructions (in much the same way that different geometric systems might go beyond straightedge and compass to include measuring devices (or might go "below" to replace a collapsible compass with a fixed compass—on this, see http://en.wikipedia.org/wiki/Compass_equivalence_theorem). But it's not clear if such procedural relativity necessarily goes beyond (if not below) Turing computability.

Logical relativity concerns the use of non-classical logics, such as relevance logics. [Copeland and Sylvan, 2000] suggest that these might lead to hypercomputation. Perhaps; but it is certainly the case that *classical* computers can compute using relevance logics [Shapiro and Rapaport, 1987].

17. [Davis, 2006b, 1218] notes that if a set A of natural numbers is Turing reducible to a set B of natural numbers (written: $A \leq_t B$), and "if B is itself a computable set, the nothing new happens; in such a case $A \leq_t B$ just means that A is computable. But if B is non-computable, then interesting things happen." According to [Davis, 2006c], of course, one of the *un*interesting things is that A will then turn out to be non-computable. (The *interesting* things have to do with "degrees" of *non*-computability: "can one non-computable set be more non-computable than another?".

18. In [Knuth, 2014, Question 13], Alfred Aho asks about "reactive distributed systems that maintain an ongoing interaction with their environment—systems like the Internet, cloud computing, or even the human brain. Is there a universal model of computation for these kinds of systems?"

Note that phrasing the question this way suggests that 'computation' is a very general term that includes not only Turing-machine computation but other kinds of computing as well (perhaps even hypercomputation).

Knuth's answer identifies "reactive processes" with "computational methods", defined in [Knuth, 1973, 4] as a non-terminating, single-processor algorithm (see §7.5, above).

This brief discussion suggests that the proper way to consider such reactive or distributed or interactive systems is *not* as some new, or "hyper", model of computation, but simply as the study of what might happen when *two or more* Turing machines interact. There's no reason to expect that they would not be more powerful in some sense than a single Turing machine. Clearly, if a Turing machine interacts with something that is not a Turing machine, then hypercomputation can be the result; this is what Davis notes. Even if it is two Turing machines that are interacting, and even if that produces the ability to "compute" non-Turing-machine-comptable functions, that would not indicate that there is any limitation to the CTCT (and this may be what Copeland had in mind when he distinguishes two interpretations of the CTCT).

19. On the Substitution Property:

> An interactive system is a system that interacts with the environment via its input/output behavior. The environment of the system is generally identified with the components which do not belong to it. Therefore, an interactive system can be referred to as an open system because it depends on external information to fulfil its task. *If the external resources are integrated into the system, the system no longer interacts with the environment and we get a new, closed system.* So, the difference between open and closed systems 'lies in the eye of the beholder'. [Prasse and Rittgen, 1998, 359, my italics]

"The components which do not belong to it" would presumably include external input. The italicized sentence is a nice statement of my interpretation of the Substitution Property.

[Prasse and Rittgen, 1998], by the way, admirably call attention to the vagueness and informality of the claims in [Wegner, 1997] (see especially §4.4 and §4.6).

20. On interaction machines:
[Prasse and Rittgen, 1998, 359] consider a program such as the following:

```
begin
    let b be a Boolean variable;
    let x be an integer variable;
    input(b);
    while b = true do
```

```
            input(x);
            output(x²);
            input(b);
      end
```

They say of a program such as this:

> Neglecting input/output, each iteration can be interpreted as a computation performed by a Turing machine. However, the influence of the (unknown) user input on the control flow makes it hard to determine what overall function is computed by this procedure (or if the procedure can be seen as a computation at all). ...
>
> The input will be determined only at run-time. The overall function is derived by integrating the user into the computation, thus closing the system.
>
> It is evident that Turing machines cannot model this behavior directly due to the missing input/output operations. Therefore, we need models that take into account inputs and outputs at run-time to describe computer systems adequately.

Wegner would agree, though Prasse & Rittgen's point is that Wegner's interpretation that this violates the CTCT is incorrect. Still, this is a nice example of the kind of hypercomputation that Wegner has in mind.

Now, we could easily write a Turing-machine program that would be a version of this while-loop. Consider such a Turing machine with a tape that is initialized with all of the input (a sequence of bs and xs, encoded appropriately). This Turing machine clearly is (or executes) an algorithm in the classical sense. Now consider a Turing machine with the same program (the same machine table), but with an initially blank tape and a user who inscribes appropriate bs and xs on the tape just before each step of the proram is executed (so the Turing machine is not "waiting" for user input, but the input is inscribed on the tape just in time). Is there a mathematical difference between these two Turing machines? Is there anything in [Turing, 1936] that rules this out? CHECK THIS

> Interaction machines are defined as Turing machines with input and output. Therefore, their internal behavior and expressiveness do not differ from that of equivalent Turing machines. Though Wegner leaves open the question of how the input/output mechanism works, it can be assumed that input and output involve only data transport, without any computational capabilities. Therefore, the interaction machine itself does not possess greater computational power than a Turing machine. However, through communication, the computational capabilities of other machines can be utilized. Interaction can then be interpreted as a (subroutine) call. [Prasse and Rittgen, 1998, 361]

Turing's *o*-machines are of this type; the call to a (possibly non-computable) oracle is simply a call to a (possibly non-computable) subroutine. So, as they say, "the machine *itself*" is just a Turing machine, and, as Davis would say, if a non-computable input is encoded on its tape, then a non-computable output can be encoded there, too.

They also note that "Wegner's concept of computability focuses more on real computers and systems than on the theoretical, function-oriented view of Church, Kleene, Markov, Post and Turing" (p. 362). This is certainly consistent with some of Smith's claims, to be discussed in §21.7.1.

21. Consistent with the organizing principle for the chapter of considering lifting restrictions on Turing machines, if we ignore temporal requirements, then we can get accelerating (or Zeus) machines: "Neither Turing nor Post, in their descriptions of the devices we now call Turing machines, made much mention of time.... They listed the primitive operations that their devices perform...but they made no mention of the *duration* of each primitive operation" [Copeland, 1998, 150][6]

We might choose to ignore or reject accelerating Turing machines on the grounds that they are "medically" (physically) impossible. After all, no (physical) device can really accelerate in that way. But then, by parity of reasoning, we would have to reject ordinary Turing machines, on the grounds that they, too, are physically impossible, because, after all no (physical) device can really have an infinite tape or even an arbitrarily extendible tape. So all physical Turing machines are limited to a tape that is no larger than the universe (large, but still finite) and hence simulable by a "hypo"computer (such as a PDA). So, if an abstract Turing machine is mathematically possible, then, surely, so is an accelerating Turing machine.

22. A tae-machine (such as the tae version of the Halting Problem program), if implemented on an accelerating Turing machine, will always let us know when it is finished "guessing":

> To produce a machine that can compute the values of the halting function...one simply equips an accelerated universal Turing machine wiht a signalling device—a hooter, say—such that the hooter blows when and only when the machine halts. ... [I]f the hooter blows within two moments of the start of the simulation then $H(p,d) = 1$, and if it remains quiet then $H(p,d) = 0$. [Copeland, 1998, 153]

(where $H(p,d)$ is the halting function such that $H(p,d) = 1$ if Turing machine p halts on input d and $H(p,d) = 0$ otherwise).

---

[6][Copeland, 1998, 151, fn 2] distinguishes between "accelerating" Turing machines and "Zeus" machines.

[Copeland, 1998, 154] observes that the hooter-enhanced machine is not a Turing machine.

23. [Shagrir, 1999, §2, pp. 132–133] observes that "The Church-Turing thesis is the claim that certain classes of computing machines, e.g., Turing-machines, compute effectively computable functions. It is not the claim that all computing machines compute solely effectively computable functions." This is consistent with Kugel's views on tae-machines; it is also consistent with the view that what makes interaction machines more powerful than Turing machines is merely how they are coupled with either other machines or with the environment, and not with the machinery itself. Shagrir's goal is to argue for a distinction between algorithms and computation, with the latter being a wider notion.

24. On interaction:

> We interact with the world through programs all the time. Very often, programs we use interact with a network of unseen programs so that their ends and ours may be realized. We are acted on by programs all the time. These entities are not just agents in the sense of being able to take actions; they are also agents in the representative sense, taking autonomous actions on our behalf. [Chopra, 2014]

Similarly, the nature of "ubiquitous" computing seems both to clearly call for such interaction as well as to consider it part of computing (not "hyper"computing); see, for example, [Dublon and Paradiso, 2014].

25. Some have argued that Turing's model of computation is based on what humans can do. Yet it is an idealized human who is modeled, for example, one that has no limits on space (recall that the tape is infinite). [Cleland, 2004, 212] points out that, in that case, one could allow other idealizations, such as no limits on speed of computation. That would make a Zeus machine at least as plausible as a Turing machine. (This fits in with the idea that hypercomputation, in general, investigates the lifting of restrictions on the classical informal notion of algorithm.) However, [Cockshott and Michaelson, 2007, §2.5, p. 235] reject these relaxations out of hand.

26. For a refutation of Wegner, see:
http://blog.computationalcomplexity.org/2006/07/principles-of-problem-solving-tcs.html

27. My point about a Turing machine being able to simulate an interactive/reactive/coupled computer (via the Substitution Property) is also mentioned in [Teuscher and Sipper, 2002].

28. [Fortnow, 2010] nicely refutes three of the major arguments in favor of hypercomputation (including analog computation). Of most interest to us is this passage, inspired by Turing's comment that "The real question at issue is 'What are the possible processes which can be carried out in computing a number?' " [Turing, 1936, §9, p. 249] (see §8.7.2.1, above):

> Computation is about process, about the transitions made from one state of the machine to another. Computation is not about the input and the output, point A and point B, but the journey. Turing uses the computable numbers as a way to analyze the power and limitations of computation but they do not reflect computation itself. You can feed a Turing machine an infinite digits of a real number..., have computers interact with each other..., or have a computer that perform an infinite series of tasks...but *in all these cases the process remains the same, each step following Turing's model.* ... So yes Virginia, the Earth is round, man has walked on the moon, Elvis is dead and *everything computable is computable by a Turing machine.* (pp. 3, 5, my italics.)

29. Good idea for intro section: [Cockshott and Michaelson, 2007, §2.5, p. 235] wonder whether hypercomputation is a Kuhnian revolutionary challenge to the "normal" science of computer science.

30. For more on oracles and relative computability, see [Dershowitz and Gurevich, 2008, §5, pp. 329f].

31.   Almost all the results in theoretical computability use relative reducibility and *o*-machines rather than *a*-machines and most computing processes in the real world are potentially online or interactive. Therefore, we argue that Turing *o*-machines, relative computability, and online computing are the most important concepts in the subject, more so that Turing *a*-machines and standard computable functions since they are special cases of the former and are presented first oly for pedagogical clarity to beginning students. [Soare, 2009, Abstract, p. 368][7]

This is an interesting passage, because it could be interpreted by hypercomputation advocates as supporting their position and by anti-hypercomputationalists as supporting theirs!

In fact, a later comment in the same paper suggests the pro-hypercomputational reading:

> The Turing *o*-machine is a better model to study them [that is, to study "online computing devices which can access or interact with some external database or other device"] because the Turing *a*-machine lacks this online capacity. [Soare, 2009, §9, p. 387]

He even cites, seemingly approvingly, passages from the introduction to a book on the topic edited by Wegner and Goldin [Goldin et al., 2006].

---

[7]A slightly different version of the same paper appears as [Soare, 2013].

He goes on to say, "The theory of relative computability developed by Turing [in his doctoral disseration] and [Emil] Post [for example, in [Post, 1943] and the *o*-machines provide a precise mathematical framework for database [or interactive] or online computing just as Turing *a*-machines provide one for offline computing processes such as batch processing" [Soare, 2009, §1.3, pp. 370–371]. This certainly suggests that some of the things that Copeland and Wegner say about hypercomputation are a bit hyperbolic; it suggests that both the kind of hypercomputation that takes non-computable input (supplied by an oracle) to produce non-computable output as well as the kind that is interactive are both well-studied and simple extensions of classical computation theory. Oracles can model both client-server interaction as well as communication with the Web [Soare, 2009, §1.3, p. 371].

He also discusses Putnam's tae-machines, which he calls "limit computable functions" [Soare, 2009, §9.2, p. 388].

However, the interesting point is that all of these are *extensions* of Turing machines, not entirely new notions. Moreover, Soare does not disparage, object to, or try to "refute" the CTCT; rather, he celebrates it [Soare, 2009, §12].

His basic point on this topic seems to be this:

> **Conclusion 14.3** The subject is primarily about incomputable objects not computable ones, and has been since the 1930's. The single most important concept is that of relative computability to relate incomputable objects. [Soare, 2009, §14, p. 395]

This is certainly in the spirit of hypercomputation without denigrating the CTCT.

32. On Gold and tae machines, cite [Hauser et al., 2002, 1577].

33. [Milner, 1993, 80] offers an example of how "a theory of concurrency and interaction requires a new conceptual framework"—though that is not necessarily to say that it is a form of "hypercomputation" or that it refutes the CTCT:

> [T]here are programs P_1 and P_2 which have the same relational meaning,[8] but which behave differently when each runs in parallel with a third program Q...:
>
> Program P_1: x := 1 ; x := x + 1
> Program P_2: x := 2
>
> In the absence of interference,[9] P_1 and P_2 both transformt he initial memory by replacing the value of x by 2, so they have the same meaning. But if you take the program

---

[8]In the case of the example below, this simply means that the two programs have the same input-output behavior.

[9]This can be thought of as interaction with the environment in the form of intermediate input.

Program Q: x := 3

and run it in parallel with P_1 and P_2 in turn:

Program R_1: P_1 par Q Program R_2: P_2 par Q

then the programs R_1 and R_2 have different meaning. (Even if an assignment statement is executed indivisibly, R_1 can end up with x equal to 2, 3, or 4, while R_2 can only end up with x equal to 2 or 3.) So a compositional semantics must be more refined; it has to take account of the way tghat a program *interacts* with the memory.

To see why R_1 can have any of those three possible values, consider that Q could be thought of as being executed before "x := 1", immediately after it, or after "x := x + 1". To make the relevance to Wegner's claims clearer, note that Q could have been "input(x)", instead of an assignment statement. But note Milner's conclusion: It is not that such interactive programming refutes the CTCT, nor that this is a form of computation that goes beyond classical Turing-machine computation; rather, it is merely the claim that a "more refined" form of semantics is needed. (Wegner & Goldin 2003 chastise Milner for "avoid[ing] the question".)

34. Nice quote on interactive computing:

    I can design a program that never crashes if I don't give the user any options. And if I allow the user to choose from only a small number of options, limited to htings that appear on a menu, I can be sure that nothing anomalous will happen, becasue each option can be foreseen in advance and its effects can be checked. But if I give the user the ability to write *programs* that will combine with my own program, all hell might break loose. [Knuth, 2001, 189–190]

35. A quote sympathetic to Wegner:

    Turing reducibility ["developed by Post" from "Turing's oracle machine"]... is the most important concept in computability theory. Today, the notion of a locla machine interacting with a remote database or remote machine is central to practical computing" [Soare, 2012, 3290]

36. One of the "constraints" on Turing's analysis of computation, as cited by [Gurevich, 2012, 4], is that of being "**Isolated** Computation is self-contained. No oracle is consulted, and nobody interferes with the computation either during a computation step or in between steps. The whole computation of the algorithm is determined by the initial state." This certainly suggests that interactive computation is not Turing-machine computation. On the other hand, it could also be interpreted to mean merely that computation must be "mechanical" or "automatic", and surely[10] this could include the "mechanical" or "automatic"

---

[10]A word that any philosopher should surely(?) take with a grain of salt!

use of input from an external source (including an oracle).

Cf. also [Gurevich, 1999, 93, 98], which talks about "algorithms that are closed in the sense that they do not interact with their environments" and ones that do so interact (§5, pp. 111–115). Gurevich also notes that, in the case of non-deterministic algorithms, "the active environment will make the choices" (p. 116).

[Ekdahl, 1999, §3, pp. 262–263] has a nice example that illustrates how interactive computing is modeled by *o*-machines and relative computability. The essence of the example considers a simplified version of an airline-reservation program. Such a program is a standard example of the kind of interactive program that Wegner claims is not Turing-machine computable, yet it is not obviously an *o*-machine, because it does not obviously ask an oracle for the solution to a non-computable problem. Suppose our simplified reservation program is this:

```
begin
  while true do:
     input(passenger, destination);
     output(ticket(passenger, destination))
  end while
end.
```

Ekdahl observes that, although writing the passenger and destination information on the input tape is computable "and can equally well be done by another Turing machine", when our reservation program then " 'asks' for two new" inputs, "**which** [inputs are] going to [be written] on the tape *is not a recursive process. . . .* So, the input of [passenger and destination] can be regarded as a question to an *oracle*. An oracle answers questions known in adavance but the answers are not possible to reckon in advance" (italics in original; my boldface).

37. On trial and error machines:

> someone who wishes to know the correct answer would have no way to know at any given time whether the latest output is the correct output (Quoted by [Nayebi, 2014, 286] from:
> Davis, Martin (2006), "The Church-Turing Thesis: Consensus and Opposition", *Lecture Notes in Computer Science 3988*: 125–132.)

That is, a trial and error machine does "compute" the uncomputable, but we can't reliably use the result. Really? What about language learning and heuristic satisficing?

38. On trial and error machines:
William James's discussion of two different approaches to knowledge seems relevant:

> But the faith that truth exists, and that our minds can find it, may
> be held in two ways. We may talk of the *empiricist* way and of the
> *absolutist* way of believing in truth. The absolutists in this matter say
> that we not only can attain to knowing truth, but we can *know when* we
> have attained to knowing it; whilst the empiricists think that although
> we may attain it, we cannot infallibly know when. To *know* is one
> thing, and to know for certain *that* we know is another. [James, 1897,
> "The Will to Believe", §V, p. 465]

Compare: The faith that a problem has a computable (or algorithmic) solution
exists, and that our computers can find it, may be held in two ways. We may
talk of the trial-and-error way and of the Turing-algorithmic way of solving
a problem. The Turing algorithmists in this matter say that we (or Turing
machines) not only can solve computable problems, but we can know when
we (or they) have solved them; while the trial-and-error hypercomputationalists
think that although we (or our computers) may solve them, we cannot infallibly
know when. For a computer to produce a solution is one thing, and for us to
know for certain that it has done so is another.

39. [Feferman, 1992, 315] claims that "notions of relative (rather than absolute)
    computability" (that is, notions based on Turing's *o*-machines rather than on
    his *a*-machines) have "primary significance for practice" and that these relative
    notions are to be understood as "generalization[s]...of computability [and "of
    the Church-Turing Thesis"] to arbitrary structures". So this seems to fly in the
    face of Wegner's claims that interaction is something new while agreeing with
    the substance of his claims that interaction is more central to modern computing
    than Turing machines are.

40. [Feferman, 1992, 321] notes that *o*-machines (where *o* is an oracle that solves a
    non-computable problem) can be "generalized to that of a *B*-machine for any set
    B":

    > In the register machine model, one simply adds to the basic
    > instructions, ones of the form:
    >
    > **(1)** $r_j := 1$ if r_k is in *B*, else $r_j := 0$.

    Essentially, this adds primitive operations to a Turing machine (or a register
    machine). If these operations can be simulated by the standard primitive
    operations of the Turing machine, then we haven't increased its power (only
    its expressivity, essentially by the use of named subroutines). But if *B* contains
    problems not solvable by the Turing machine, then, of course, we have increased
    its power.

    On named subroutines: [Feferman, 1992, 339–340] says that the "built-in
    functions" of "actual computers" are "given by a 'black box'—which is just
    another name for an 'oracle'—and a program to compute a function *f* from one
    or more of these" built-in functions "is really an algorithm for computation of *f*

relative to" those built-in functions.

More importantly, Feferman goes on to observe that "the arguments for the Church-Turing Thesis lead one strongly to accept a relativized version: (C-T)^r [a set] *A* is effectively computable from *B* if (and only if)" *A* is "Turing computable from" (or "Turing reducible to") *B*. To say that *A* is Turing computable from (or Turing reducible to) *B* is to say that *x* is in *A* iff the *B*-machine outputs 1 when its input is *x* (where output 1 means "yes, *x* is in *A*"). This is called Feferman then says that "Turing reducibility gives the most general concept of relative effective computability".

And here is Feferman on the crucial matter:

> Uniform global recursion provides a much more realistic picture of computing over finite data structures than the absolute computability picture, for finite data bases are constantly being updated. As examples, we may consider... airline reservation systems. [Feferman, 1992, 342]

He does, however, go on to say that "while notions of relativized (as compared to absolute) computability theory are essentially involved in actual hardware and software design, the bulk of methods and results of recursion theory have so far proved to be irrlevant to practice" [Feferman, 1992, 343]. That certainly is congenial to Wegner's complaints.

41. Perhaps the issue is not so much whether it is *possible* to compute the uncomputable (by extending or weakening the notion of Turing-machine computation), but whether it is *practical* to do so. [Davis, 2006a, 126] finds this to be ironic:

> ... computer scientists have had to struggle wih the all-too-evident fact that from a practical point of view, Turing computability does not suffice. ... With these [that is, with NP-complete] problems Turing computability doesn't help because, in each case, the number of steps required by the best algorithms available grows exponentially with the length of the input, making their use in practive problematical. How strange that despite this clear evidence that computability alone does not suffice for practical purposes, a movement has develoope under the banner of "hypercomputation" proposing the practicality of computing the non-computable.

42. Trial and error computation is equivalent to computations by *o*-machines that solve the halting problem:

> If the computation is to determine whether or not a natural number *n* as input belongs to some set *S*, then it turns out that sets for which such "trial and error" computation is available are exactly those...

that are computable relative to. . . a oracle that provides correct answers to queries concerning whether a given Turing machine. . . will eventually halt. [Davis, 2006a, 128]

43. O-machines show us that not all that is studied in computation theory is Turing-equivalent. [Aizawa, 2010, 230]

Note the subtle difference between saying this and saying something like: All computation is equivalent to Turing-machine computation (which is a version of the CTCT).

44. [Weizenbaum, 1976, Ch. 5, p. 135] interestingly distinguishes between "computers" and "robots", where the latter (but not the former) "have perceptors. . . and effectors". This could be interpreted as a way of distinguishing between Turing-equivalent computers and interactive computers, the latter being Weizenbaum's "robots". On the other hand, even Turing machines have to have perceptors and effectors in the sense of having a read-write head.

45. On oracle computation:
One technique for simulating solutions to the N-body problem is to use an ordinary computer linked to a graphics processing unit that is far more powerful than the ordinary computer. Is such a combination like a Turing machine with an oracle? See [Hayes, 2015, esp. pp. 92–93]

11.7   Further Sources of Information

MAYBE REORGANIZE THIS TO INCLUDE SEPARATE SECTIONS OF "PRO" AND "CON" HYPERCOMPUTATION PAPERS.

1. Books and Essays:

   - Aaronson, Scott (2008), "The Limits of Quantum Computers", *Scientific American* (March): 62–69.

     – Contains some comments casting doubt on certain models of hypercomputation.

   - Aaronson, Scott (2013), *Quantum Computing Since Democritus* (New York: Cambridge University Press).

     – Contains discussions of both quantum computing and hypercomputation; see Aaronson 2006, below.

   - Aaronson, Scott (2014), "Quantum Randomness", *American Scientist* 102(4) (July-August): 266–271.

     (a) Accessed 8 September from:
         http://www.americanscientist.org/issues/pub/quantum-randomness

     (b) "If there's no predeterminism in quantum mechanics, can it output numbers that truly have no pattern?"

   - Arkoudas, Konstantine (2008), "Computation, Hypercomputation, and Physical Science", *Journal of Applied Logic* 6(4) (December): 461–475.

     – Preprint accessed 30 June 2014 from:
       http://people.csail.mit.edu/kostas/papers/jal.pdf

     – Argues against Copeland that a version of the CTCT that is immune to hypercomputation objections follows from the "systematic predictability of observable behavior" (and that even *o*-machines are, in fact, "deterministic digital computers") (see §3 of his paper).

   - Brassard, Gilles (1995), "Time for Another Paradigm Shift", *ACM Computing Surveys* 27(1) (March): 19–21.

     – A commentary on [Hartmanis, 1995a], arguing that probabilistic and—especially—quantum computers "may be qualitatively more powerful than classical machines"; see [Hartmanis, 1995b] for a reply.

   - Bringsjord, Selmer; & Arkoudas, Konstantine (2004), "The Modal Argument for Hypercomputing Minds", *Theoretical Computer Science* 317: 167–190.

     – Accessed 25 June 2014 from:
       http://research.cs.queensu.ca/home/akl/cisc879/papers/PAPERS_FROM_THEORETICAL_COMPUTER_SC

     – "[W]e give herein a novel, formal modal argument showing that since it's mathematically *possible* that human minds are hypercomputers, such minds *are* in fact hypercomputers." (from the abstract)

   - Bringsjord, Selmer; & Zenzen, Michael (2002), "Toward a Formal Philosophy of Hypercomputation", *Minds and Machines* 12(2) (May): 259–280.

   - Button, Tim (2009), "SAD Computers and Two Versions of the Church-Turing Thesis", *British Journal for the Philosophy of Science* 60: 765–792.

- **Chaitin, Gregory (2006), "How Real Are Real Numbers?",** *International Journal of Bifurcation and Chaos*

  - 2006 version accessed 6 November 2012 from:
    http://www.cs.auckland.ac.nz/∼chaitin/olympia.pdf
  - 2009 version accessed 6 November 2012 from:
    http://www.umcs.maine.edu/∼chaitin/wlu.html
  - "We discuss mathematical and physical arguments against continuity and in favor of discreteness" (from the abstract). "In addition to this mathematical soul-searching regarding [the reality of] real nuymbers, some physicists are beginning to suspect that the physical universe is. . . perhaps even a giant computer [. . . Wolfram, 2002]" (from §1).

- Chaitin, Gregory (2006), "The Limits of Reason", *Scientific American* (March): 74–81.

  - Accessed 6 November 2012 from:
    http://www.umcs.maine.edu/∼chaitin/sciamer3.pdf

- Cockshott, Paul; & Michaelson, Greg (2007), "Are There New Models of Computation? Reply to Wegner and Eberbach", *The Computer Journal* 50(2): 232–247.

  - Preprint accessed 30 June 2014 from:
    http://www.dcs.gla.ac.uk/∼wpc/reports/wegner25aug.pdf
  - Argues that Wegner's interaction machines do not exceed Turing machines in computational power; also argues against a number of other hypercomputation proposals, and contains an excellent summary of Turing machines and complexity theory.

- Cooper, S. Barry (2012), "Incomputability after Alan Turing", *Notices of the AMS* 59(6): 776–784.

  - Accessed 21 July from:
    http://www.ams.org/notices/201206/rtx120600776p.pdf
  - An interesting, illustrated, historical survey.

- Copeland, B. Jack (2002), "Accelerating Turing Machines", *Minds and Machines* 12(2) (May): 303–326.

- Copeland, B. Jack (guest ed.) (2002), Special Issue on Hypercomputation, *Minds and Machines* 12(4) (November).

  - Includes Bringsjord & Zenzen 2002; Copeland 2002, "Hypercomputation'; Kugel 2002; Steinhart 2002.
  - Complete table of contents accessed 5 February 2014 from:
    http://link.springer.com/journal/11023/12/4/

- Copeland, B. Jack (guest ed.) (2003), Special Issue on Hypercomputation (continued), *Minds and Machines* 13(1) (February).

  - Complete table of contents accessed 5 February 2014 from:
    http://link.springer.com/journal/11023/13/1/

- Copeland, B. Jack; & Proudfoot, Diane (1999), "Alan Turing's Forgotten Ideas in Computer Science", *Scientific American* (April): 98–103.

  – Accessed 18 July 2014 from:
    http://www.cs.virginia.edu/~robins/Alan_Turing's_Forgotten_Ideas.pdf

  – For a rebuttal, see:
    Hodges, Andrew, "The Professors and the Brainstorms"
    accessed   8   November   2012   from
    http://www.turing.org.uk/philosophy/sciam.html

- Copeland, B. Jack; & Shagrir, Oron (2011), "Do Accelerating Turing Machines
  Compute the Uncomputable?", *Minds and Machines* 21(2): 221–239.

- Copeland, B. Jack; & Shagrir, Oron (2013), "Turing versus Gödel on Computability
  and the Mind", in B. Jack Copeland, Carl J. Posy, & Oron Shagrir (eds.),
  Computability: Turing, Gödel, Church, and Beyond (Cambridge, MA: MIT Press):
  1–33, esp. §1.2.3 (pp. 12–15).

- Cotogno, P. (2003), "Hypercomputation and the Physical Church-Turing Thesis",
  British Journal for the Philosophy of Science 54(2): 181–224.

  – Argues that hypercomputation does not refute the CTCT.

  – For a reply, see:
    Welch, P.D. (2004), "On the Possibility, or Otherwise, of Hypercomputation",
    British Journal for the Philosophy of Science 55: 739–746.

- Davies, E.B. (2001), "Building Infinite Machines", *British Journal for the
  Philosophy of Science* 52: 671–682.

  (a) Preprint accessed 4 June 2014 from:
      http://www.mth.kcl.ac.uk/staff/eb_davies/jphilsci.pdf

  (b) Argues that an accelerating computer could be built "in a continuous
      Newtonian universe" (as opposed to the Hogarth-Malament spacetime
      mentioned in the motto to this chapter), though not "in the real universe".

- **Davis,   Martin   (2006),   "Why   There   Is   No   Such   Discipline   as
  Hypercomputation",** *Applied Mathematics and Computation* **178: 4–7**

  – Accessed 8 November 2012 from:
    http://tinyurl.com/Davis2006[11]

- Dresner, Eli (2008), "Turing-, Human- and Physical Computability: An Unasked
  Question", *Minds and Machines* 18(3) (Fall): 349–355.

- Frenkel, Karen (1993), "An Interview with Robin Milner", *Communications of the
  ACM* 36(1) (January): 90–97.

  – Accessed 26 May 2014 from:
    http://delivery.acm.org/10.1145/160000/151241/a1991-frenkel.pdf

  – A companion piece to [Milner, 1993], with interesting observations on AI, the
    semantics of programming languages, program verification, and the nature of
    computer science.

- Gold, E. Mark (1965), "Limiting Recursion", *Journal of Symbolic Logic* 30(1):
  28–48.

---

[11]http://research.cs.queensu.ca/home/akl/cisc879/papers/PAPERS_FROM_APPLIED_MATHEMATICS_AND_COMPUTATION/
Special_Issue_on_Hypercomputation/davis%5b1%5d.pdf

- – Presents the mathematics behind tae machines: "A class of problems is called decidable if there is an algorithm which will give the answer to any problem of the class after a *finite* length of time. The purpose of this paper is to discuss the classes of problems that can be solved by *infinitely* long decision procedures in the following sense: An algorithm is given which, for any problem of the class, generates an infinitely long sequence of guesses. The problem will be said to be *solved in the limit* if, after some finite point in the sequence, all the guesses are correct and the same..." (from the abstract, my italics).

- **Gold, E. Mark (1967), "Language Identification in the Limit",** *Information and Control* **10: 447–474.**

  - – Accessed 27 June 2014 from:
    http://web.mit.edu/~6.863/www/spring2009/readings/gold67limit.pdf
  - – For an excellent guide to understanding Gold's paper, see:
    Johnson, Kent (2004), "Gold's Theorem and Cognitive Science", *Philosophy of Science* 71 (October): 571–592.

    - ∗ Accessed 27 June 2014 from:
      http://www.afhalifax.ca/magazine/wp-content/sciences/for-emgold/Johnson.GoldsTheorem.pdf

- Goldin, Dina Q.; Smolka, Scott A.; Attie, Paul C.; & Sonderegger, Elaine L. (2004), "Turing Machines, Transition Systems, and Interaction", *Information and Computation* 194(2) (November), Special Issue Commemorating the 5oth Birthday Anniversary of Paris C. Kanellakis, pp. 101–128.

  - – Accessed 25 June 2014 from:
    http://www.cse.uconn.edu/~dqg/papers/its.pdf
  - – "We present *Persistent Turing Machines* (PTMs), a new way of interpreting Turing-machine computation, one that is both interactive and persistent. A PTM is a nondeterministic 3-tape Turing machine that upon receiving an input token, computes for a while and then outputs the result, and this process is repeated forever. It is persistent in the sense that the work-tape contents are maintained from one computation...to the next." (from the abstract)

- Goldin, Dina; & Wegner, Peter (2004), "The Origins of the Turing Thesis Myth", accessed 27 June 2014 from:
  https://web.archive.org/web/20120212032143/http://www.engr.uconn.edu/~dqg/papers/myth.pdf

  - – A sequel to [Wegner, 1997] and Wegner & Goldin 2003.

- Goldin, Dina; & Wegner, Peter (2008), "The Interactive Nature of Computing: Refuting the Strong Church-Turing Thesis", *Minds and Machines* 18(1) (Spring): 17–38.

- Hintikka, Jaakko; & Mutanen, Arto (1997), "An Alternative Concept of Computability", in Jaakko Hintikka (ed.), *Language, Truth, and Logic in Mathematics* (Dordrecht, The Netherlands: Springer): 174–188.

- Hodges, Andrew (2012), "Beyond Turing's Machines", *Science* 336 (13 April): 163–164.

  - – Accessed 21 July 2014 from:
    http://211.144.68.84:9998/91keshi/Public/File/41/336-6078/pdf/163.full.pdf

- Discusses "whether all types of computation—including that of our own minds—can be modeled as computer programs", in the context of quantum computation.

- Johnson, Kent (2004), "Gold's Theorem and Cognitive Science", *Philosophy of Science* 71 (October): 571–592.

- Kugel, Peter (1986), "Thinking May Be More than Computing", *Cognition* 22(2) (March): 137–198.

- Kugel, Peter (1986), "When Is a Computer Not a Computer?", *Cognition* 23(1) (June): 89–94.

- Kugel, Peter (2004), "Toward a Theory of Intelligence", *Theoretical Computer Science* 317: 13–30.

- Kugel, Peter (2005), "It's Time to Think outside the Computational Box", *Communications of the ACM* 48(11) (November): 33–37.

  - For a response, see:
    Wegner, Peter; & Goldin, Dina (2006), Forum, *Communications of teh ACM* 49(3): 11.

- Lloyd, Seth (2000), "Ultimate Physical Limits to Computation", *Nature* 406 (31 August): 1047–1054.

  (a) Pre-prints accessed 4 June 2014 from:
      http://arxiv.org/pdf/quant-ph/9908043.pdf?origin=publicationDetail
      and
      http://cds.cern.ch/record/396654/files/9908043.pdf

  (b) Investigates "the fundamental physical limits of computation as determined by the speed of light..., the quantum scale... and the gravitational constant".

- **Milner, Robin (1993), "Elements of Interaction",** ***Communications of the ACM 36(1) (January): 78–89.***

- Parker, Matthew W. (2009), "Computing the Uncomputable; or, The Discrete Charm of Second-Order Simulacra", *Synthese* 169: 447–463.

  - Preprint accessed 30 June 2014 from:
    http://philsci-archive.pitt.edu/3905/1/Discrete_Charm_i.pdf

  - Argues that "non-computable behavior in a model...[can be] revealed by computer simulation" (which is, of course, computable).

- Piccinini, Gualtiero (2003), "Alan Turing and the Mathematical Objection", *Minds and Machines* 13: 23–48,
  http://www.umsl.edu/~piccininig/Alan_Turing_and_Mathematical_Objection.pdf
  (accessed 13 February 2015).

  - Primarily about Turing's views on AI (see Ch. 19), but also discusses his theory of computation and the role of "oracle" machines.

- Piccinini, Gualtiero (2011), "The Physical Church-Turing Thesis: Modest or Bold?", *British Journal for the Philosophy of Science* 62: 733–769.

  - Preprint accessed 21 July 2014 from:
    http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=CEE1AFD492550C99D72CE9A7D90D4E37?doi

  - §4 discusses hypercomputational challenges to the "modest" CTCT, which states that "any function that is physically computable is Turing computable" (p. 734).)

- Schächter, Vincent (1999), "How Does Concurrency Extend the Paradigm of Computation?", *The Monist* 82(1) (January): 37–57.

- Sloman, Aaron (1996), "Beyond Turing Equivalence", in P.J.R. Millican & A. Clark (eds.), *Machines and Thought: The Legacy of Alan Turing*, Vol. I, Ch. 10, pp. 179–219.

  – Preprint accessed 21 July 2014 from:
    http://www.cs.bham.ac.uk/research/projects/cogaff/Sloman.turing90.pdf
  – Clarifies how hypercomputation can show how some aspects of *human* cognition might not be Turing-machine computable. Nevertheless, the question remains whether cognition *in general* is Turing-machine computable (or can be approximated by a Turing machine).

- Steinhart, Eric (2002), "Logically Possible Machines", *Minds and Machines* 12(2) (May): 281–301.

- van Leeuwen, Jan; & Wiedermann, Jiří (2000), "The Turing Machine Paradigm in Contemporary Computing", in Bjorn Engquist & Wilfried Schmid (eds.), *Mathematics Unlimited—2001 and Beyond* (Berlin: Springer-Verlag): 1139–1155.

  (a) Preprint accessed 4 June 2014 from:
    http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.32.3925&rep=rep1&type=pdf
  (b) Takes up the challenge of formalizing Wegner's ideas about interaction machines; argues that " 'interactive Turing machines with advice'…are more powerful than ordinary Turing machines."

- Wegner, Peter (1999), "Towards Empirical Computer Science", *The Monist* 82(1) (January): 58–108.

- Wegner, Peter; & Goldin, Dina (1999), "Interaction, Computability, and Church's Thesis", preprint accessed 4 June 2014 from:
  http://cs.brown.edu/~pw/papers/bcj1.pdf

  – An (unpublished) attempt at formalizing some of Wegner's claims about interaction machines.

- Wegner, Peter; & Goldin, Dina (2003), "Computation beyond Turing Machines", *Communications of the ACM* 46(4) (April): 100–102.

  – Accessed 25 June 2014 from:
    http://www.pld.ttu.ee/~vadim/AIRT/8_computation_beyond_turing_machines.pdf

- Wegner, Peter; & Goldin, Dina (2006), "Principles of Problem Solving", *Communications of the ACM* 49(7): 27–29.

  – For a response, arguing that Wegner and Goldin are wrong, see:
    Fortnow, Lance (2006), "Principles of Problem Solving: A TCS Response", ¡i¿Computational Complexity¡/i¿ (blog), accessed 27 June 2014 from:
    http://blog.computationalcomplexity.org/2006/07/principles-of-problem-solving-tcs.html

- Wells, Benjamin (2004), "Hypercomputation by Definition", *Theoretical Computer Science* 317: 191–207.

  – Accessed 18 July 2014 from:
    http://research.cs.queensu.ca/home/akl/cisc879/papers/PAPERS_FROM_THEORETICAL_COMPUTER_SCIENCE/050510111233316076.
  – §2 contains a discussion of [Cleland, 1995, Cleland, 2001].

- – §3 contains a discussion of Tarski's views on the relation between decidability and recursiveness, and on the relation between the CTCT and the P=NP problem.

- **On Quantum Computers:**

  - – Aaronson, Scott (2008), "The Limits of Quantum Computers", *Scientific American* (March): 62–69.

    - ∗ "Quantum computers would be exceptionally fast at a few specific tasks, but it appears that for most problems they would outclass today's computers only modestly. This realization may lead to a new fundamental physical principle."

  - – Bacon, Dave; & van Dam, Wim (2010), "Recent Progress in Quantum Algorithms", *Communications of the ACM* 53(2) (February): 84–93.

  - – Deutsch, David (1985), "Quantum Theory, the Church-Turing Principle and the Universal Computer", *Proceedings of the Royal Society of London A* 400: 97–117

    - ∗ Preprint accessed 8 April 2014 from:
      http://www.cs.berkeley.edu/∼christos/classics/Deutsch_quantum_theory.pdf

  - – Gover, Lov K. (1999), "Quantum Computing", *The Sciences* (July/August): 24–30.

    - ∗ Accessed 9 December 2013 from:
      http://cryptome.org/qc-grover.htm

  - – Hayes, Brian (2014), "Programming Your Quantum Computer", *American Scientist* 102(1) (January-February): 22–25.

    - ∗ Accessed 19 January 2014 from:
      https://www.americanscientist.org/issues/pub/programming-your-quantum-computer/
      "The quantum system serves as an 'oracle,' answering questions that can be posed in a format suitable for qubit computations" (p. 24).

  - – Monroe, Christopher R.; & Wineland, David J. (2008), "Quantum Computing with Ions", *Scientific American* (August): 64–71.

    - ∗ Accessed 29 March 2014 from:
      http://www.cs.virginia.edu/∼robins/Quantum_Computing_with_Ions.pdf

2. Websites:

- Aaronson, Scott (2006), "PHYS771 Lecture 4: Minds and Machines", http://www.scottaaronson.com/democritus/lec4.html (accessed 28 June 2014).

  - – Part of a course on Quantum Computing Since Democritus. The first part discusses oracles and Turing reducibility in a very clear (but elementary) way, concluding that hypercomputation is not a serious objection to the CTCT; later parts discuss AI.

  - – Some of the material also appears in Aaronson 2013.

- Aaronson, Scott (2012), "The Toaster-Enhanced Turing Machine", ¡¿Shtetl-Optimized¡/¿,
  http://www.scottaaronson.com/blog/?p=1121

- Argues against hypercomputation via a parallel argument that, because Turing machines can't toast bread, a toaster-enhanced Turing machine that "allows bread as a possible input and includes toasting it as a primitive operation" would be more powerful than a classic Turing machine.

- American Mathematical Society (2003), AMS Sectional Meeting Full Program

  - Accessed 18 July 2014 from:
    http://www.ams.org/meetings/sectional/2096_progfull.html
  - Contains the titles and authors of many papers on hypercomputation that were presented in four separate sessions on "Beyond Classical Boundaries of Computability".
  - Published versions of many of the papers (including papers by Bringsjord & Arkoudas, Cleland, Copeland, Kugel, and Shagrir) are online at:
    http://research.cs.queensu.ca/home/akl/cisc879/papers/PAPERS_FROM_THEORETICAL_COMPUTER_SCIENCE/

- Burgin, Mark, & Wegner, Peter (organizers) (2003), Special Sessions on Beyond Classical Boundaries of Computability (Parts I, II, III, & IV), 2003 Spring Western Section Meeting, American Mathematical Society.

  - Accessed 8 November 2012 from:
    http://www.ams.org/meetings/sectional/2096_progfull.html

- E-mail discussion between Carl Hewitt (a major figure in concurrent computing) and Peter Wegner (the major advocate of interactive computing as a counterexample to the CTCT), accessed 26 May 2014 from:
  http://www.cse.buffalo.edu/~rapaport/510/actors-vs-church-thesis.txt

- "Interactive Computation Is More Powerful Than Non Interactive", accessed 26 May 2014 from:
  http://c2.com/cgi/wiki?InteractiveComputationIsMorePowerfulThanNonInteractive

  - An debate between a hypercomputation skeptic and a believer.

- **On Hypocomputation:** If "hyper"computation is computing that is "above" the Turing "speed limit", so to speak, then "hypo"computation is computing "below" it:

  - Lindell, Steven (2004), "Revisiting Finite-Visit Computations", accessed 5 February 2014 from: http://tinyurl.com/Lindell2004[12]
  - Lindell, Steven (2006), "A Physical Analysis of Mechanical Computability", accessed 5 February 2014 from: http://tinyurl.com/Lindell2006[13]

---

[12]http://www.haverford.edu/cmsc/slindell/Presentations/Revisiting%20finite-visit%20computations.pdf

[13]http://www.haverford.edu/cmsc/slindell/Presentations/A%20physical%20analysis%20of%20mechanical%20computability.pdf

Chapter 12

What Is a Computer Program? I. Software vs. Hardware

program: n. 1. A magic spell cast over a computer allowing it to turn one's input into error messages.. . .
—*The Jargon Lexicon*, http://www.jargon.net/jargonfile/p/program.html
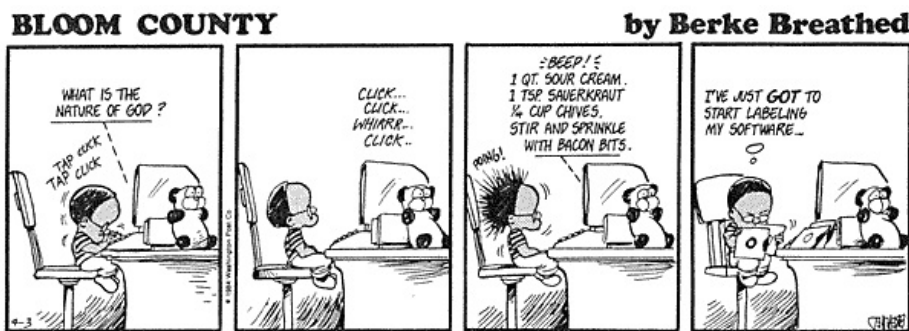
Figure 12.1: ©2010, Berkeley Breathed

12.1   Recommended Readings:

1. Required:

   - Moor, James H. (1978), "Three Myths of Computer Science", *British Journal for the Philosophy of Science* 29(3) (September): 213–222.

     – For the purposes of Ch. 12, concentrate on §§1–2.

2. Very Strongly Recommended:

   - Colburn, Timothy R. (1999), "Software, Abstraction, and Ontology", *The Monist* 82 (1): 3–19.

     – Reprinted (in slightly different form) as Ch. 12 of Colburn, Timothy R. (2000), *Philosophy and Computer Science* (Armonk, NY: M.E. Sharpe), pp. 198–209.

3. Strongly Recommended:

   - Suber, Peter (1988), "What Is Software?", *Journal of Speculative Philosophy* 2(2): 89–119.

     – Revised version accessed 8 November 2012 from:
        http://www.earlham.edu/∼peters/writing/software.htm

12.2 Introduction

[NOTE FOR NEXT DRAFT: This section could be re-written as a general introduction to the "Part" on "What Is a Computer Program?", whose first chapter would be "Software vs. Hardware".]

We have just been exploring what an *algorithm* is. We are now going to ask about the relation between algorithms and computer *programs*: In the course of the next few chapters, we will ask these questions:

- What *is* a computer program?

- Do computer programs "implement" algorithms?

- What is the nature of implementation?

- What are "software" and "hardware", and how are they related?

- Can computer programs be copyrighted, or should they be patented instead?

- Can (some) computer programs be considered to be scientific theories?

Typically, one tends to consider a computer program as an expression, in some language, of an algorithm. The language is typically a programming language such as Java, or Lisp, or Fortran. The *algorithm* is something more "abstract", whereas the *program* is something that is more "concrete". Perhaps the relationship between an algorithm and a program is something like the relationship between a number and a numeral: Just as a single algorithm, such as the algorithm for binary search, can be expressed in many different programming languages, so the number "two" can be expressed with many different numerals, such as '2' or 'II', and many different words, such as 'two', 'deux', 'zwei', and so on.

In fact, we can't really talk about numbers or algorithms without using some kind of language, so maybe there really aren't any of these abstract things called 'algorithms'. Maybe the only things are programs, some of which might be written in programming languages that can be directly used to cause a computer to execute the program (or execute the algorithm?), and some of which might be written in a natural language, such as English.

In the early days of computers programs were "hard-wired" into the computer (perhaps certain physical switches were set in certain ways). In a sense, these programs were physical parts of the computer's hardware. The program could be changed by re-wiring the computer (perhaps by re-setting the switches). Was such wiring (or switch-setting) a computer program? Yet computer programs are typically considered to be "software", not "hardware".

And what about a program written on a piece of paper? Does it differ from the very same program written on a computer file? The former just sits there doing nothing. So does the latter, but the latter can be used as input to other programs on the computer that will use the information in the program to "set the switches" so that the computer can execute the program. But is the medium on which the program is written the only difference between these two programs?

12.3   What Is Software and Its Relation to Hardware?

Our first main issue concerns the dual nature of software: It can be considered to be both text and machine (or mechanism).

To clarify this dual nature, consider this problem:

> . . . Bruce Schneier authored a book entitled *Applied Cryptography*, which discusses many commonly used ciphers and included source code for a number of algorithms. The State Department decided that the book was freely exportable because it had been openly published but refused permission for export of a floppy disk containing the same source code printed in the book. The book's appendices on disk are apparently munitions legally indistinguishable from a cluster bomb or laser-guided missile. . . . [The] disk cannot legally leave the country, even though the original book has long since passed overseas and all the code in it is available on the Internet. [Wallich, 1997, 42][1]

How can a program written on paper be considered a different thing from the very same program "written" on a floppy disk? What if the paper that the program was written on was Hayes's "magic paper" that was discussed in §9.4? But isn't that exactly what a floppy disk is, at least, when it is being "read" by a computer?

12.3.1   Moor's Theory of the Nature of Software

Figure 12.2: ©2011, Universal Uclick

Let's begin our discussion with an examination of James H. Moor's theory of the nature of software [Moor, 1978]. According to Moor, both computers and computer programs can be understood on different levels: They can be understood as physical objects, subject to the laws of physics, electronics, and so on; a computer disk containing a program would be a clear example of this level. But they can also be understood on a symbolic level: A computer can be considered as a calculating device, and a computer program can be considered as a set of instructions; the text of the computer program that is engraved on the disk would be a clear example of this level.

---

[1]This case is discussed at length in [Colburn, 1999].

So far, we merely have a restatement of the dual nature. But the larger point is that, for very many phenomena, a single entity can be viewed from multiple perspectives or "stances": The philosopher Daniel C. Dennett has suggested that a chess-playing computer or its computer program can be understood in three different ways [Dennett, 1971]:

physically:
> From the *physical stance*, its behavior can be predicted or explained on the basis of its physical construction together with physical laws. Thus, we might say that it made (or failed to make) a certain move because its logic gates numbered 5, 7, and 8 were open, or because transistor #41 was broken.

semantically:
> From the *design stance*, its behavior can be predicted or explained on the basis of information or assumptions about how it was designed or how it is expected to behave, assuming that it *was* designed to behave that way and isn't malfunctioning. Thus, we might say that it made (or failed to make) a certain move because line #73 of its program has an if-then-else statement with an infinite loop.

intentionally:
> From the *intentional stance*, its behavior can be predicted or explained on the basis of the language of "folk psychology": ordinary people's informal (and not necessarily scientific) theories of why people behave the way they do, expressed in the language of beliefs, desires, and intentions. For instance, I might explain your behavior by saying that (a) you desired a piece of chocolate, (b) you believed that someone would give you chocolate if you asked them for it, so (c) you formed the intention of asking me for some chocolate. Similarly, we might say that the chess-playing computer made a certain move because (a) it desired to put my king in check, (b) it believed that moving its knight to a certain square would put my king in check, and so (c) it formed the intention of moving its knight to that position.

Each of these "stances" has different advantages for dealing with the chess-playing computer: If the computer is physically broken, then the physical stance can help us repair it. If the computer is playing poorly, then the design stance can help us debug its program. If I am playing chess against the computer, then the intentional stance can help me figure out a way to beat it.

Moor offers a definition of 'computer program' that is neutral with respect to the software-hardware duality:

> a computer program is a set of instructions which a computer can follow (or at least there is an acknowledged effective procedure for putting them into a form which the computer can follow) to perform an activity [Moor, 1978, 214].

Let's make this a bit more precise:

Definition 1:

> Let *C* be a computer.
> Let *P* be a set of instructions.
> Then *P is a computer program for C* $=_{def}$

> > 1. there is an effective procedure for putting *P* in a form...
> >
> > 2. ... that *C* can "follow"...
> >
> > 3. ... in order to perform an activity.

We could be even more precise:

Definition 2:

> Let *C* be a computer.
> Let *P* be a set of instructions.
> Let *A* be an activity.
> Then *P is a computer program for C to do A* $=_{def}$

> > 1. there is an effective procedure for putting *P* in a form...
> >
> > 2. ... that *C* can "follow"...
> >
> > 3. ... in order to perform *A*.

Definition 2 makes the "activity" *A* a bit more perspicuous. We looked at the role of an algorithm's *purpose* in Ch. 7. But, for now, it might be easier to focus on Definition 1.

On that definition, *being a computer program* is not simply a *property* of some set of instructions. Rather, it is a binary *relation* between a set of instructions and a computer.[2] A set of instructions that is a computer program for one computer might not be a computer program for a different computer, perhaps because the second one lacks an effective procedure for knowing how to follow it.

For instance, the Microsoft Word program that is written for an iMac computer running MacOSX differs from the Microsoft Word program that is written for a PC running Windows, because the underlying computers use different operating systems and different machine languages. This would be so even if the two programs' "look and feel" (that is, what the user sees on the screen and how the user interacts with the program) were identical.

There are several questions we can ask about this definition:

- *Does* a computer "follow" instructions? Or does it merely *behave in accordance with* them? As we have seen, Turing machines—which are models of hardwired, single-purpose computers—merely behave in accordance with their machine table. On the other hand, universal Turing machines—which are models of programmable, general-purpose computers—can be said to "follow" instructions. Compare the human use of natural language: When we speak or write, do we "follow" the rules of grammar, in the sense of consulting them (even if unconsciously) before generating our speech or writing? Or does it make more

---

[2]On Definition 2, it is a *ternary* relation among a set of instructions, a computer, and an activity. But here I just want to focus on the role of the computer, which is why we're just going to consider Definition 1.

sense to say that the rules of grammar merely *describe* our linguistic behavior? Note, however, that a computer programmed to understand and generate natural language might, in fact, speak and write by explicitly following rules of grammar that are encoded in its suite of natural-language-processing programs. We have also seen a similar question when we considered whether the solar system "follows" Kepler's laws of planetary motion or whether the planets' movements are merely best described by Kepler's laws (§9.7.2). It might be useful to have a neutral term for what it is that a computer does when it is doing its processing, whether it is a Turing machine or a universal Turing machine; perhaps 'execute' could be used for this purpose. So, we could say that a Turing machine *executes* the instructions in its machine table, but doesn't *follow* them, whereas a universal Turing machine *executes* the fetch-execute cycle and thereby *follows* the program encoded on its tape.

- Is the set of instructions physical, that is, hardwired? Or are the instructions written in some language? (Could they be drawn, instead, perhaps as a flowchart? Could they be spoken?) Here, Moor's answer seems to be that it doesn't matter, as long as there is a way for the computer to "interpret" the instructions and thus carry them out. Importantly, the "way" that the computer "interprets" the instructions must itself be a computable function (it must be "effective"). Otherwise, it might require some kind of "built-in", *non*-computable method of "understanding" what it is doing.

Begin Digression
HERE, DISCUSS INTEPRETATION VS COMPILATION OF PROGRAMS, JUST FOR CLARIFICATION THAT THIS IS *NOT* WHAT I MEAN BY "INTERPRETATION".
End Digression

Figure 12.3: ©2009, Zits Partnership

Moor's next distinction is between software and hardware. The informal and traditional distinction is that a computer *program* is "software" and a *computer* is "hardware". But this raises the problem of whether the "hardwiring" in a hardwired computer is hardware (because it involves physical wires) or software (because it is

the computer's program). And, of course, it gives rise to the problem mentioned by Wallich at the beginning of this chapter. So, Moor suggests a better set of definitions:

> For a given person and computer system the software will be those programs which can be run on the computer system and which contain instructions the person can change, and the hardware will be that part of the computer system which is not software [Moor, 1978, 215].

In line with this, [Vahid, 2003, 27, my boldface, original italics] notes that, in the early days of computing, "the frequently **changing** programs, or *software*, became distinguished from the **unchaning** hardware on which they ran."

Again, let's make this a bit more precise:

Definition 3:
Let *C* be a computer.
Let *P* be a person, perhaps *C*'s programmer.[3]
Let *S* be some entity (possibly a part of *C*).
Then *S is software for* computer *C and* person *P* $=_{def}$

1. *S* is a computer program for *C*,[4] and

2. *S* is changeable by *P*.

and *H is hardware for C and P* $=_{def}$

1. *H* is (a physical) part of *C*, and

2. *H* is *not* software for *C* and *P*.

Note that *being software* is a *ternary* relation among three entities: a computer program, a programmer, and a computer;[5] it is not a simple property such that something either is, or else it isn't, software. In other words, software is in the eye of the beholder: One person's software might be another's hardware!

And a physical part of a computer will be hardware for a programmer and that computer if *either* it is not a computer program for that computer *or* it is not changeable by that programmer.

That seems to allow for the following possibility: Suppose that there is a computer program P_{Java} written in Java that runs on my computer. Even if P_{Java} is changeable by the programmer who wrote it or the lab technician who operates the computer—and therefore *software* for that person—it will be *hardware* for me if I don't know Java or don't have access to the program so that I could change it.

So, if a programmer can rewire a computer, then that computer's program is software, even if it is physical.

Question to Think About:
[Tanenbaum, 2006, 8] points out that hardware and software are "logically equivalent"

---

[3]Please note that here, '*P*' refers to a person, not a program, as in the two previous definitions.

[4]This would have to be modified to include activity *A* if we want to use Definition 2 for 'computer program'.

[5]Or it might be a *quaternary* relation among *four* things, if we include activity *A*.

in the sense that anything doable in hardware is doable in software, and vice versa. How does this relate to Moor's definition?

12.3.2 Suber's Theory of the Nature of Software

Peter Suber [Suber, 1988] argues that "software is pattern *per se*, or syntactical form" ([Suber, 1988, online abstract]). *Any* syntactical form? Yes! In other words, *everything* is software. (You might ask yourself how this relates to Searle's claim that everything is a computer!)

Does he really mean *everything*—even random bits or "noise"? Why does Suber say this? He claims that "software patterns do not carry their own meanings" [Suber, 1988, §6, p. 103]. In other words, they are purely formal syntax, meaningless marks, symbols with no intrinsic meanings. (This is consistent with the claim of [Tenenbaum and Augenstein, 1981, 6] that information has no meaning (see also http://www.cse.buffalo.edu/~rapaport/563S05/data.html).) But they need to be interpreted by a suitable computer: "They need only make a fruitful match with another pattern (embodied in a machine) [which] *we* create" [Suber, 1988, §6, p. 103, my italics]. So, software is pure syntax and needs *another* piece of syntax to interpret it. This is not far from Moor's definition: Both require someone (or something) to interpret the syntax.

[NOTE FOR NEXT TIME: THIS WILL MAKE MUCH MORE SENSE IF THERE'S AN EARLIER CHAPTER IN WHICH I EXPLAIN SYNTACTIC SEMANTICS AND HOW THEY RELATE (VIZ., TWO SYSTEMS OF SYNTAX CAN BE SEEN AS INTERPRETING EACH OTHER). See [Rapaport, 1988, Rapaport, 1995, Rapaport, 2012b].]

How does this relate to Moor's definition? Could a hardwired program and a written program both be software, perhaps because they have the same syntactic form? I think the answer is 'yes'.

Here is a possible refinement: Software is a pattern that is read*able* and execut*able* by a machine. And this is roughly Moor's definition of computer program. But, for Suber, *all* patterns are read*able* and execut*able*.

The bottom line is that Suber's notion of software is closer to Moor's notion of computer program.

12.3.3 Colburn's Theory of the Nature of Software

Finally, [Colburn, 1999, Colburn, 2000] argues that software is not a machine made out of text. Thus, he would probably disagree with Hayes's definition of a computer as "magic paper". Colburn says this because he believes that there is a difference between software's "medium of description" and its "medium of execution". The former is the text in a formal language (an abstraction). The latter consists of circuits and semi-conductors (it is concrete). Consequently, Colburn says that software is a "concrete abstraction".

He offers several analogies to positions that philosophers have taken on the mind-body problem (see §]refinanyfield); we might call this the "abstract-program/concrete-program problem". Consider monism, the view that there are not both minds and

brains, but only minds (which is the view called "idealism", associated primarily with the philosopher George Berkeley) or else only brains (the view called "materialism"). Similarly, one might hold that either software is abstract or else it is concrete, but it cannot be both. No matter how strong the arguments for, say, materialism might be as the best answer so far to the mind-body problem, monism as a solution to the software-hardware problem fails to account for its dual nature (according to Colburn).

So let's consider dualism. In the mind-body problem, there are several versions of dualism (the view that there are both minds and brains). They differ in how they explain the relationship between minds and brains. The most famous version is called 'interactionism', due to Descartes. This says that (1) there are minds (things that think, but are not physical, and that obey only psychological laws), and (2) there are brains (things that are physically extended in space and that obey only physical and biological laws), and (3) minds and brains interact. The problem for interactionism as a solution to the mind-body problem is that there is no good explanation of *how* they interact. After all, one is physical and the other isn't. So you can't give a physical explanation of how they would interact, because the laws of physics don't apply to the mind. And you can't give a psychological explanation of how they interact, because the laws of psychology don't apply to the brain. Similarly, according to Colburn, when applied to the software-hardware problem, an interactionist perspective fails to account for the *relation between* abstractions and concrete things, presumably because the relations themselves are either abstract, in which case they don't apply to concrete things, or they are concrete and, so, don't apply to abstractions. (There is, however, a possible way out, which we will explore in depth in Ch. 14, namely, perhaps the relationship between them is one of implementation, or semantic interpretation, along the lines of Suber's theory.)

There is another version of dualism, called the dual-aspect theory, due to Spinoza. Here, instead of saying that there are two different kinds of "substance", mental substance and physical substance, it is said that there is a single, more fundamental kind of substance of which minds and brains are two different "aspects". (For Spinoza, this more fundamental substance, which he believed had more than just two aspects, though we humans are only cognizant of those two, was "nature", which he identified with God.) As a solution to the software-hardware problem, Colburn points out that we would need some way to characterize that more fundamental underlying substance, and he doesn't think that any is forthcoming. Again, however, an alternative possibility is to think of how a single abstraction can have multiple implementations.

Finally, another dualism is known as parallelism: There are minds, and there are brains; they are not identical (hence this is a dualistic theory), but they operate in parallel, and so there is no puzzle about interaction. There are two versions of parallelism. One, called occasionalism, says that God makes sure that every mental event corresponds to a parallel brain event (this keeps God awfully busy on very small matters!). Another, called pre-established harmony, says that God set things up so that minds and their brains work in parallel, much in the way that two clocks can keep the same time, even though neither causally influences the other. For Colburn, this would mean that a concrete or hardwired program would behave the same way as an abstract or textual program CHECK THIS. It is also consistent with my view that a single abstraction can have two "parallel" implementations.

MAYBE FACTOR OUT MY COMMENTS ON IMPLMN AND PUT THEM HERE. Before exploring this notion of implementation in more detail (in Ch. 14) we will consider more of the ontological puzzles that arise from the dual nature of software, this time in the legal realm: Can programs be copyrighted? Or should they be patented instead?

After all, if a program is a written text, then it can be copyrighted, but, if it is a machine, then it can be patented—yet nothing can (legally) be both copyrighted and patented!

12.4   NOTES FOR NEXT DRAFT

1. The basic question, "What is a computer program?", is not necessarily answered by considering the nature of software vs. hardware, copyright vs. patent, etc. We dealt with some of the issues in Ch. 7, and so some of the ideas from there, especially those arising from my emails with Robin Hill, might be worth putting, or rehearsing, in this chapter.

   Presumably, a program is a textual expression (or, in the old, hardwired days, a physical, hardware implementation of) an algorithm, where an algorithm is something abstract, perhaps a mathematical structure of some kind, and a program is something concrete (even if only written on non-magic paper). On this view, perhaps the best analogy is that program is to algorithm as numeral is to number. Another analogy would be that of a proposition to a sentence; presumably, the English sentence "It is snowing." and the French sentence "Il neige." both express the same (abstract) proposition that it is snowing.

   So, for instance, a Turing-machine program for a binary search algorithm is not identical to that algorithm, because that very same algorithm could have been expressed in a Lisp program. The algorithm is something more abstract and independent of any expression of it in a particular programming language. Moreover, there might be several different Turing-machine programs or Lisp programs that implement binary search.

   One possible difference between these analogies is this: Presumably, there is only one number "one" and arbitrarily many numerals representing it. But given some task, say, sorting, there might be arbitrarily many algorithms that accomplish it: First, there are many different, "standard" sorting algorithms: insertion sort, Quicksort, merge sort, heapsort, etc. Second, for each of these, there are many different possible variations; for instance, Shell sort is (arguably) a variation of insertion sort. Third, for any of these, one can always interpolate a small, "do-nothing" loop in the middle of the algorithm; is a standard insertion sort with

   > **begin**
   > $x := 5$;
   >   **while** $x > 0$ **do**
   > $x := x - 1$;
   > **end**

   interpolated somewhere in the middle of it a different algorithm from the same insertion sort without that loop? So, where the standard understanding of the number-numeral relationship is that it is a one-many relation, it would seem that the algorithm-program relationship is many-many.

   On the other hand, mathematical "structuralists" have argued that numbers are not "things" (that exist is some Platonic heaven somewhere), but are more like "roles" (think of the role of Hamlet in Shakespeare's play)) in a mathematical structure (defined, say, by Peano's axioms) that can be "played" by many

different things (think of actors like Richard Burton, Laurence Olivier, etc.), such as different sets, Arabic numerals, etc. In that case, the number-numeral relationship is also many-many.

2.   Turing's work clearly showed the extensive interchangeability of hardware and software in computing. [Hartmanis, 1993, 11]

3. In a Turing machine, is its machine table software, or hardware? It's certainly hardwired. If you think that it is a kind of category mistake to talk about whether an abstract, mathematical entity such as a Turing machine can have software or hardware, then consider this: Suppose you have a hardwired, single-purpose, physical computer that (let's say) does nothing but accept two integers as input and produces their sum as output. (This would be a physical implementation of a Turing machine.) Is the program that runs this adder software or hardware? I would say that it's hardware.

   In a *universal* Turing machine, is *its* machine table (which is (a description of?) its fetch-execute cycle) software, or hardware? And what about the program that is stored on its tape? Again, if you prefer to limit the discussion to physical computers, then consider this: Suppose that you have a smartphone, one of whose apps is a calculator that also can add two integers (but, not only can the calculator do other mathematical operations, the smartphone itself can do many other things (play music, take pictures, make phone calls, etc.), *and* it can download new apps that will allow it to do many other things. (This would be a physical implementation of a universal Turing machine.) Is the program that runs the smartphone's adder (calculator) software? And is the program that allows the smartphone to do all of the above hardware? I'd say that the program that is stored on the Turing machine's tape (for example, the smartphone's calculator app) is software; therefore, the universal Turing machine's machine table (for example, the smartphone's CPU) is hardware; and, therefore, the Turing machine's machine table (for example, the adder's program) is also hardware.

4. On claims that hardware and software are logically equivalent:
   Similar or analogous cases of such logical equivalence of distinct things are Turing machines, the lambda-calculus, and recursive functions. Also, such an equivalent-but-different situation corresponds to the intensional-extensional distinction: Two intensionally distinct things can be extensionally identical.

5. On Moor's definitions of software and hardware:
   I can't modify a Java program, because I don't know Java. Is a Java program therefore hardware for me? Suppose Pat can't modify the Java program, even though Pat is a Java programmer (perhaps Pat has no access to the source code). Is that Java program hardware for Pat? These are probably different senses of 'can't'.

6. On Suber's definitions:
   How (if at all) does Suber's position relate to Searle's [Searle, 1990] claim that anything can be considered to be a computer, even the wall behind me.

7.   By fixed programming we mean the kind of programming which controls your automatic dishwasher for example. Here the sequence of operations is fixed and built into the wiring of the control or sequencing unit. Once started, the dishwasher will proceed through a regular series of operations, washing, rinsing and drying. Of course, if one wished, one could change the wiring to alter the program. [Samuel, 1953, 1226–1227].

This is like a Turing machine's hardwired program. Note, by the way, that modern dishwashers allow for some "programming" by pushbuttons that can alter its operations.

For the universal Turing machine's software (program stored on its tape), Samuel has this analogy:

Suppose you wished to give your assistant a large number of instructions for manual computations all in advance. You could do this by supplying him with a prepared set of instructions, or you could dictate the instructions and have him write them down, perhaps at the top of the same sheet of paper on which he is later to perform the computations. Two different situations are here involved, although at first glance the distinction appears trivial. In the first case the instructions are stored on a separate instruction form, while in the second case they are stored by the same medium which is used for data. Both situations are found to exist in computing machines. The first case is exemplified by certain machines which use special program tapes. The second situation is becoming quite common in the newer machines and is the case usually meant when the term "stored program" is used. [Samuel, 1953, 1227]

We can now go back and consider one property of the "stored program" method of operation which is rather unique and which really must be understood to appreciate the full value of such a concept. This property is that of being able to operate on the instructions themselves just as if they were ordinary data. This means that the entire course of a computation can be altered, including the operations themselves, the choice of data on which the operations are to be performed, and the location at which the results are to be stored, and this can all be done on the basis of results obtained during the course of the calculations through the use of conditional transfer instructions. [Samuel, 1953, 1228]

Shades of Hayes's "magic paper"!

8. On the difference between software (program) and the process produced by it when executed:

In software culture, we no longer have "documents." Instead, we have "software performances."

> I use the word "performance" because what we are experiencing is constructed by software in real time. Whether we are exploring a website, playing a video game, or using an app on a mobile phone to locate nearby friends or a place to eat, we are engaging with the dynamic outputs of computation.
>
> Although static documents may be involved, a scholar cannot simply consult a single PDF or JPEG file the way 20th-century critics examined a novel, movie, or TV program. ...For instance, a user of Google Earth is likely to experience a different "earth" every time he or she uses the application. Google could have updated some of the satellite photographs or added new Street Views and 3D buildings. [Manovich, 2013, B11]

Not only does this emphasize what a process is, but it also points out an interesting difference, relevant to copyright issues that we will explore in Ch. 13, between the static, written text of a program and the potentially dynamic version engraved on an executable CD. A similar point is made later in Manovich's essay:

> The attraction of "reading the code" approach for the humanities is that it creates an illusion that we have a static and definite text we can study—i.e., a program listing. But we have to accept the fundamental variability of the actual "software performances." So rather than analyzing the code as an abstract entity, we may instead trace how it is executed, or "performed," in particular user sessions. To use the terms from linguistics, rather than thinking of the code as language, we may want to study it as speech. [Manovich, 2013, B13]

9. On Dennett's intentional stance:
   See the discussion of "levels of description" in [Newell, 1980, §6]. One major difference between Newell and Dennett is that the former holds that "these levels of description do not exist just in the eye of the beholder, but have a reality in...the real world" [Newell, 1980, 173].

10. The earliest use of the word 'software' in its modern sense has been traced back to [Tukey, 1958, 2]:

    > Today the "software" comprising the carefully planned interpretive routines, compilers, and other aspects of automative [sic] programming are at least as important to the modern electronic calculator as its "hardware" of tubes, transistors, wires, tapes and the like.

    (See [Shapiro, 2000a], [Mahoney and Haigh (ed.), 2011, 238, n. 20], and http://www.historyofinformation.com/expanded.php?id=936 for more information on the history of the word, which is likely older than Tukey's use.) This use of the term strongly suggests that the things that count as software are more abstract than the things that count as hardware.

Interestingly, an earlier citation than Tukey's, [Carhart, 1956, 149], equates software with the *people* who operate a computer system (where the computer system is identified as the hardware); programs (or other modern notions of software) are not mentioned. (An even earlier use, dating to 1850 [Shapiro, 2000a, 69], equates it with "vegetable and animal matters—everything that will decompose" in the realm of "rubbish-tip pickers"; insofar as decomposition is a form of changeability, this is, at least, consistent with Moor's interpretation!)

11. **Exercise for the Reader:**
    Find a (short) article on the mind-body problem (for example, the *Wikipedia* article at http://en.wikipedia.org/wiki/Mind-body_problem). Replace all words like 'mind', 'mental', etc., with words relating to 'software'; and replace all words like 'body', 'brain', etc., with words like 'hardware', 'computer', etc. Discuss whether your new paraphrased article makes sense, and what this says about the similarities (or differences) between the mind-body problem and the software-hardware problem.[6]

12. On the idea that some kinds of software, in particular, computer programs (and the activity of programming) are mathematical in nature, see [Dijkstra, 1974, Scherlis and Scott, 1983, Hoare, 1986]. Hoare, for example, says that "computer programs are mathematical expressions" (p. 115). This suggests that (at least some kinds of) software are abstract, mathematical entities (or linguistic expressions thereof, perhaps more along the lines of numerals than of numbers).

13. [Vahid, 2003, 27, 31, 32] suggests that "the processors, memories, and buses— what we previously considered a system's unchangeable hardware—can actually be quite soft" (p. 32). What he seems to mean by this is that *embedded systems*—"hidden computing systems [that] drive the electronic products around us" (p. 27)—can be swapped for others in the larger systems that they are components of, thus becoming "changeable" in much the way that software is normally considered to be. But this seems to me to just be the same as the old rewiring of the early days of programming (except that, instead of changing the wires or switches, it is entire, but miniaturized, computers that are changed).

14. In an unpublished talk given to the Center for Cognitive Science at SUNY Buffalo (21 September 2005), Amnon H. Eden offers three ways to think about what a computer program is. The first is as a "mathematical expression" (citing the work of C.A.R. Hoare). This is consistent with the view mentioned elsewhere in this chapter of a program being like a numeral, where an algorithm is like a number. But it is a bit different, I think; here, the idea is that programs themselves are mathematical objects capable of being studied mathematically. We will return to this theme in Chapter 16.

    The second way is as a "natural kind" (see §3.4). Computer programs exist, so we should study them as we find them in nature, so to speak. We can try to

---
[6]Thanks to James Geller for this idea.

categorize them, identify their properties, determine there relative merits, and so on.

The third way is "as an engineered artefact". Whereas the study of programs as natural kinds is more or less descriptive, the study of them as engineered artifacts is "normative": We should try to specify ways in which programs can be designed so as to be more efficient, more reliable, etc.

Finally, Eden offers a fourth way: "as a cognitive artefact: Software is conceived and designed at a level of abstraction higher than the programming language".

15. An example of the algorithm-program relationship that is along the lines of the number-numeral relationship is offered by Stewart Shapiro: When trying to study a dynamic *process* mathematically,

> It is common practice in mathematics...[to associate] a class of (static) mathematical objects with the process in question and then 'transferring' results about the mathematical objects to the processes. In mathematical logic, for example, a process of deductive reasoning is studied through the use of formal deductions; in mathematical physics, the movement of particles is studied by means of an associated class of curves in geometrical space.... ...The technique employed in the theory of computability is to associate an algorithm with a written set of instructions for computation, usually in a formal language, and to study the class of algorithmic descriptions syntactically. [Shapiro, 2000b, 204]

16. On Moor's changeability criterion:
[Piccinini, 2008, §§3.1.1–3.1.2, p. 40] distinguishes between "hard programmability" and "soft programmability": The former refers to "the mechanical modification of...[a computer's] functional organization; the latter refers to "modification involv[ing] the supply of appropriately arranged digits (instructions)" "without manually rewiring any of the components". This can be viewed either as a further breakdown of software into these two kinds, or else as limiting the changeability to "soft" changeability.

17. [Irmak, 2012] argues that software is an "abstract artifact", where an "artifact...is an intentional product of human activity" (pp. 55–56) and an "abstract" object is one that is "non-spatial" but that "may or may not have some temporal properties" (p. 56). Irmak likens software to another abstract artifact: musical works (§2, pp. 65ff). I agree that there are close similarities. For instance, I think that a Turing machine (or any hard-wired computer that can perform only one task) is like a music box that can only play one tune, whereas a player piano is like a universal Turing machine.

One difference between software and music that Irmak points out concerns "a change or a revision on a musical work once composed" (p. 67). This raises some interesting questions: How should musical adaptations or jazzy versions of a piece of music be characterized? What aobut different player's

interpretations?: One pianist's version of, say, Bach's *Goldberg Variations* will sound very different from another's, yet, presumably, they are using the same "software". Are there analogies to these with respect to software? (In addition, problems about small changes in software seem analogous to issues of personal identity through time.)

[Irmak, 2012, 68] says, "the idea that software and musical works are created is...central to our beliefs". Perhaps, but here there are similarities with issues in the philosophy of mathematics: Are theorems or proofs similarly "created"? Or should we take a mathematically Platonic attitude towards all of these kinds of things, and revise our "manifest image" view that they are *created* to a more "scientific image" view that they are *discovered*? Mendelssohn (or Mill?) is alleged to have been depressed when he learned that there were only a finite number of possible combinations of notes, hence only a finite number of possible musical compositions.

For more on the relationships between software and improvisational music, see §10.5.2.

18.   [H]ardware is at least tangible. Software by contrast is elusively
         intangible.   In essence, it is the behavior of the machines when
         running.   It is what converts their architecture to action....
         [Mahoney and Haigh (ed.), 2011, 33].

On this view, programs (as texts) are part of the hardware; it is the *process* that is the software. Whether or not one wants to draw the boundary there, it does point out the different role of process from both computer and program. As a historian, Mahoney is also concerned with the fact that processes are ephemeral: They leave no record, and are difficult, if not impossible, to repeat (legacy software can only be run on machines that no longer exist).

12.5 Further Sources of Information

- Bond, Gregory W. (2005), "Software as Art", *Communications of the ACM* 48(8) (August): 118–124.

  1. "A programming language is a language after all, albeit a hightly constrained one. As such, it is a perfect medium for the poet comfortable with other highly constrained poetic forms like the sonnet or haiku" (p. 123).

  2. On the literary value of programs, see also:

     (a) Chandra, Vikram (2014), *Geek Sublime: The Beauty of Code, the Code of Beauty* (Graywolf Press).
         - See also:
           Gleick, James (2014), "A Unified Theory" (review of Chandra 2014), *New York Times Book Review* (24 August): 17; accessed 8 September from:
           http://www.nytimes.com/2014/08/24/books/review/geek-sublime-by-vikram-chandra.html

         Knuth, Donald E. (1984), "Literate Programming", *The Computer Journal* 27(2): 97–111.
         i. Accessed 29 Jully 2014 from:
            http://comjnl.oxfordjournals.org/content/27/2/97.full.pdf+html
         ii. Reprinted in:
            Knuth, Donald E. (1992), *Literate Programming* CSLI Lecture Notes 27 (Stanford, CA: Center for the Study of Language and Information), http://www-cs-faculty.stanford.edu/∼uno/lp.html

     (b) Schneider, David (2007), "Neatness Counts: Messy Computer Code Is Normally Frowned On—But Not Always", *American Scientist* (May-June): 213–214
         - Accessed 1 August 2014 from:
           http://www.americanscientist.org/issues/pub/neatness-counts

     (c) Turner, Zeke (2015), "Beautiful Code", *The New Yorker* (30 March): 21–22, http://www.newyorker.com/magazine/2015/03/30/beautiful-code (accessed 4 April 2015)
         - On the first art auction of computer code.

  3. For some examples of "aesthetic" or "playful" programs, see:

     (a) "Most Adaptable Programs" (1991, 1996), accessed 29 July 2014 from:
         http://www2.latech.edu/∼acm/helloworld/multilang.html
         (two examples of programs that can be compiled and run in several different programming-language systems, as if there were a sentence that was grammatical and meaningful in both English and French—is there such a sentence?)

     (b) "dodsond1", Text to "Pig Latin" translator, 12th International Obfuscated C Code Contest (1995), accessed 29 July 2014 from:
         http://www.ioccc.org/1995/dodsond1.c
         and
         http://www.ioccc.org/years-spoiler.html
         (must be seen to be believed!)

- Duncan, William (2009), "Using Ontological Dependence to Distinguish between Hardware and Software", accessed 1 August 2014 from:
  https://www.jiscmail.ac.uk/cgi-
  bin/webadmin?A3=ind1103&L=SWORD&E=base64&P=519469&B=————
  050201060409000909050704&T=application

  - Argues that a formal ontology is needed before a useful distinction between hardware and software can be made, and that, on such an ontology (the Basic Formal Ontology), "a piece of computational hardware is an ontologically independent entity, whereas a software program is an ontologically dependent entity."

- Eden, Amnon H.; & Turner, Raymond (2007), "Problems in the Ontology of Computer Programs", *Applied Ontology* 2(1): 13–36.

  - Accessed 1 August 2014 from:
    http://www.eden-study.org/articles/2007/problems-ontology-programs_ao.pdf

- Kroes, Peter (2010), "Engineering and the Dual Nature of Technical Artefacts", *Cambridge Journal of Economics* 34: 51–62.

  - Argues that "technical artefacts are physical structures with functional properties"

- Mahoney, Michael Sean (2011), *Histories of Computing*, ed. by Thomas Haigh (Cambridge, MA: Harvard University Press).

  - Ch. 13, "Extracts from *Software as Science—Science as Software*", discusses the history of software; the section titled "Extract 3: Software as Science" is especially interesting.

- Miller, Christopher A. (guest ed.) (2004), "Human-Computer Etiquette: Managing Expectations with Intentional Agents", *Communications of the ACM* 47(4) (April): 31–34.

  - Contains some interesting follow-ups from the perspective of computer science to Dennett's theory of different "stances".

- Turner, Raymond; & Eden, Amnon H. (2007), "Towards a Programming Language Ontology", in Gordana Dodig-Crnkovic & Susan Stuart (eds.), *Computing, Information, Cognition—The Nexus and the Liminal* (Cambridge, UK: Cambridge Scholars Press): 147–159.

  - Accessed 1 August 2014 from:
    http://www.eden-study.org/articles/2007/towards-pl-ontology_dodigstuart.pdf

Chapter 13

What Is a Computer Program?
II. Can Software Be
Copyrighted or Patented?

Figure 13.1: ©2010, *American Scientist*

497

Figure 13.2: ©2007, UFS, Inc.

13.1   Recommended Readings:

1. Required:

   - Newell, Allen (1985-1986), "The Models Are Broken, the Models Are Broken",
     University of Pittsburgh Law Review 47: 1023–1031.

     – Accessed 5 August 2014 from:
       http://digitalcollections.library.cmu.edu/awweb/awarchive?type=file&item=356219
     – For follow-up (including links to commentary by Donald Knuth), see:
       "Newell 1986: The Models Are Broken", accessed 5 August 2014 from:
       https://web.archive.org/web/20120527101117/http://eupat.ffii.org/papers/uplr-newell86/index.en.html

2. Very Strongly Recommended:

   - Samuelson, Pamela (1990), "Should Program Algorithms Be Patented?",
     Communications of the ACM 33(8) (August): 23–27.

3. Strongly Recommended:

   - Koepsell, David R.; & Rapaport, William J. (1995), "The Ontology of Cyberspace:
     Questions and Comments", *Technical Report 95-25* (Buffalo: SUNY Buffalo
     Department of Computer Science) and *Technical Report 95-09* (Buffalo: SUNY
     Buffalo Center for Cognitive Science)

     – Accessed 9 November 2012 from:
       http://www.cse.buffalo.edu/~rapaport/Papers/cyber.pdf
     – This document consists of two papers, both originally presented at the Tri-
       State Philosophical Association Meeting, St. Bonaventure University, 22 April
       1995:
       (a) Koepsell, David (1995), "The Ontology of Cyberspace: Preliminary
           Questions"
       (b) Rapaport, William J. (1995), "Comments on 'The Ontology of
           Cyberspace' "
     – See esp. Koepsell's §§1 and 4, and Rapaport's §§2 and 3.

13.2 Introduction

We are trying to understand what algorithms, computer programs, and software are, and how they are related to each other. One way to approach these questions is by considering the legal issues of whether any of them can be copyrighted or patented. (There is also a legal notion of "trade-secret protection", which we will not explore.)[1]

One of the first questions is what kind of entity might be copyrighted or patented:

- algorithms?
  (These seem to be abstract, mathematical entities. Can abstract, mathematical entities—such as numbers, formulas, theorems, proofs, etc.—be copyrighted or patented?)

- computer programs?
  (These seem to be "implementations" of algorithms, expressed in a programming language. We have seen that programs might be analogous to *numerals* (as opposed to *numbers*)—can numerals be copyrighted or patented?)

- programs written on paper?
  (These might seem to be "literary works", like poems or novels, which can be copyrighted but not patented.)[2]

- programs implemented in hardware (that is, "machines")?
  (If programs are implementations of algorithms, then are hardware implementations of programs thereby implementations of implementations of algorithms? And physical implementations might seem to be the kind of thing that is patentable but not copyrightable.)

- software?
  ('Software' might be a neutral term covering both algorithms and programs, or software might be a more controversial entity not necessarily indistinguishable from certain kinds of hardware, as we saw in the previous chapter.)

Here are some excerpts from the legal definitions. The full sources are: "Copyright Basics", accessed 5 August 2014 from: http://www.copyright.gov/circs/circ01.pdf and "United States Patent and Trademark Office", accessed 5 August 2014 from http://www.uspto.gov/patents/index.jsp

Note what is said about computer programs, procedures, methods, and processes!

Copyright:
"Copyright is a form of protection provided by the laws of the United States (title 17, *U.S.Code*) to the authors of 'original works of authorship,' including literary, dramatic, musical, artistic, and certain other intellectual works. ... [T]the owner of copyright [has] the exclusive right to do and to authorize others to do the following:

---

[1] But see the Further Readings for essays dealing with trade secrets, especially Bender 1985-1986, and also [Samuelson et al., 1994, §§2.2.1,5.3.3].

[2] See the Further Readings for Ch. 12 for some references to work on the literary value of programs, at Bond 2005.

- reproduce the work. . .

- prepare derivative works based upon the work

- distribute copies. . . of the work to the public by sale or other transfer of ownership, or by rental, lease, or lending

- perform the work publicly, in the case of literary, musical, dramatic, and choreographic works, pantomimes, and motion pictures and other audiovisual works

- display the work publicly, in the case of literary, musical, dramatic, and choreographic works, pantomimes, and pictorial, graphic, or sculptural works, including the individual images of a motion picture or other audiovisual work. . .

"What Works Are Protected?

Copyright protects 'original works of authorship' that are fixed in a tangible form of expression. The fixation need not be directly perceptible so long as it may be communicated with the aid of a machine or device. Copyrightable works include the following categories:

1. literary works

2. musical works, including any accompanying words

3. dramatic works, including any accompanying music

4. pantomimes and choreographic works

5. pictorial, graphic, and sculptural works

6. motion pictures and other audiovisual works

7. sound recordings

8. architectural works

"These categories should be viewed broadly. For example, computer programs and most 'compilations' may be registered as 'literary works'; maps and architectural plans may be registered as 'pictorial, graphic, and sculptural works.'

"What Is Not Protected by Copyright?

Several categories of material are generally not eligible for federal copyright protection. These include among others:

- works that have not been fixed in a tangible form of expression (for example, choreographic works that have not been notated or recorded, or improvisational speeches or performances that have not been written or recorded)

- titles, names, short phrases, and slogans; familiar symbols or designs; mere variations of typographic ornamentation, lettering, or coloring; mere listings of ingredients or contents

- ideas, procedures, methods, systems, processes, concepts, principles, discoveries, or devices, as distinguished from a description, explanation, or illustration

- works consisting entirely of information that is common property and containing no original authorship (for example: standard calendars, height and weight charts, tape measures and rulers, and lists or tables taken from public documents or other common sources)"

Patents:

> "A patent is an intellectual property right granted by the Government of the United States of America to an inventor 'to exclude others from making, using, offering for sale, or selling the invention throughout the United States or importing the invention into the United States' for a limited time in exchange for public disclosure of the invention when the patent is granted.

> "There are three types of patents. **Utility patents** may be granted to anyone who invents or discovers any new and useful process, machine, article of manufacture, or composition of matter, or any new and useful improvement thereof. … **Design patents** may be granted to anyone who invents a new, original, and ornamental design for an article of manufacture. **Plant patents** may be granted to anyone who invents or discovers and asexually reproduces any distinct and new variety of plant."

Furthermore, "utility patents" include "inventions involving computer programming, computer program listings" (http://www.uspto.gov/patents/resources/types/utility.jsp).

13.3 Preliminary Considerations

Let's begin by trying to distinguish between algorithms and "corresponding" computer programs. Recall that algorithms compute functions. So let us begin by considering some function, that is, some set of input-output pairs that are functionally related. That function can be "implemented" by different algorithms.

What do I mean by 'implemented'? We'll explore that in Ch. 14. A synonym is 'realized' (that is, made real). For now, we will just say that "implementation" is the relation between a function and an algorithm that computes it: Any such algorithm "implements" that function.

Take a look at Figure 13.3 (at the end of this chapter). In the figure, merely as an example, I consider a function being implemented by two different serial algorithms and one parallel algorithm.

(For a concrete example, consider a sorting function that takes as input a set of rational numbers and that yields as output a sequence of those numbers sorted in increasing order. One serial algorithm that implements this function is Quicksort; another is Merge Sort. A parallel algorithm that implement the function might be a parallel version of Merge Sort.)

Take a look at serial algorithm 1. The figure suggests that it might be expressable in two different programming languages: a parallel programming language and a serial programming language.

Consider now our original function implemented by serial algorithm 1 that is, in turn, expressed in serial programming language 2. There might be several ways in which that program might itself be implemented:

- on a virtual machine—a theoretical model of computation for our serial programming language 2,

- on an actual computer that uses our serial programming language 2 as its machine language,

- compiled into a serial machine language,

- compiled into a different serial machine language,

- compiled into a parallel machine language.

And now consider parallel algorithm 3 that implements our original function. It might be expressed in a *serial* programming language 3, which itself might be implemented in a virtual machine, an actual computer, or compiled into several different machine languages. Or it might be expressed in a *parallel* programming language 4, which itself might be compiled into serial machine language 5 or else into a parallel machine language.

And so on.

Which (if any) of these should be copyrightable?

Which (if any) of these should be patentable?

13.4   Copyright

What does it mean to copyright something? For a person to copyright a *text* is to give that person the legal *right* to make *copies*—hence the term 'copyright'—of that text for the purpose of protecting the *expression of ideas*. An *idea* cannot be copyrighted; only the *expression* of an idea can be copyrighted. Ideas are abstract, perhaps in the way that algorithms are abstract. But if you express that idea in language—if you "implement" it in language—then the expression of that idea in that language *can* be copyrighted, perhaps in the way that a computer program that implements an algorithm might be copyrightable.

Consider [Mooers, 1975, 50] on the distinction between an uncopyrightable idea and a copyrightable expression of an idea:

> Where does the "expression" leave off, and the "idea" take over? The best insight into this matter comes from discussions of copyright as applied to novels and dramatic productions. In these, "expression" is considered to include the choice of incident, the personalities and development of character, the choice of names, the elaboration of the plot, the choice of locale, and the many other minor details and gimmicks used to build the story. In other words, "expression" is considered to include not only the marks, words, sentences, and so on in the work, but also all these other details or structures as they aggregate into larger and larger units to make up the expression of the entire story.
>
> In other words, after the bare abstract "idea" has been chosen (e.g., boy meets girl, boy loses girl, boy wins girl), the "expression" to which copyright applies covers the remaining elements of original choice and artistry which are supplied by the author in order for him [sic] to develop, express, and convey his version of the bare idea.

Recall FROM SOME EARLIER CHAPTER ON SYN VS SEM our picture of two domains: the world and our description of the world in some language. The former is

the realm of a semantic interpretation of the latter, which is the realm of syntax. Perhaps copyright is something that applies only to the realm of syntax, whereas patents are things that apply only to entities in the world.

Algorithms, construed abstractly, seem more like "ideas" than like "expressions", which would suggest that they cannot be copyrighted. On this view, it would be a computer program—a text written in some language—that would be copyrightable. Programs are definitely expressions: They are "non-dramatic literary works". Why "literary"? After all, they don't seem to read like novels! (But see the references cited at the end of Ch. 7 on "literate computing".) But all that 'literary' means in this context is that they can be written and read. Moreover, programs can be "performed", that is, executed, just like lectures, plays, or movies.

On the other hand, there is an entity that is even more abstract than an algorithm, and with respect to which it is the algorithm that appears to be a detailed "expression", namely, the *function* that the algorithm computes. So, one could argue that it is a *function* that is more like a "bare abstract idea" (boy meets girl, etc.) and that it is an *algorithm*—which would have to specify *how* the boy meets the girl, etc.—that is more like an expression (in this case, an "expression" of a function). On this view, it is the *algorithm* that would be copyrightable!

So, here are two problems to consider:

Problem #1

Consider the input-output behavior of a program-being-executed (a "process'; see §3.7). That is, consider how the process "looks and feels" to a user. A clever programmer could create a new algorithm (or recreate an old one) and then implement a new program that will have the same look and feel. Should that new program be separably copyrightable?

Consider two computer designcompanies; call them 'Pear Computers' and 'Macrohard'. Pear Computers might write an operating system with "windows", "icons", and a "mouse". Macrohard, seeing how this operating system works, might write their own operating system that also uses "windows", "icons", and a "mouse" and that would have the same (or almost the same) functionality and the same (or almost the same) "look and feel", but it would be a different expression of that same (or almost the same) idea, hence separably copyrightable. Or would it?[3]

Problem #2

It is part of copyright law that you cannot copyright an "article of utility"; that's something that can only be patented.
IS THAT RIGHT? NEED TO FIND SOURCE OF QUOTE
But surely an executable program is useful; that is, it is an "article of utility".

---

[3]This is not far from the actual situation that allegedly existed between the Xerox Palo Alto Research Center and Apple Computer, on the one hand, and (later) between Apple and Microsoft, on the other. The story is that Apple "borrowed" or was inspired by Xerox PARC's creation of the graphical user interface and used it for its Mac OS, and that Microsoft "borrowed" or was inspired by Apple's version and used it for Windows. See [Fisher, 1989], [Samuelson et al., 1994, 2334–2335], [Gladwell, 2011], and
http://en.wikipedia.org/wiki/Apple_Computer,_Inc._v._Microsoft_Corp. (accessed 8 February 2014).

But, according to Mooers, it is a translation of a copyrightable text; hence, it, too, is copyrightable.

One diagnosis of these problems is that there is no legal definition of "look and feel" [Samuelson, 1989]. Another diagnosis is that the distinction between an idea and its expression may not be applicable in the realm of software [Samuleson, 1991], [Koepsell, 2000]. We will explore this in a later section.
WHERE? NEED A SECTION ON KOEPSELL!

13.5   Patent

Patents are weird: Not just *any*thing can be patented, but lots of strange things *can* be! (I have read patents for devices or techniques that claim to be based on carefully developed scientific theories, but that I know, from personal experience, are not based on any such thing, and I have read patents that claim to provide mechanisms for completely solving problems that have not yet been solved, and may never be—such as fully automatic, high-quality machine translation.)

The purposes of a patent are to foster invention and to promote the disclosure of an invention. So, the inventor's *ideas* must be made *public*. (Does that mean that they have to be implemented?)

It follows that you cannot patent something that would *inhibit* invention, namely, you cannot patent any of the following:

- abstract ideas

- mathematical processes

- scientific principles

- laws of nature (and yet genes have been patented!)

- mental processes

But you *can* patent "any new and useful process, machine, article of manufacture, or composition of matter, or any new and useful improvement thereof" (see the definition of 'patent', above). Does "process" here include software?

Here are some problems to consider:

- Each step of certain cognitive algorithms (such as those used in reasoning, perception, or natural-language generation and understanding) are mental processes. Does that mean that some algorithms cannot be patented, even if others can? If so, how does one distinguish between these two kinds of algorithms?

- Lots of software are mathematical processes. Does this mean that some, but not all, software cannot be patented?

- If too much software is patented, that might *impede* invention, contrary to one of the explicit purposes of patent law.

13.6 Allen Newell's Analysis

In this section, we will look at one person's analysis of these problems. But it is not just any one person; it is a well-respected computer scientist. And it is not just any computer scientist; it is one of the founders of the fields of AI and of cognitive science, and a winner of the Turing Award: Allen Newell. In a paper written for a law journal [Newell, 1986], Newell argues that the "conceptual models" of algorithms and their uses are "broken", that is, that they are "inadequate" for discussions involving patents (and copyrights).

Newell's definition of 'algorithm' is very informal, but "perfectly reasonable, not arcane; we can live with it": An algorithm is "an unambiguous specification of a conditional sequence of steps or operations for solving a class of problems". (See §7.5, above, and [Chisum, 1986, "The Definition of an Algorithm", §B, pp. 974–977] for other definitions.)

TO BE ADDED, MAYBE IN AN APPENDIX: THESE OTHER DEFS; SEE http://www.cse.buffalo.edu/~rapaport/584/c-vs-pat.html item 3.

Algorithms are mathematical objects, but there are non-mathematical algorithms, so, Newell points out, you cannot make a decision on the patentability of algorithms on the basis of a distinction between mathematical processes and non-mathematical ones.

Moreover, "humans think *by means of* algorithms" (my italics; CITE). This is a very strong version of a thesis in the philosophy of mind called "computationalism". It is, in fact, one reason why Newell is so famous: Newell and Nobel-prize winner Herbert Simon wrote one of the first AI programs, which they interpreted as showing that "humans think by means of algorithms" (see [Newell et al., 1958]). Note that Newell doesn't say what *kind* of algorithms he has in mind here, nor does he say whether they are Turing-machine–equivalent. His point is that some algorithms are sequences of *mental* steps; so, you can't make patentability decisions based on any alleged distinction between mental processes and algorithms.

He also points out that you cannot patent natural laws or mathematical truths (for instance, you cannot patent the integers, or addition). Are algorithms natural laws? (In §10.4, we saw that [Shapiro, 2001] thinks that they are.) Are they mathematical truths? They are certainly among the items that some mathematicians study. In either case, algorithms would not be patentable. On the other hand, they *are* "applicable", hence patentable. But *all* of computer science concerns application, and algorithms are certainly part of computer science's theoretical foundations.

In addition, Newell points out, it is difficult to distinguish an algorithm (an abstract program) from a computer program (which adds implementation details that are not, strictly speaking, part of the algorithm), because there are only *degrees* of abstraction. For example, you could have a really good interpreter that would take an algorithm and execute it directly (this is the holy grail of programming: to talk to a computer in English and have it understand you without your having to program it to understand you).

But, contrary to his own definition, he points out that algorithms need only be "specifications that determine the behavior of a system", not necessarily *sequences* of steps. Programs in Prolog or Lisp, for instance, are *sets*, not sequences, of statements. So, you can't identify whether something is an algorithm just by looking at it.

The bottom line is that there are two intellectual tasks: A *philosophical and computer-scientific task* is to devise good models ("ontologies") of algorithms and other computer-scientific items. This is a task that people like Amnon Eden, Barry Smith, Brian Cantwell Smith, and Ray Turner are working on. (See [Smith, 1996, Smith, 2002, Eden, 2007], CITE TURNER, CITE BARRY SMITH)

A *legal task* is to devise good legal structures to protect these computational items. This is a task that people like David Koepsell and Pamela Samuelson investigate. (See [Koepsell, 2000] and Samuelson's writings in the "Further Sources of Information" at the end of this chapter.)

I close with a quotation from [Newell, 1986, 1035, my italics]:

> I think fixing the models is an important intellectual task. It will be difficult. The concepts that are being jumbled together—methods, processes, mental steps, abstraction, algorithms, procedures, determinism—ramify throughout the social and economic fabric.... The task is to get...new models. There is a fertile field to be plowed here, to understand what models might work for the law. It is a job for lawyers and, importantly, theoretical computer scientists. *It could also use some philosophers of computation, if we could ever grow some.*

Readers of this book, take note!

13.7 NOTES FOR NEXT DRAFT

1. In case I need a definition of 'virtual machine', here's a possibility: A virtual machine is a Turing machine that is simulated by a universal Turing machine. In other words, let t be a Turing machine. Let u be a universal Turing machine. Encode all Turing machines using, say, Turing's coding scheme. Then encode t's code onto u's tape, along with t's data, and let u simulate t. As [Copeland, 1998, 153] says, "the universal machine will perform evry operation that t will, in the same order as t (although interspersed with sequences of operations not performed by t)". The virtual t machine is a software version (a software implementation?) of the hardware t machine. (See also:

   ```
   \item
   Pollock, John L.\
   (2008),
   ``What Am I?
   Virtual Machines and the Mind/Body Problem'',
   {\it Philosophy and Phenomenological Research\/}
   76(2) (March): 237--309.
   ```

   for another possible defintion.)

   [Samuelson et al., 1994, §1.3] discusses the notion in terms of "conceptual metaphors" (such as the metaphor of "paper" for a word-processing program):

   > Cutting and pasting (in a virtual sense) is so easily accomplished in a word processor tht one soon begins to think of the text as a physical entity that can be picked up and moved around. (p. 2325)

   By 'virtual' here, they mean "metaphorical". Note, in connection with our discussion of magic in Ch. 3, that this "virtual" paper is an "illusion": As Samuelson et al. note, we have the illusion that, when inserting text, "the old text obligingly moves over to make room for the new words" (p. 2324).

   > Thge automated spreadsheet metaphor has so fundamentally changed the experience of using a computer that users feel they are using a spreadsheet, not just a computer. (p. 2325)

   They go on to argue that, because "conceptual metaphors" (by which, I take it, they mean virtual machines) "are remote from the program text. . . and partly because of their abstract character, they woud likely be regarded as unprotectable by copyright law" (p. 2326).

2. [Galbi, 1971] offers a useful history of computer patents. One point that becomes clear from reading it is that the issue of whether software can or should be patented or copyrighted is not merely an ontological issue. It is also, perhaps even more so, an economic issue.

   Initially, "a computer operating in accordance with a particular program" was patentable "if the program is new, useful, and unobvious" [Galbi, 1971, 274].

That is, it was the physical machine executing the program that was patentable. This was later modified to allow the "method or algorithm for solving certain problems" to be patentable "irrespective of what apparatus is used to solve the problem" [Galbi, 1971, 274]—a clear move away from the patentability of a piece of hardware to the patentability of its software instead. Arguably, though, this could be seen as a move to say that it is still a machine that is patentable, but it doesn't have to be a specific machine: It could be any machine that is executing the program under consideration.

Related to this is the following excerpt from a footnote to a 1969 ruling in a case called *Prater and Wei*:

> In one sense, a general purpose digital computer may be regarded as a storeroom of parts and/or electrical components. But once a program has been introduced, the general purpose digital computer becomes a special purpose digital computer (i.e. a specific electrical circuit with or without electromechanical components) which along with the process by which it operates, may be patented, subject, of course, to the requirements of novelty, utility and nonobviousness.

But think of a smartphone capable of running multiple "apps" or a computer capable of running multiple programs—more generally, of a universal Turing machine capable of simulating multiple Turing machines. Might a (physical implementation of) universal Turing machine running one program be patentable, whereas the very same machine running a different program not be patentable? Is it one machine, or many?

An aspect of the economic, rather than purely philosophical/ontological, issue concerns "whether copyright protection is narrower or broader than patent protection"; this "is subject to some degree of disagreement", but "the most that can clearly and absolutely e said is that the nature of protection granted by a patent is fundamentally differentfrom thetype of protection granted by a copyright" [Galbi, 1971, 274n1]. And another economic aspect concerns "whether or not these patents are, in fact, enforceable" [Galbi, 1971, 277].

3. Samuelson 1990, p. 23 (required readings), points out that computer scientists differ on what is the proper object of legal protection: code (or program), look and feel, algorithm. So, these need to be distinguished. What is "look and feel"? Is it related (or identical) to the "process", that is, the program as it is being executed? Is it the output of the program?

4.   Although programs are texts and their texts can be valuable, the most important property of programs is their behavior (i.e., the set of results brought about when program instructions are executed). [Samuelson et al., 1994, 2314]

        [The primary source of value in a program is its behavior, not its text. [Samuelson et al., 1994, 2315]

Does this mean that it's the "idea", not its "expression" that counts? I don't think so, simply because the "behavior" of a program is more than merely the "idea". (The "idea" might be more akin to the "specifications", or the input-output behavior.) But "behavior" here might be something akin to "look and feel". So, what to the authors mean, exactly, by "behavior"?

5. A second feature of programs is that the behavior of one program can be precisely reproduced by a textually entirely different program, so "program text and behavior are independent" [Samuelson et al., 1994, 2315]

6.   Third, programs are, in fact, machines (entities that bring about useful results, i.e., behavior) that have been constructed in the medium of text (source and object code). The engineering designs embodied in programs could as easily be implemented in hardware as in software, and the user would be unable to distinguish between the two. [Samuelson et al., 1994, 2316]

But programs, even if engraved on a CD-ROM rather than merely inked on paper, can't be executed without some kind of interpreting (and behaving (e.g., printing) machine. So, it's not really the program by itself that can "bring about useful results", but the combination of program and intepreting/behaving—in short, executing—device.

The second point in this quote about hardware and software, is, of course, the point about the relation of a Turing machine to a universal Turing machine. (Could there be a "Turing test" imitation game to disguish between these?!) I think that this is a separate issue from the question about whether a program is a machine.

This is clarified a bit later:

> Not only do consumers not value the text that programmers write (source code), they do not value any form of program text, not even object code. Computer science has long observed that software and hardware are interchangeable: Any behavior that can be accomplished with one can alsob e accomplished with the other. In principle, then, it would be possible for a puchaser of, say, a word processor to get a piece of electronic hardware that plugged into her computer, instead of a disk containing object code. When using that word processor, the purchaser would be unable to determine whether she was using the software or the hardware implementation. [Samuelson et al., 1994, 2319]

Right! That's the point about universal Turing machines.

Somewhere, Samuelson et al. remark that no one can read the object code on a CD-ROM, which marks a difference between software and, say, the script of a play. But that may be the wrong analogy. What about a moved on a DVD? We can't read the DVD any more than we can read a CD-ROM (in both cases, we need a machine to interpret them, to convert them to a humanly perceivable

form). Or even a movie on film: We don't typically look at the individual frames; we need them projected at a certain speed.

7. On programs as machines, [Samuelson et al., 1994, 2320] go on to say:

> Physical machines... produce a variety of useful behaviors, and so it is with programs.

And then they add, in a footnote:

> Traditional texts may tell humans how to do a task; they do not, however, perform the task.

But the same is true for a program, whether written in ink on paper, as source code entered into a computer, or as object code after compilation: It still takes a "physical machine" to do what the program tells it to do. The physical machine translates or associates parts of the program text (however inscribed) with "switch settings" in the computer. Once those switches are set in accordance with the program, the computer—the physical machine—will perform the task.

8.   All this stands in sharp contrast to traditional literary works which are valued because of their expression.... Programs have almost no value to users as texts. Rather, their value lies in behavior. [Samuelson et al., 1994, 2319]

Yes, but this is a misleading analogy. A better analogy is not between, say, a program and a novel, but between a program and musical score or a play script, and between a program's behavior and a performance of that music or play. Only musicians and literary or theatrical people would be interested in the score or the script (just as only programmers would be interested in a program text). Audiences are more interested in the performance (in all cases).

(There is one major difference between programs/behavior on the one hand and scores-scripts/performances on the other: The very same play or score will be performed (interpreted) very differently by different performers. But the very same program should be executed identically by different computers. That, however, may have more to do with the detail of the instructions than with the nature of the interpreters. For more on this, see the discussion about recipes in §10.5.2.)

9.   While conceiving of programs as texts is not incorrect, it is seriously incomplete. A crucially important characteristic of programs is that they behave; programs exist to make computers perform tasks. [Samuelson et al., 1994, 2316]

I agree with the first sentence. For reasons cited above, I disagree with the claim that the program *itself* behaves. But I agree with the final claim: It is the *computer* that behaves, that does something, that "performs a task". And it does so in virtue of its program. But the program *by itself* is static, not dynamic.

What tasks to programmed computers perform? A Turing machine can print, erase, and move; and it can do anything else that can be decomposed into those primitive actions. More sophisticated computers can, of course, do many other things, though, at bottom, all they're doing are forms of "printing", "erasing", and "moving" (such as "printing" certain pixels on a screen)

10. Behavior is not a secondary by-product of a program, but rather an essential part of what programs are. To put the point starkly: No one would want to buy a program that did not behave, i.e., that did nothing, no matter how elegant the source code "prose" expressing that nothing. [Samuelson et al., 1994, 2317]

I suspect that Samuelson et al. are using 'program' not to refer to text but to a product intended to be installed in a computer and executed. For mathematically speaking, there can certainly be programs that have no output (but see §7.5.

This becomes clearer later, when they point out that:

People pay substantial sums of money for a program not because they have any intrinsic interest in what its text says, but because they value what it does… (its behavior). The primary proof that consumers buy behavior, rather than text, is that in acquiring a program, they almost never get a readable instance of the program text. [Samuelson et al., 1994, 2318]

11. Hence, every sensible program behaves. This is true even though the program has neither a user interface nor any other behavior evident to the user. [Samuelson et al., 1994, 2317]

In a footnote, they define "user interface" as "the means by which the user and the program interact to produce the behavior that will accomplish the user's tasks."

A "sensible" program (by which I will now understand the program-as-product that I described above) with no user interface might be akin to a Turing machine that simply computes with no input other than what is pre-printed on its tape. Having a user interface would seem to turn a "sensible" program into an Wegnerian "interactive" program or an *o*-machine.

12. Two programs with different texts [such as Microsoft's Excel and Apple's Numbers, to use a contemporary example] can have completely equivalent behavior. A second comer can develop a program having idential behavior, but completely different text through… "black box" testing. This involves having a programmer run the program through a variety of situations, with different inputs, making careful motes about its behavior. A second programmer can use this description to develop a new program that produces the specified behavior (i.e., functionally identical to the firsomething program) without having access to the text of the first program, let alone copying anything from it. A skilled programmer can, in other

> words, copy the behavior of a program exactly, without appropriating
> *any* of its text. [Samuelson et al., 1994, 2317-2018]

Let's grant that Excel and Numbers don't have exactly the same input-output behavior or even look-and-feel. But black-box testing could, indeed—in theory, at least—allow two completely textually different programs to have exactly the same input-output behavior (and look-and-feel), for the simple, mathematical reason that there can be more than one algorithm for implementing a function.

Note, however, that, in practice, such black-box testing is really akin to the kind of inductive inference that underlies trial-and-error machines: Unless the second programmer has a complete input-output specification, that programmer is limited to reconstructing potentially infinite behavior on the basis of a finite number of examples.

13.   The independence of text and behavior is one important respect in
        which programs differ from other copyrighted works. Imagine trying
        to create two pieces of music that have different notes, but that
        sound indistinguishable. Imagine trying to create two plays with
        different dialogue and characters, but that appear indistinguishable
        to the audience. Yet, two programs with different texts can be
        indistinguishable to users. [Samuelson et al., 1994, 2318]

This is interesting!   If true, it suggests that my analogies between Abstraction/Implementation, score/performance, script/play, etc. (see §14.4, might not be as close as I would want. But is it true? First, consider a play and its script. Suppose that I write a version of *Hamlet* that incorporates all of Shakespeare's lines but adds material that is specifically designed not to be uttered during a performance (perhaps extra characters who are always offstage and silent). That would seem to be a counterexample to Samuelson et al. Second, consider two different algorithms that both compute GCDs. Is it not conceivable that, when the operations in each are decomposed to the primitive operations of a Turing machine or the basic recursive functions, the algorithms would be identical? Or consider a musical score for, say, a four-part choral work. I could produce two different written scores simply by re-ordering the parts (or, by writing them in different musical notations). Yet their performances would not differ.

14. All of my philosophical quibbles notwithstanding, [Samuelson et al., 1994, 2323] do make an interesting point about the appropriateness of copyright for legal protection of programs:

> Program text is, thus, like steel and plastic, a medium in which other
> works can be created. A device built in the medium of steel or plastic,
> if sufficiently novel, is patentable; an original sculpture built of steel
> or plastic is copyrightable. In these cases, we understand quite well
> that the medium in which the work is made does not determine the
> character of the creation. The same principle applies to software. The
> legal character of a work created in the medium of software should

no more be determined by the medium in which it was created than
would be a work made of steel or plastic. In this respect, it makes
no more sense to talk about copyrighting programs than to talk about
copyrighting plastic or steel; it confuses the *medium of creation* and
the *artifact* created. In the case of software, the artifact created is
some form of innovative behavior, whether utilitarian or fanciful.

This suggests that patent is more appropriate than copyright. But what about a
program whose sole purpose is to produce an image on a screen, like a painting
or a film? Wouldn't copyright be more appropriate then? Still, the point remains
that it's the behavior of the program (when executed by a machine! that is more
important than the text.

15.   We suggest that programs should be viewed as virtual machines
        and that this has interesting consequences for the proper form of
        protection. [Samuelson et al., 1994, 2324]

This is a newly introduced term that they will have to go on to explain. But
its introduction is evidence that the authors have been ignoring the CPU that
inteprets and executes the program. For our definition of 'virtual machine' and
more comments on this viewpoint, see 1 above.

16. Summary of [Samuelson et al., 1994]:

        The predominantly functional nature of program behavior and other
        industrial design aspects of programs precludes copyright protextion,
        while the incremental nature of innovation in software largely
        precludes patent protection. (p. 2333)

The argument about patents is this [Samuelson et al., 1994, 2346]:

  (a) "Patent law requires an inventive advance ofver the prior art"

  (b) But the innovations in "functional program behavior, user interfaces, and
      the industrial design of programs... are typically of an incremental sort."

A similar argument applies to chip design; as a result, "Congress passed the
Semiconductor Chip Protection Act of 1984". The point is that a form of
legal protection other than patent and copyright was needed, recognized, and
implemented.

Although this is their principal argument, they also offer a separate argument
against patenting software [Samuelson et al., 1994, 2345]:

  (a) Patents are given for "methods of achieving results",

  (b) Patents are not given "for results themselves".

  (c) "It is... possible to produce functionally indistinguishable program
      behaviors through use of more than one method."

  (d) ∴ a patent could be given for one method of producing a result, but that
      would not "prevent the use of another method".

(e) If it is the result, not the method, that is the "principal source of value" of a program, then the patent on the one method would not protect the result produced by the other method

Their argument against the appropriateness of copyright law boils down to this [Samuelson et al., 1994, 2350]:

(a) "computer programs are machines whose medium of construction is text"

(b) "Copyright law does not protect the behavior of physical machines (nor their internal construction)"

That is, "program behavior...is unpretectable by copyright law on account of its functionality" (p. 2351).

17. One point to make clear early in this chapter is that I'm concerned more with the ontology of software than I am with legal or economic issues in themselves.

18. On Koepsell 1995:

(a) The issue of what software is and whether it should be copyrightable or patentable is investigated by Koepsell 1995 under the heading of "the ontology of cyberspace". Recall from §2.8 that ontology (as a branch of philosophy) deals with issues about the nature and properties of things that exist (or even things that don't exist). So, the ontology of cyberspace asks questions like these:

> What is cyberspace? Is it or does it have dimension? Are there things in cyberspace? Are things in cyberspace properly called objects? Are such objects or is cyberspace itself substance(s) or process(es)? Is cyberspace or the objects in it real or ideal? What is the categorical scheme of cyberspace? How should cyberspace fit into a broader categorical scheme? (p. 2)

(b) INCORPORATE MY "COMMENTS" (Rapaport 1995 from Koepsell & Rapaport 1995) INTO THIS CHAPTER IN APPROPRIATE PLACES.

19. Letter to Vicky Traube:

> I think you might be in a perfect position to answer what I hope is a relatively simple question.
>
> During my retirement, I've been working on a textbook on the philosophy of computer science, and am currently working on a chapter on the nature of software (computer programs) and whether they can or should be copyrighted or patented.
>
> Some people argue that programs are like the scripts of plays; because plays can be copyrighted (and not patented), so should programs. Others argue that programs are more like machines; because machines can be patented (but not copyrighted), so should programs. Finally, others argue that programs are unlike play scripts

(because it's not just the text that matters—it's what they can do—and, anyway, most people can't read the "text" of a program, as they can for a play) *and* that they are unlike machines (because a computer without a program can't do anything); hence, there should be some new kind of legal protection mechanism.

One more piece of background before I get to my question: Presumably, if I go to a bookstore and buy a copy of the script for, say, *Oklahoma*, I'm not liable for copyright infringement if I read it, or even if I act it out with some friends (as long as I don't charge admission, right?). Similarly, if I buy a DVD of the movie version, I'm not liable for copyright infringement if I watch it, even if I watch it with some friends (again, as long as I don't charge admission?). But, if I decide to mount a production of it for my local little theater group, I'd have to get permission (from you?).

Now, my question: What is the legal basis for that permission requirement? Is it the copyright law? Or is it some other legal protection mechanism?

Thanks for any clarification you can provide!

And her reply:

Terrific question. Two things. First of all, under the copyright law, permission is required and royalties may be charged for a public performance even if no admission is charged. If you read a play with friends in your living room, that is probably not public, but if you do it anywhere else and people come to see it, even just people you know, with no admission charge, I consider that a public performance, requiring authors' permission. The legal basis for the permission requirement is in the Copyright Act (Section 106), which gives the Copyright Owner the exclusive right to "do and authorize any of the following: . . . (4) in the case of literary, musical, dramatic and choreographic works, pantomines, and motion pictures and other audiovisual works, to perform the copyrighted work publicly."

Section 117 relates to Limitations on exclusive rights; Computer programs.

I hope this is helpful.

§117 is online at http://www.law.cornell.edu/uscode/text/17/117.

20. Abstract ideas are not patentable subject matter. In order to prevent abstract ideas from being indirectly patented, there are exceptions in the law that will not allow patenting an "invention" made of printed text based on the content of the text. The implication of Gödel. . . [numbering] is that the symbols that represent an abstract idea need not be printed text. The symbols can be something abstract like exponents of prime numbers. They can be something physical

> like electromagnetic signals or photonic signals. ... Does the law exempt abstract ideas from patenting only when they are written in text or does it exempt all physical representations? [PolR, 2009, §"Symbols and Reducibility"]

This seems to be another version of the paradox that this chapter opened with. Here's another:

> Now consider the Church-Turing thesis that computer programs are the same thing as recursive functions. ... Would this be of consequence to the patentability of software? Developing a computer program that fulfills a specific purpose is mathematically the same thing as finding a number that has some specific properties. Can you patent the discovery of a number?
>
> Another way to look at it is that there is a list of all the possible programs. We can make one by listing all possible programs in alphabetic order. Any program that one may write is sitting in this list waiting to be written. In such a case, could a program be considered an invention in the sense of patent law? Or is it a discovery because the program is a preexisting mathematical abstraction that is merely discovered?
>
> This idea that programs are enumerable numbers is supported by the hardware architecture of computers. Programs are bits in memory. Strings of 0s and 1s are numbers written in binary notations. Every program reads as a number when the bits are read in this manner. Programming a computer amounts to discovering a number that suits the programmer's purpose. [PolR, 2009, §"Enumerability"]

21. Related to an issue that [Samuelson et al., 1994] raised (see above, somewhere), [PolR, 2009, §"A Universal Algorithm"] notes that if

> I am sued for infringement on a software patent[,] I may try defending myself arguing I did not implement the method that is covered by the patent. I implemented the lambda-calculus algorithm which is not covered by the patent. And even if it did, lambda-calculus is old. There is prior art. What happens to software patents then?
>
> Anyone that argues in favor of method patents on software must be aware that the current state of technology permits this possibility.

22. On [Samuelson et al., 1994]'s suggestion that it is the virtual machine that should be legally protected (whether by copyright, patent, or something sui generis), compare this:

> For example consider this sentence from In re Alappat:[4]

---

[4]See http://digital-law-online.info/cases/31PQ2D1545.htm

> We have held that such programming creates a new
> machine, because a general purpose computer in effect
> becomes a special purpose computer once it is programmed
> to perform particular functions pursuant to instructions from
> program software.

In a single sentence the court tosses out the fundamental principle
that makes it possible to build and sell digital computers. You don't
need to create a new machine every time you perform a different
computation; a single machine has the capability to perform all
computations. This is what universal Turing machines are doing.

But the "new machine" mentioned in the ruling is a virtual machine.

23. It's one thing to wonder whether software in the form of a computer program
is legally protected (whether by copyright, patent, or something else). But what
about the programming language that the program is written in? On the one
hand, a programming language can itself be viewed as a form of software (it's
surely not hardware). On the other hand, while an author of a novel written in
English can copyright that novel, English itself cannot be copyrighted. (But what
about "created" languages, like Esperanto or Klingon?)

In a 2012 legal case, Oracle sued Google for Google's use of the Java
programming language, which Oracle claimed to own the copyright on. Google's
defense was that programming languages are not copyrightable. The first
decision was in favor of Google, but, on appeal, Oracle seems to have one;
Google is appealing that decision.

Unfortunately, the issues aren't as clear cut as they might be, because it's
not so much the Java language that is in question, as its "API"s (application
programming interfaces), which may, or may not, be considered to be part of
the language itself (as opposed to software written in the language). And, to
further the complications, Google apparently used an "open-source" (that is, not
copyright-protected) version of the Java APIs. (The original briefs by Oracle and
by Google can be found at [Oracle America, 2012] and [Google, 2012]. Good
overviews are [Macari, 2012], [McAllister, 2012], and [McSherry, 2014]; see
also the *Wikipedia* article at
[http://en.wikipedia.org/wiki/Oracle_v._Google].)

13.8   Further Sources of Information

1. Books and Essays:

   - **Bender, David (1985-1986), "Protection of Computer Programs: The Copyright/Trade Secret Interface",** *University of Pittsburgh Law Review* **47: 907–958.**

   - Boyle, James (2009), "What Intellectual Property Law Should Learn from Software", *Communications of the ACM* 52(9) (September): 71–76.

   - Crichton, Michael (2006), "This Essay Breaks the Law", *New York Times* (19 March): WK13.

     – Accessed 24 September 2014 from:
       http://www.nytimes.com/2006/03/19/opinion/19crichton.html
     – Written in a sarcastic manner by a well-known novelist, this essay presents "A case for reforming the way we grant patents", on the grounds that natural laws should not be patentable. For a follow-up article, see "LabCorp. v. Metabolite", accessed 24 September 2014 from:
       http://www.oyez.org/cases/2000-2009/2005/2005_04_607

   - Editorial (2009, 8 November), "Quick, Patent It!", *New York Times*

     – Accessed 9 November 2012 from:
       http://www.nytimes.com/2009/11/08/opinion/08sun3.html

   - Forester, Tom; & Morrison, Perry (1994), *Computer Ethics: Cautionary Tales and Ethical Dilemmas in Computing; Second Edition* (Cambridge, MA: MIT Press).

     – Three sections on pp. 57–68 from Ch. 3 survey the copyright-patent issues in software.

   - Johnson, Deborah G. (2001), "Property Rights in Computer Software", Ch. 6, pp. 137–167, of her *Computer Ethics* (Upper Saddle River, NJ: Prentice Hall).

     – A textbook discussion of copyright, patent, and trade-secret issues surrounding software.

   - Klemens, Ben (2006), "Private Numbers", *Chronicle [of Higher Education] Review* 52(22) (3 February): B2

     – Excerpted from:
       Klemens, Ben (2006), *Math You Can't Use: Patents, Copyright, and Software* (Brookings Institution Press)
     – Accessed 9 November 2012 from:
       http://chronicle.com/article/Collective-Longing-Private/4987/
     – The Klemens passage begins about halfway down the page, beginning with the sentence, "An entirely new economic arrangement has appeared in mathematics and its offspring, computer science."

   - Menand, Louis (2014), "Crooner in Rights Spat: Are Copyrifght Laws Too Strict?", *New Yorker* (20 October): 84–89.

     – Accessed 5 November 2014 from:
       http://www.newyorker.com/magazine/2014/10/20/crooner-rights-spat
     – A survey of the history of copyright laws, especially as they apply to works of art.

- Petroski, Henry (2008), "Twists, Tags and Ties", *American Scientist* 96(3) (May-June): 188–192.

  – On the role of implementation: "As with any well-written patent, in the one granted to McHenry he is careful not to restrict his invention to only one 'embodiment'..." (p. 192).

- **Samuelson, Pamela (1988), "Is Copyright Law Steering the Right Course?", *IEEE Software* (September): 78–86.**

  – An excellent overview of the copyright-vs.-patent dispute, arguing in favor of patent protection for software.

- Samuelson, Pamela (1991), "Digital Media and the Law", *Communications of the ACM* 34(10) (October): 23–28.

  – "Six characteristics of digital media seem likely to bring about significant changes in the law": ease of replication, ease of transmission and multiple use, plasticity (related to Moor's §12.3.1 notion of software's changeability), equivalence of works in digital form (referring to the fact that, when digitized, different kinds of copyrightable works all get implemented in the same medium), compactness (digitized works take up little space, hence are more prone to theft), and nonlinearity (due to the availability of hyperlinks, for instance).

- Samuelson, Pamela (2003), "Unsolicited Communications as Trespass?", *Communications of the ACM* 46(10) (October): 15–20.

  – Accessed 9 November 2012 from:
  http://people.ischool.berkeley.edu/∼pam/papers/acm_vol46_p15.pdf

  – Abstract: "Attempting to stretch existing laws to address previously unforeseen technological issues."

- Samuelson, Pamela (2007), "Software Patents and the Metaphysics of Section 271(f)", *Communications of the ACM* 50(6) (June): 15–19.

  – Accessed 24 September 2014 from:
  https://www.law.berkeley.edu/php-programs/faculty/facultyPubsPDF.php?facID=346&pubID=187

  – "Microsoft argues that neither the intangible sequence of ones and zeros of the object code, nor the master disks onto which the ogject code has been loaded, should be considered a component of a patented invention.... Only when object code has actually been installed on a...computer does it become a physical component of a physica device..." (pp.15–16).

- Samuelson, Pamela (2007), "Does Copyright Law Need to Be Reformed?", *Communications of the ACM* 50(10) (October): 19–23.

- Samuelson, Pamela (2008), "Revisiting Patentable Subject Matter", *Communications of the ACM* 51(7) (July): 20–22.

  – "Is everything under the sun made by humans patentable subject matter?"

- Samuelson, Pamela (2013), "Is Software Patentable?", *Communications of the ACM* 56(11) (November): 23–25.

  – Accessed 26 September 2014 from:
  https://www.law.berkeley.edu/php-programs/faculty/facultyPubsPDF.php?facID=346&pubID=257

- Samuelson, Pamela (2015), "Copyrightability of Java APIs Revisited", *Communications of the ACM* 58(3) (March): 22–24, http://radar.oreilly.com/2014/11/copyrightability-of-java-apis-revisited.html (accessed 24 March 2015).

- Shaw, Alex; Li, Vivian; & Olson, Kristina R. (2012), "Children Apply Principles of Physical Ownership to Ideas", *Cognitive Science* 36(8) (November-December): 1383–1403.

  – Accessed 26 September 2014 from:
    https://depts.washington.edu/uwkids/Shaw.Li.Olson.2012.pdf
  – "Adults apply ownership not only to objects but also to ideas. But do people come to apply principles of ownership to ideas because of being taught about intellectual property and copyrights? ...like adults, children as young as 6 years old apply rules from ownership not only to objects but to ideas as well."

- Sprigman, Christopher Jon (2015), "*Oracle v. Google:* A High-Stakes Legal Fight for the Software Industry", *Communications of the ACM* 58(5) (May): 27–29.

- Thatcher, Matt E.; & Pingry, David E. (2007), "Software Patents: The Good, the Bad, and the Messy", *Communications of the ACM* 50(10) (October): 47–52.

  – For commentary, see:
    Oldehoeft, Rod; Montague, John; Thatcher, Matt E.; & Pingry, David E. (2007), "Patented Algorithms Are Bad, Copyrighted Software Is Good" (letters to the editor), *Communications of the ACM* 50(12) (December): 9–10.

2. Websites:

- "Time to Abolish Software Patents?", *Slashdot* (29 February 2008), http://yro.slashdot.org/story/08/02/29/0344258/time-to-abolish-software-patents (accessed 9 November 2012).

  – A blog, with some useful links.

- Edwards, David A., "Platonism Is the Law of the Land" (earlier title: "Why Shouldn't Laws of Nature Be Patentable?"), accessed 8 September 2014 from: http://www.math.uga.edu/∼davide/Platonism_is_the_Law_of_the_Land.pdf

- PolR (2009), "An Explanation of Computation Theory for Lawyers", *Groklaw* (11 November), http://www.groklaw.net/article.php?story=20091111151305785, http://www.groklaw.net/staticpages/index.php?page=20091110152507492, and http://www.groklaw.net/pdf2/ComputationalTheoryforLawyers.pdf (accessed 16 December 2013).

  – "Computers don't work the way some legal documents and court precedents say they do."
  – "The phrase 'effective method' is a term of art [in mathematics and philosophy]. This term has nothing to do with the legal meaning of 'effective' and 'method'. The fact that these two words also have a meaning in patent law is a coincidence."

- Sloman, Aaron (2008), "Why Virtual Machines Really Matter—for Several Disciplines", lecture slides accessed 27 October 2014 from: http://www.cs.bham.ac.uk/research/projects/cogaff/talks/information.pdf

– Views virtual machines as mathematical abstractions that can have causal relations with the physical world.

- Touretzky, David (2008), "Gallery of CSS Descramblers",
  http://www.cs.cmu.edu/∼dst/DeCSS/Gallery/
  (accessed 9 November 2012).

  – "If code that can be directly compiled and executed may be suppressed under the DMCA [[Digital Millenium Copyright Act], as Judge [Lewis A.] Kaplan asserts in his preliminary ruling, but a textual description of the same algorithm may not be suppressed, then where exactly should the line be drawn? This web site was created to explore this issue, and point out the absurdity of Judge Kaplan's position that source code can be legally differentiated from other forms of written expression." For example, it used to offer a copy of otherwise suppressed code on a T-shirt!

FUNCTION (I/O PAIRS)

|implemented by

serial alg 1   serial alg 2   parallel alg 3

serial alg 1   parallel alg 3
expressed in   expressed in
parallel   serial
proglang 1   proglang 3

serial alg 1   parallel alg 3
expressed in   expressed in
serial   parallel
proglang 2   proglang 4

virtual   actual   serial alg 1   ser.alg 1   parallel   parallel
machine: computer   in serial   in serial   alg 3 in   alg 3 in
theoretical   w/ serial   proglang 2   proglang 2   proglang 4   proglang 4
model of   proglang 2   compiled   compiled   compiled   compiled
computation   as its   into serial   into   into   into
for serial   machine   mach.lang 1   serial   serial   parallel
proglang 2   lang   mach.lang 2   mach.lang 5   mach.lang 6

                                           ser.alg 1
                                           in serial
                                           proglang 2
                                           compiled
                                           into
                                           parallel
                                           mach.lang 3

-- -- -- ETC. -- -- --

WHICH (IF ANY) OF THESE SHOULD BE COPYRIGHTABLE?

WHICH (IF ANY) OF THESE SHOULD BE PATENTABLE?

Figure 13.3:

Chapter 14

What Is Implementation?

"I wish to God these calculations had been executed by steam!"
—Charles Babbage (1821), quoted in [Swade, 1993, 86].

What... [Howard H. Aiken, who built the Harvard Mark I computer] had in mind... was the construction of an electromechanical machine, but the plan he outlined was not restricted to any specific type of mechanism; it embraced a broad coordination of components that could be resolved by various constructive mediums. [Chase, 1980, 199, 226]

[W]hy wasn't Mark I an electronic device? Again, the answer is money. It was going to take a lot of money. Thousands and thousands of parts! It was very clear that this thing could be done with electronic parts, too, using the techniques of the digital counters that had been made with vacuum tubes, just a few years before I started, for counting cosmic rays. But what it comes down to is this: if [the] Monroe [Calculating Machine Co.] had decided to pay the bill, this thing would have been made out of mechanical parts. If RCA had been interested, it might have been electronic. And it was made out of tabulating machine parts because IBM was willing to pay the bill. —Aiken, quoted in [Chase, 1980, 200]

Figure 14.1: ©2015, Scott Adams Inc./Universal UClick

14.1 Recommended Readings:

1. Required:

   - Chalmers, David J. (1993), "A Computational Foundation for the Study of Cognition", *Journal of Cognitive Science* (South Korea) 12(4) (October-December 2011): 323–357.

     (a) Accessed 3 November 2014 from:
        http://j-cs.org/gnuboard/bbs/board.php?bo_table=__vol012i4&wr_id=2

     (b) The 2011 version was accompanied by commentaries (including [Egan, 2012], [Rescorla, 2012b], [Scheutz, 2012], and [Shagrir, 2012]) and a reply by Chalmers [Chalmers, 2012].

     (c) Read §§1–2.
        – §2 of this paper was published (in slightly different form) as "On Implementing a Computation", *Minds and Machines* 4 (1994): 391–402.

     (d) Skim the rest.

   - Rapaport, William J. (1999), "Implementation Is Semantic Interpretation", *The Monist* 82(1): 109–130.

     – This article was based on a longer essay (which it alludes to as "in preparation"), which has since been published as:
        Rapaport, William J. (2005), "Implementation Is Semantic Interpretation: Further Thoughts", *Journal of Experimental and Theoretical Artificial Intelligence* 17(4) (December): 385–417.

2. Very Strongly Recommended:

   - Putnam, Hilary (1988), Appendix, *Representation and Reality* (Cambridge, MA: MIT Press): 121–125.

     (a) "*Theorem.* Every ordinary open system is a realization of every abstract finite automaton."

     (b) Putnam's argument for this "theorem" is similar to [Searle, 1990]'s argument about the wall behind me implementing Wordstar, but it is much more detailed.

   - Chalmers, David J. (1993), "Does a Rock Implement Every Finite-State Automaton?", *Synthese* 108 (1996): 309–333.

     – Accessed 9 November 2012 from http://consc.net/papers/rock.html
     – This is a reply to Putnam's argument, above.
     – Chalmers corrects "an error in my arguments" in this paper in [Chalmers, 2012, 236–238].

3. Strongly Recommended:

   (a) Suber, Peter (1997), "Formal Systems and Machines: An Isomorphism"

     - Accessed 9 November 2012 from:
        http://www.earlham.edu/∼peters/courses/logsys/machines.htm

   (b) Suber, Peter (2002), "Sample Formal System S"

     - Accessed 9 November 2012 from:
        http://www.earlham.edu/∼peters/courses/logsys/sys-xmpl.htm

14.2 Introduction

One concept that has repeatedly cropped up in our discussions is that of "implementation":

- In Chapter 3 (§3.8), we saw that [Denning et al., 1989, 12] said that computer science studies (among other things) the "implementation" of algorithmic processes.

- In Chapter 5 (§5.5), we read that [Loui, 1987] said that computer science is, among other things, the "software- and hardware-implementation of algorithms".

- In Chapter 9 (§9.3.5), we saw [Searle, 1990] saying that "the wall behind my back is right now implementing the Wordstar program", and we talked about "physical implementations of Turing machines" and "human cognition...implemented by neuron firings".

- In Chapter 10 (§10.5.2), we discussed "implement[ing] a plan [by] copy[ing] an abstract design into reality".

- In Chapter 12, we talked about programs "implementing" algorithms and the idea of there being "multiple" implementations.

- And in Chapter 13, we looked at how copyright only applies to "implementations" of ideas in language, and we considered the possibility of there being implementations of implementations.

In computer science, we talk about algorithms implementing functions, about computer programs implementing algorithms, about data structures implementing abstract data types, and about computers implementing Turing machines. So, what is an implementation?

In this chapter, we will look at two theories about the nature of implementation. One, due to David Chalmers, was designed to reply to [Searle, 1990]. The other will illuminate the nature of the relation between syntax and semantics.

14.3 Chalmers's Theory of Implementation

David Chalmers's essay, "A Computational Foundation for the Study of Cognition" [Chalmers, 2011] (published in a slightly shorter version as [Chalmers, 1995]; see also [Chalmers, 1996]), concerns both implementation and cognition. In this section, we will focus only on what he has to say about implementation.

One of his claims is that we need a "bridge" between the *abstract theory* of computation and *physical* systems that "implement" them. Other phrases that he mentions as synonyms for 'implement' are: 'realize' (that is, make real) and 'are described by'.

(In all three cases, the entity that is doing the implementing, doing the realizing, or being described is, typically, a physical ("real") thing. And the entity that is being

implemented, or being realized, or doing the describing is, typically, an "abstract" thing
(or a linguistic description).)

That is, we need a *theory* of what implementation is. And one reason we need
such a theory is in order to refute [Searle, 1990]'s claim that "*any* given [physical]
system can be seen to implement *any* computation *if interpreted appropriately*"
[Chalmers, 2011, §I, my italics].

On the first version of Chalmers's theory of implementation, a physical system P
implements a computation C when the "causal structure" of P "mirrors" the *formal
structure* of C.

There are many aspects of this that need clarification:

- What kind of physical system is P?

  (It need not be a computer, according to Chalmers.)

- What kind of computation is C?

  (Is it merely an abstract algorithm? Is it more specifically a Turing-machine
  program? Is it a program being executed—what I called in Ch. 13 a "process"?
  In any case, it would seem to be something that is more or less abstract, which
  raises another question: What does it mean for something to be "abstract"?)

- What does 'when' mean?

  (Is this intended to be just a sufficient condition, or a stronger biconditional
  ("when *and only when*")?)

- What is a "causal structure"?

- What is a "formal structure"?

- And the most important question: What does 'mirror' mean?

So, here is Chalmers's second version, which begins to answer some of these
questions:

P implements *C* if and only if:

1. the physical states of P can be grouped into state-types, and

2. there is a 1–1 map from the formal states of C to the physical state-types of P,
   and

3. the formal state-types (that is, the ones that are related by an abstract state-
   transition relation) map onto the physical state-types (that is, the ones that are
   related by a "corresponding" *causal* state-transition relation).

Figure 14.2 (at the end of this chapter) might help to make this clear.

In this figure, the physical system *P* is represented by a set of dots, each of which represents a physical state of *P*. These dots are partitioned into subsets of dots, that is, subsets containing the physical states. Each subset represents a state-type, that is, a set of states that are all of the same type. That takes care of part 1 of Chalmers's account.

The computation *C* is also represented by a set of dots. Here, each dot represents one of *C*'s formal states. The arrows that point *from* the dots in *C* (that is, from *C*'s formal states) *to* the subsets of dots in *P* (that is, to the state-types in *P*) represent the 1–1 map from *C* to *P*. To make it a 1–1 map, each formal state of *C* must map to a *distinct* physical state-type of *P*.[1] That takes care of part 2 of Chalmers's account.

The arrows in set *C* represent the abstract state-transition relation between *C*'s formal states. And the arrows in set *P* between *P*'s subsets represent the causal state-transition relation between *P*'s state-types. Finally, because *C*'s formal states are mapped *onto P*'s physical state-types, the 1–1 map is a bijection. Chalmers also says that *C*'s *abstract* state-transition relations "correspond" to *P*'s *causal* state-transition relations. I take it that this means that the 1–1 map is a "homomorphism" (that is, a structure-preserving map).[2] Because the map is also "onto", it is an "isomorphism". (An isomorphism is a structure- or "shape"-preserving bijection.) So, *P* and *C* have the same *structure*. That takes care of part 3 of Chalmers's account.

The only other clarifications that we need to make are that a state is "formal" if it's part of the *abstract*—that is, non-physical—computation *C*, and a state is "causal" if it's part of the *physical* system *P*: Here, 'formal' just means 'abstract', and 'causal' just means 'physical'.

We can then say that a physical system (or a physical process?) *P implements* an abstract computation *C* (which might be a Turing machine, or a less-powerful finite-state automaton,[3] or a "combinatorial-state automaton"[4]) if and only if there is a "reliably causal"[5] isomorphism $f : C \rightarrow P$. Such an *f* is a relation between an abstract, computational model and something in the real, physical, causal world. This *f* is 1–1 and onto[6]—a structure-preserving homomorphism such that the abstract, input-output and processing relations in *C* correspond to reliably causal processes in *P*.

It follows from this analysis that:

- *Every* physical system implements *some* computation. That is, for every physical system *P*, there is some computation *C* such that *P* implements *C*.

- But *not* every physical system implements any given computation. That is, it is

---

[1] More precisely, a function from set *A* to set *B* is 1–1 $=_{def}$ for any two members a, a' of *A*, and for any member *b* of *B*, if $f(a) = f(a')$, then $a = a'$. In other words, a function is 1–1 if, whenever two of its outputs are the same, then their inputs must have been the same. Yet another way to put it is this: A function is 1–1 if no two (distinct) objects in its domain have the same image in its range.

[2] *f* is a homomorphism $=_{def} f(R(c_1, \ldots, c_n)) = f(R)(f(c_1), \ldots, f(c_n))$. That is, if the c_i are related by some relation *R*, and if that relationship is mapped by *f*, then the image of $(R(c_1, \ldots, c_n)$ will be the image of *R* applied to the images of the c_i. That's what it means to preserve structure.

[3] EXPLAIN WHAT THIS IS AND WHY IT'S LESS POWERFUL

[4] EXPLAIN

[5] EXPLAIN

[6] Really? Even if *C* is abstract? Typically, if *C* is abstract and *P* is physical, *P* will have more "parts" (will be more detailed) than *C*. But if *f* is a bijection, then every part of *P* must map to some part of *C*, which seems to violate the abstract-physical distinction. Perhaps, however, the parts of *P* that are due merely to its being physical (that is, the implementation-dependent details) can be ignored.

not the case that, for every P and for every C, P implements C. That is, there is some P and there is some C such that P does *not* implement C. For example, it is highly unlikely that the wall behind me implements Wordstar, because the computation is too complex.

- A given physical system *can* implement more than one computation. That is, for any P, there might be two different computations $C_1 \neq C_2$ such that P implements C_1 and P implements C_2. For example, my computer, right this minute as I type this, is implementing the VI text-processing program, a clock, Powerpoint, and several other computer programs.

Consider the first of these consequences—namely, that *every P* implements *some C*. Does that make the notion of computation vacuous? No, because the fact that some P implements some C is not necessarily *the reason why P* is a P. (But, in the case of cognition, it might be!)

14.4   Implementation as Semantic Interpretation

According to Chalmers, implementation is an isomorphism between an abstract specification and a concrete, physical process. But computer scientists also use the term 'implementation' to refer to the relation between one abstract data type (for example, a stack) and its "implementation" or "representation" by *another* abstract data type (for example, a linked list).

Let me a bit more precise. Some programming languages, such as Lisp, do not have stacks as a built-in data structure. So, a programmer who wants to write a program that requires the use of stacks must find a substitute. In Lisp, whose only built-in data structure is a linked list, the stack would have to be built out of a linked list: Stacks in Lisp can be implemented by linked lists. Here's how:

First, a stack is a particular kind of data structure, often thought of as consisting of a set of items structured like a stack of cafeteria trays: New items are added to the stack by "pushing" them on "top", and items can be removed from the stack only by "popping" them from the top. Thus, to define a stack, one needs (i) a way of referring to its top and (ii) operations for pushing new items onto the top and for popping items off the top. That, more or less (mostly less, since this is informal), is a stack defined as an ADT.

Second, a linked list ('list', for short) is itself an abstract data type. It is a sequence of items whose three basic operations are:

1. *first*(l), which returns the first element on the list l,

2. *rest*(l), which returns a list consisting of all the original items except the first, and

3. *make-list*(i, l) (or *cons*(i, l)), which recursively constructs a list by putting item i at the beginning of list l.

Finally, a stack *s* can be implemented as a list *l*, where $top(s) := first(l)$, $push(s,i) := make\text{-}list(l,i)$, and $pop(s)$ returns $top(s)$ and redefines the list to be $rest(l)$.

The main point is that not all examples of implementation concern the implementation of something abstract by something concrete. As we have just seen, sometimes one abstract thing can implement another abstract thing. What we need is a more general notion, namely, semantic interpretation.

So, what is semantic interpretation? Consider a "formal system" (also sometimes called a "symbol system", or a "theory"—understood as a set of sentences—or a "formal language". Such a system consists of:

1. primitive (or atomic) symbols (sometimes called "tokens" or "markers", to be thought of as the playing pieces in a board game like Monopoly)

   (a) These are assumed to have no interpretation, no meaning, hence the term 'marker'. The racecar token in Monopoly isn't interpreted as a racecar in the game; it's just a token that happens to be racecar shaped, so as to distinguish it from the token that is top-hat shaped. (And the top-hat token isn't interpreted as a top hat in the game: Even if you think that it makes sense for a racecar to travel around the Monopoly board, it makes no sense for a top hat to do so!)

   (b) Examples of such atomic symbols include the letters of an alphabet, or (some of) the vocabulary of a language, or (possibly) neuron firings, or even states of a computation.

2. (recursive) rules for forming new (complex, or molecular) symbols, sometimes called 'well-formed formulas' (wffs)—that is, grammatically correct formulas— from "old" symbols (that is, from previously formed symbols), beginning with the atomic symbols as the basic "building blocks".

   (a) These rules might be spelling rules (if the atomic symbols are alphabet letters), or grammar rules (if the atomic symbols are words), or bundles of synchronous neuron firings (if the atomic symbols are single neuron firings).

   (b) If the molecular symbols are "linear" (1-dimensional), then they can be thought of as "strings" (that is, sequences of atomic symbols), or words (if the atomic symbols are letters), or sentences (if the atomic symbols are words). If the molecular symbols are $\geq$ 2-dimensional, then they are something called 'complexes'.

3. a "distinguished" (that is, singled-out) subset of wffs

   (a) These are usually called 'axioms'.

   (b) But having axioms is optional. If English is considered as a formal system [Montague, 1970], it doesn't need axioms. But if geometry is considered as a formal system, it usually has axioms.

4. recursive rules (called 'rules of inference' or 'transformation rules') for forming ("proving") new wffs (called 'theorems') from old ones (usually, but not always, beginning with the axioms).

The "syntax" of such a system is the study of the *properties of* the symbols of the system and the relations *among* the symbols. Among these relations are the "grammatical" relations, which specify which strings of symbols are well formed (according to the rules of grammar), and the "proof-theoretic" (or "logical") relations, which specify which sequences of wffs are proofs of theorems (that is, which wffs are derivable by means of the rules of inference).

Here is an analogy: Consider a new kind of monster/building-block toy system, consisting of Lego blocks that can be used to construct Transformer monsters. (This wouldn't be a very practical real toy, because things made out of Legos tend to fall apart. That's why this is a thought experiment, not a real one!) The basic Lego blocks are the primitive symbols of this system. Transformer monsters that are built from Legos are the wffs of the system. And the sequences of moves that transform the monsters into trucks are the proofs of theorems.

Real examples of formal systems include propositional logic, first-order logic, Douglas Hofstadter's Mu system [Hofstadter, 1979], Suber's "S",[7] and a system discussed at http://www.cse.buffalo.edu/~rapaport/Papers/book.ch2.pdf. (Parts of that document will be included here as an appendix to this chapter.)

The important fact about a formal system and its syntax is that there is no mention of truth, meaning, reference, or any other "semantic" notion.

The "semantics" of a formal system is the study of the relations *between* the symbols of the system (on the one hand) and something else (on the other hand). The "something else" might be what the symbols "represent", or what they "mean", or what they "refer to", or what they "name", or what they "describe". Or it might be "the world". If the formal system is a language, then semantics studies the relations between, on the one hand, the words and sentences of the language and, on the other hand, their meanings. If the formal system is a theory, then semantics studies the relations between the symbols of the theory and the world—the world that the theory is a theory *of*.

Semantics, in general, requires three things:

1. a syntactic domain; call it 'SYN'—typically, but not necessarily, a formal system,

2. a semantic domain; call it 'SEM'—characterized by an "ontology"

   - An ontology is, roughly, a theory or description of the things in the semantic domain. It can be understood as a (syntactic!) theory of the semantic domain, in the sense that it specifies (a) the parts of the semantic domain (its members, categories, etc.) and (b) their properties and relationships (structural as well as inferential or logical). Such an ontology can be called a "model theory".

---

[7]http://www.earlham.edu/~peters/courses/logsys/sys-xmpl.htm

3. a semantic *interpretation* mapping from SYN to SEM. SEM is a "model" or "interpretation" of SYN; SYN is a "theory" of SEM.

Here are several examples of semantic domains that are "implementations" of syntactic domains:

algorithms	are implemented by	computer programs (in language *L*)
computer programs (in language *L*)	are implemented by	computational processes (on machine *m*)
abstract data types	are implemented by	data structures (in language *L*)
musical scores	are implemented by	performances (by musicians)
play scripts	are implemented by	performances (by actors)
blueprints	are implemented by	buildings (made of wood, bricks, etc.; cf. "The Three Little Pigs")
formal theories	are implemented by	(set-theoretic) models

So,
it looks as if we can say that semantic domains *implement* syntactic domains. That is, implementation is the relation of semantic interpretation.

Typically, implementations are *physical* "realizations" of "abstractions". But physical realizations are a special case, as shown by the example of a linked list that implements a stack, both of which are abstract. So, what we really have is the following relationship:

I is an implementation of abstraction *A* in medium *M*.

Implementation *I* could be either an abstraction itself or something concrete. By an "abstraction" *A*, I have in mind a generalization of the notion of an abstract data type, as long as it is kept in mind that one abstraction can implement another abstraction. And the medium *M* could be either an abstraction or something concrete.

In fact, there could be a sequence of implementations (or a "continuum"; see [Smith, 1987]): a stack can be implemented by a linked list, which, in turn, could be implemented in Pascal, which, in turn, could be implemented (that is, compiled into) some machine language *L*, which, in turn, could be implemented on my Mac computer. But it could also be more than a mere sequence of implementations, because there could be a *tree* of implementations: The very same linked list could be implemented in Java, instead of Pascal, and the Java or Pascal program could be implemented in some other

machine language on some other kind of computer. ([Sloman, 1998, §2, p. 2] makes the same point.)

We see this same phenomenon in another, though related, situation: For an introductory course I once taught, I wrote a very simple Pascal program that added two integers. This program was compiled (that is, implemented) in a very simple assembly language designed for instructional purposes. That assembly language program was written (that is, implemented) in a dialect of Pascal called MacPascal, which was implemented in MacOS assembly language, which was implemented in the machine language that was implemented on a Mac II computer. (A similar situation is described in [Sloman, 1998, §2, p. 2].) An interesting question to think about is this: When two integers are input to my original Pascal program, and their sum is output, "where" does the actual addition take place? Is it my Pascal program that adds the two integers? Or is it "really" the Mac II computer that adds them? Or is it one (or all?) of the intermediate implementations?

Here's something else to think about: Consider a program written in a high-level programming language. Suppose that the program has a data structure called a "student record", containing information about (that is, a representation of) a student (something like the "person record" in the *Bloom County* cartoon at the end of this chapter.) For instance, Lucy's student record might look something like this:

```
(student-record:
 (name Lucy)
 (student-number 12345)
 (major computer-science)
 (gpa 3.99)
)
```

This is merely a piece of syntax: a sequence of symbols. You and I reading it might think that it represents a student named 'Lucy' whose student number is 12345, whose major is computer science, and whose GPA is 3.99. But as far as the computer (program) is concerned, this record might just as well have looked like this:

```
(SR:
 (g123 n456)
 (g124 12345)
 (g125 c569)
 (g126 3.99)
)
```

As long as the program 'knows' how to input new data, how to compute with these data, and how to output the results in a humanly readable form, it really doesn't matter what the data look like *to us*. That is, as long as the relationships among the symbols are well specified, it doesn't matter—as far as computing is concerned—how those symbols are related to other symbols that might be meaningful to us. That is why it is syntax.

Now, there are at least two ways in which this piece of syntax could be implemented. One implementation, of course, is Lucy herself in the real world: A

person named Lucy, with student number 12345, who is majoring in computer science, and who has a GPA of 3.99. This Lucy is an implementation of that data structure; she is also a semantic interpretation of it.

Another implementation is the way in which that data structure is actually represented in the computer's machine language. That is, when the program containing that data structure is compiled into a particular computer's machine language, that data structure will be represented in some other data structure expressed in that machine language. That will actually be another piece of syntax. And that machine-language syntax will be an implementation of our original data structure.

But when that machine-language program is being executed by the computer, some region of the computer's memory will be allocated to that data structure (to the computer's representation of Lucy, if you will), which will probably be an array of '0's and '1's, that is, of bits in memory. These bits will be yet another implementation of the original data structure, as well as an implementation of the machine-language data structure.

Now, here is a question to think about: What is the relation between the human (Lucy herself) and this region of the computer's memory? Does the memory location "simulate" Lucy? (Do bits simulate students?) Does the memory location *implement* Lucy? (Do bits implement students?) Also: The '0's and '1's in memory can be thought of as the '0's and '1's on a Turing-machine tape, and Lucy can be thought of as an interpretation of that Turing-machine tape. Now, recall from §10.5.1 what Cleland said about the difference between Turing-machine programs and mundane procedures: The former can be understood independently of any real-world interpretation (that is, they can be understood purely syntactically, to use the current terminology)—understood in the sense that we can describe the computation in purely '0'/'1' terms. (Can we? Don't we at least have to interpret the '0's and '1's in terms of a machine-language data structure, interpretable in turn in terms of a high-level programming-language data structure, which is interpretable, in turn, in terms of the real-world Lucy?) Mundane procedures, on the other hand, must be understood in terms of the real world (that is, the causal world) that they are manipulating. (This is a good topic for class discussion!)

So, why do I think that implementation is semantic interpretation? The word 'implement' comes from a Latin word meaning "to fill up, to complete", as in filling in the details of an abstraction. And I have suggested the following:

> Let *I* be something abstract or concrete.
> Let *A* be an "abstraction", that is, a generalization of the notion of an abstract data type (as long as it is kept in mind that one abstraction can implement another abstraction).
> And let *M* be any abstract or concrete "medium".
> Then:
>
> > *I is an implementation of A* in medium *M*
> >
> > > means:
> >
> > *I* is a semantic interpretation of syntactic domain *A* in semantic domain *M*.

As another example of an "abstract implementation", consider a top-down-design, stepwise-refinement of a computer program (see §6.4.3): Each level (each refinement) is an abstract implementation of the previous, higher-level one. A "concrete implementation" would be an implementation in a physical medium.

Is implementation "sui generis"? That is, is it something that is not like anything else? Or is it, in fact, something else, such as one of the following:

- individuation:
  the relation between the lowest level of a genus-species tree (such as "dog" or "human") and individual dogs or humans (for example, my cat Bella "individuates" the species *Felis catus*).

- instantiation:
  the relation between a specific instance of something and the kind of thing that it is (for example, the specific redness of my notebook cover is an instance of the color "red").

- exemplification:
  the relation between a (physical) object and a property that it has (for example, Bertrand Russell exemplifies the property of being a philosopher).

- reduction:
  the relation between a higher-level object and the lower-level objects that it is made of (for example, water is reducible to a molecule consisting of two atoms of hydrogen and one atom of oxygen, or, perhaps, the emotion of anger is reducible to a certain combination of neuron firings).

In [Rapaport, 1999] and [Rapaport, 2005b], I argue that each of these may be implementations, but not vice versa, and that all of them are semantic interpretations.

14.5 NOTES FOR NEXT DRAFT

1. On implementation as reduction:
   [Goodman, 1987, 480] argues that the kind of reduction involved in reducing ordered pairs to Wiener-Kuratowski doubletons—and, presumably, in reducing natural numbers to sets of one kind or another—is different from the kind of reduction involved in reducing heat to mean molecular kinetic energy. There are many ways of reducing ordered pairs or numbers to sets; there is only one way to reduce heat to something physical. The former is multiply realizable; the latter is not. Presumably, the reduction of "algorithm" to Turing machines is of the former kind, given the equivalent reductions to recursive functions, etc.

   Moreover, he argues, because knowledge of the non-reduced term is independent of knowledge of the reducing term, such reductions are intensional.

2. On Lego-like objects, see the definition of 'constructive object' in
   http://www.encyclopediaofmath.org/index.php/Constructive_object

3. On multiple realization:

   > The search for a single causative factor—Factor X—is likely to be successful if the event being studied has a fixed form and fixed determinants. But the essence of migraine...lies in the variety of forms it may take and the variety of circumstances in which it may occur. Therefore, though one type of migraine may be associated with Factor X, and another with Factor Y, it seems impossible, on *prima facie* grounds, that all attacks of migraine could have the same aetiology. [Sacks, 1999, 175–176]

   Sacks concludes that, to explain migraine, or any other phenomenon that exhibits such multiple realizability, more than one "universe of discourse" is needed. At the very least, there is, first, what I have called the Abstraction, or the functional–causal-network description; second, what I have called the Implementation, or the anatomical-physiological description; and, third, there might also be what Sacks calls the experiential (p. 176).

   Also, there is evidence that nervous systems are multiply realized even among animals; see [Pennisi, 2013].

   Also, not only can all 16 truth-functional connectives be implemented as (electronic) logic gates, they can also be implemented in bacteria! [Benenson, 2013]

4. The Abstraction-Implementation distinction is mirrored in the "occupant"-"role" distinction made in functionalist theories of the mind ([Lycan, 1990, 77]; on functionalism, see §20.4). Hamlet is a role; Richard Burton (in the 1964 Broadway production of *Hamlet*) occupied that role. Alternatively, we could say that Burton implemented Hamlet in the medium of human being (and a drawing of Hamlet implements Hamlet in the medium of an animated cartoon version of the play).

5. The ideas that Abstractions can implement other Abstractions and that there can be "continua" of implementations is a consequence of what [Lycan, 1990, 78] refers to as the "relativity" of implementation: He wants

> to attack the dichotomies of "software"/"hardware".... ...my objection is that "software"/"hardware" talk encourages the idea of a bipartite Nature...as against reality, which is a multiple *hierarchy* of levels of nature.... See Nature as hierarchically organized in this way, and the "function"/"structure" distinction *goes relative*: something is a role as opposed to an occupant, a functional state [that is, an Abstraction, in my terms] as opposed to a realizer [that is, an Implementation], or vice versa, only *modulo* a designated level of nature.

6. The relationships among a chess game, a Civil War battle, and a computer program whose input-output relationships can be interpreted as either of those can be viewed as an implementation (chess game) that is abstracted (into a computer program), which is then re-implemented as a Civil War battle. "One consequence of... [this] approach is that isomorphic models are equivalent. If there is a one-to-one function from the domain of one model onto the domain of another that preserves the relations of the model, then any sentence of the formal language that is true in one model is true in the other. ...[T]he best a formal theory can do is to fix its interprettion 'up to isomorphism'. Any model of the theory can be changed into another model just by substituting elements of the domain and renaming" [Shapiro, 1996, 158–159]. Here, "models" are just

7. It's surely worth mentioning somewhere that the problem of what implementation is is closely related to the mind-body (or the mind-brain) problem (very roughly, (how) is the mind implemented in the brain?). Also, there is a wide literature in philosophy on "realization" (especially as related to the mind-brain problem), which is only briefly touched on here and in the Further Sources bibliography. For a nice discussion of this, see [Scheutz, 2001, §1]

8. Perhaps another "great insight" (see Ch. 7, §7.6)—and one that Peter Denning would probably applaud (Ch. 7, §35)—is that the first three insights can be implemented in matter. Alan Kay divides this insight into a "triple whammy": The "core of computer science" (Guzdial's phrase)...

> ...is all about the triple whammy of computing.
>
> 1 Matter can be made to remember, discriminate, decide and do
> 2 Matter can remember descriptions and interpret and act on them
> 3 Matte can hold and interpret and act on descriptions that describe anything that matter can do.

He later suggests that the third item is the most "powerful", followed by the first, and then the second. He also suggests that issues about the limits of computability and multiple realizability are implicit in these. [Guzdial and Kay, 2010]

Note that the triple whammy is closely related to Pat Hayes's notion of "magic paper" (§9.4).

9. [Rescorla, 2013, §1, p. 682] usefully dubs Chalmers's view of implementation "structuralism about computational implementation". It is the fact that the *structure* of the physical system matches ("mirrors", in Chalmers's terms; more precisely, is isomorphic to) the structure of the computational system that matters. Although Rescorla agrees that such structural identity is necessary for a physical system to implement a computation, he denies that it is sufficient. That is, although any physical system that implements a computation must have the same structure as the computation, there are (according to Rescorla) physical systems that have the same structure as certain computations but that are not implementations of them (§1, p. 683). This is because *semantic* "relations to the social environment sometimes help determine whether a physical system realizes a computation" (Abstract, p. 681). The key word here is 'sometimes': "On my position, the implementation conditions for some but not all computational models essentially involve semantic properties" (§2, p. 684). Thus, Rescorla disagrees *both* with those such as [Ladyman, 2009] who hold to the "semantic view of computational implementation" (the view that if "a physical system implements a computation...[then] the system has representational properties") *and* with those such as [Chalmers, 1995] and [Piccinini, 2006a] who hold to the "non-semantic view" (the view "that semantics [n]ever informs computation" (§2, p. 683).

To argue for this, Rescorla must provide a "counterexample to the non-semantic view" (§1, p. 684). That is, he must provide an example of a physical implementation that requires a representational feature (that requires semantics). It is not enough to find an implementation that merely *has* a semantics; there are plenty of those, because a semantic interpretation can always be given to one. So here is one example that he gives, a Scheme program for Euclid's algorithm for computing GCDs (§4, p. 686):

```
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b)))))
```

This is a recursive algorithm that we can paraphrase in English as follows:

> To compute the GCD of integers *a* and *b*, do the following:
> If $b = 0$, then output *a*
> else compute the GCD of *b* and the remainder of dividing *a* by *b*.[8]

---

[8]Or, if you prefer:
To compute the GCD of *a* and *b*, do:
  If $b = 0$, then output *a*
  else do:
    Divide *a* by *b*; let *r* be the remainder;
    Compute the GCD of *b* and *r*.

Rescorla points out that, "To do that, the machine must represent numbers. Thus, the Scheme program contains content-involving instructions..." (§4, p. 687). A "content-involving instruction is [one that] is specified, at least partly, in semantic or representational terms" (§3, p. 685; he borrows this notion from [Peacocke, 1995]). So, the Scheme program is specified in semantic terms (specifically, it is specified in terms of integers). Therefore, if a physical system is going to implement the program, that physical system must represent integers; that is, it requires semantics. Hence, "The Scheme program is a counter-example to the non-semantic view of computational implementation" (§4, p. 687).

I can see that the machine *does* represent numbers (or can be interpreted as representing them). But why does he say that it *must* represent them? I can even see that for an agent to use such a physical computer to compute GCDs, the *agent must* interpret the computer as representing numbers. But surely an agent could use this computer, implementing this program, to print out interesting patterns of uninterpreted symbols.

To respond to this kind of worry, Rescorla asks us to consider two copies of this machine, calling them M_{10} and M_{13}. The former uses base-10 notation; the latter uses base-13. When each is given the input pair ('115', '20'), each outputs '5'. But only the former computes the GCD of 115 and 20. (The latter was given the integers 187 and 26 as inputs; but their GCD is 1.) So M_{10} implements the program, but M_{13} does not; yet they are identical physical computers.

One possible response to this is that the semantics lies in the user's interpretation of the inputs and outputs, not in the physical machine. Thus, one could say that the machines do both implement the program, but that it is the user's interpretation of the inputs, outputs, and that program's symbols that makes all the difference. After all, consider the following Scheme program:

```
(define (MYSTERY a b)
  (if (= b 0)
    a
    (MYSTERY b (remainder a b)))))
```

If we are using base-10 notation, then we can interpret 'MYSTERY' as GCD; if we are using base-13 notation, then we might either be able to interpret 'MYSTERY' as some other numerical function or else not be able to interpret it at all. In either case, our two computers both implement the 'MYSTERY' program.

One possible conclusion to draw from this is that any role that semantics has to play is at the level of the abstract computation, not at the level of the physical implementation.

Rescorla's response to this might be incorporated in these remarks:

> The program's formal structure does not even begin to fix a unique semantic interpretation. Implementing the program requires more

> than instantiating a causal structure that mirrors relevant formal structure. (§4, p. 688).

I agree with the first sentence: We can interpret the 'MYSTERY' program in many ways. I disagree with the term 'requires' in the second sentence: I would say that implementing the program only requires instantiating the mirroring causal structure. But I would go on to say that if one wanted to use the physical implementation to compute GCDs, then one would, indeed, be required to do something extra, namely, to provide a base-10 interpretation of the inputs and outputs (and an interpretation of 'MYSTERY' as GCD).

In fact, Rescorla agrees that the semantic interpretation of 'MYSTERY' as GCD is required: "there is more to a program than meaningless signs. The signs have an intended interpretation..." (§4, p. 689). But it is notoriously hard (some would say impossible) to pin down what "the intended interpretation" of any formal system is.

Further discussion of Rescorla's arguments would take us too far afield, but here are some questions to consider: Are the inputs to the Euclidean GCD algorithm *numerals* (like '10') or *numbers* (like 10 or 13)?[9] What about the inputs to a computer program written in Scheme that implements the Euclidean algorithm: Are its inputs numerals or numbers? (It may help to consider this analogous question: Is the input to a word-processing program the letter 'a' or an electronic signal or ASCII-code representing 'a'?)

10. [Rescorla, 2012a, 12; italics in original] gives another example of semantic computation, in the sense of a computation that requires *numbers*, not (merely) *numerals*:

> A register machine contains a set of memory locations, called *registers*. A program governs the evolution of register states. The program may individuate register states syntactically. For instance, it may describe the machine as storing *numerals* in registers, and it may dictate how to manipulate those syntactic items. Alternatively, the program may individuate register states representationally. Indeed, the first register machine in the published literature models computation *over natural numbers* [[Shepherdson and Sturgis, 1963, 219]]. A program for this numerical register machine contains instructions to execute elementary arithmetical operations, such as *add 1 or subtract 1*. A physical system implements the program only if can execute the relevant arithmetical operations. A physical system executes arithmetical operations only if it bears appropriate representational relations to numbers. Thus, a physical system implements a numerical register machine program only if it bears appropriate representational relations to numbers. Notably, a numerical register machine program ignores *how* the physical

---

[9]Note that the base-10 numeral '10' represents the number 10, but the base-13 numeral '10' represents the number 13.

system represent numbers. It applies whether the systems numerical notation is unary, binary, decimal, etc. The program characterizes internal states representationally (e.g. *a numeral that represents the number 20 is stored in a certain memory location*) rather than syntactically (e.g. *decimal numeral "20" is stored in a certain memory location*). It individuates computational states through denotational relations to natural numbers. It contains mechanical rules (e.g. *add 1*) that characterize computational states through their numerical denotations.

I agree that this is a *semantic* computation. Note that it is *not* a Turing machine (which *would* be a purley syntactic computation). And note that there cannot be a *physical* numerical register machine, only a *syntactic* one. This is not because there are no numbers, but because (if numbers do exist) they are not physical!

11. Re: Chalmers 1993:
    Note that Chalmers's implementing function (§2, p. 327) maps *from* the physical system *to* the FSA (or CSA). Thus, it is a *description* function: the *inverse* of a semantic *interpretation* function.

12. Rescorla seems to agree with me that implementation is semantic interpretation:

    > Physical system *P* realizes/implements computational model *M* just in case [c]omputational model *M* accurately describes physical system *P*. (§2, p. 1278)

    Concerning what I call 'implementation details', [Rescorla, 2014b, §2, p. 1280] says: "A physical system usually has many notable properties besides those encoded by our computational model: colors, shapes, sizes, and so on."

    However, a significant difference is that Rescorla's version is somewhat more narrowly focused than mine. Where I view any form of semantic interpretation as a form of implementation, Rescorla makes the notion more limited by elaborating on the following idea (§3, p. 1280): "*M* accurately describes *P* just in case *P* reliably moves through 'state space' according to mechanical instructions encoded by *M*." Putting these two statements together, we get this formulation:

    > Physical system *P* realizes/implements computational model *M* just in case *P* reliably moves through 'state space' according to mechanical instructions encoded by *M*.

    The difference in focus is due in part to the fact that Rescorla is more interested in the special case of implementation of a computation, whereas I am more interested in implementation more generally.

13. One place where Chalmers very clearly states that (what I call) an Abstraction is a *description* of (what I call) an implementation is here:

> Certainly, I think that when a physical system implements an a-computation [that is, a computation abstractly conceived], the a-computation can be seen as a description of it. [Chalmers, 2012, 215]

14. Somewhere in this chapter, must emphasize that abstraction is the opposite of implementation. (For a backward reference in §17.4.) (For some discussion of this, see Rapaport 1996, Ch.2, discussion of Rosenblueth & Wiener.)

14.6   Further Sources of Information

- Brown, Curtis (2004), "Implementation and Indeterminacy", in J. Weckert & Y. Al-Saggaf (eds.), *Computing and Philosophy Conference, Canberra: Conferences in Research and Practice in Information Technology*, Vol. 37 (Australian Computer Society): 27–31.

    – Accessed 23 October 2014 from:
      http://digitalcommons.trinity.edu/cgi/viewcontent.cgi?article=1000&context=phil_faculty

    – A critique of Chalmers's theory of implementation.

- Dresner, Eli (2010), "Measurement-Theoretic Representation and Computation-Theoretic Realization", *Journal of Philosophy* 108(6) (June): 275–292.

    – Examines "the association between numbers and the physical world that is made in measurement" and argues that implementation "and (measurement-theoretic) representation" are "a single relation (or concept) viewed from different angles" (p. 276).

- Eden, Amnon H.; & Turner, Raymond (2007), "Problems in the Ontology of Computer Programs", *Applied Ontology* 2(1): 13–36.

    – Preprint accessed 23 October 2014 from:
      http://www.eden-study.org/articles/2007/problems-ontology-programs_ao.pdf

    – §4 discusses "concretization...a process during which an entity or entities of one category are synthesized (come into being) from entities of a more abstract category".

- Kramer, Jeff (2007), "Is Abstraction the Key to Computing?", *Communications of the ACM* 50(4) (April): 36–42.

    – Accessed 23 October 2014 from:
      http://www.ics.uci.edu/∼andre/informatics223s2007/kramer.pdf

    – The role of abstraction (an inverse of implementation) is considered for its role in computer education.

- Ladyman, James (2009), "What Does It Mean to Say that a Physical System Implements a Computation?", *Theoretical Computer Science* 410(4–5) (17 February): 376–383.

    1. Accessed 28 October 2014 from:
       http://ac.els-cdn.com/S0304397508007238/1-s2.0-S0304397508007238-
       main.pdf?_tid=04248730-5f6c-11e4-8b44-
       00000aab0f26&acdnat=1414588006_ff4ead0d50f1774e6d0779cf705e19e6

    2. "A logical transformation is a map from a *set* of logical states to a *set* of logical states, whereas a physical process is a change in a physical system whereby it goes from a *particular* physical state to a *particular* physical state. Hence, strictly speaking a physical process cannot be said to implement a logical transformation because all it could ever do is implement the part of the map that takes one of the logical input states to another logical input state. ...For a physical system to implement a logical transformation there must be a family of processes and each of the physical states that represent the logical input states must be taken by one member of the family to the appropriate physical state, that is the one that represents the right logical output state." (p 379)

- Lycan, William G. (1987), "The Continuity of Levels of Nature", in William G. Lycan (ed.), *Mind and Cognition: A Reader* (Basil Blackwell): 77–96; excerpted from Chs. 4–5 of William G. Lycan, *Consciousness* (Cambridge, MA: MIT Press, 1987).

  – "...'software'/'hardware' talk encourages the idea of a bipartite Nature, divided into two levels, roughly the physiochemical and the (supervenient) 'functional' or higher-organizational—as against reality, which is a multiple *hierarchy* of levels of nature.... See Nature as hierarchically organized in this way, and the 'function'/'structure' distinction *goes relative*: something is a role as opposed to an occupant, a functional state as opposed to a realizer, or vice versa, only *modulo* a designated level of nature."

- McMullin, Ernan (1985), "Galilean Idealization", *Studies in History and Philosophy of Science* 16(3): 247–273.

  – Primarily an examination of the nature of idealization in the development of scientific theories (and how or whether such idealizations "falsify" or "misrepresent" reality), but also contains useful discussions of the differences between scientific theories and scientific models (esp. pp. 257ff).

- Popper, Karl (1978), "Three Worlds",
  http://tannerlectures.utah.edu/_documents/a-to-z/p/popper80.pdf
  (accessed 20 February 2015).

  – Popper's "three worlds" are the "world" of physical objects, the "world" of thinking and subjective experiences, and the "world of the products of the human mind", including languages, scientific theories, and physical artifacts. Many world-3 objects "are *embodied* or *physically realized*, in...world-1 physical objects".

- Rescorla, Michael (2012), "Are Computational Transitions Sensitive to Semantics?", *Australian Journal of Philosophy* 90(4) (December): 703–721.

  1. Preprint accessed 30 October 2014 from:
     http://www.philosophy.ucsb.edu/docs/faculty/papers/formal.pdf

  2. "Mathematical models of computation, such as the Turing machine, are abstract entities. They do not exist in space or time, and they do not participate in causal relations. Under suitable circumstances, a physical system *implements* or *physically realizes* an abstract computational model. Some philosophers hold that a physical system implements a computational model only if the system has semantic or representational properties [...[Ladyman, 2009]]. Call this *the semantic view of computational implementation.* In contrast, [[Chalmers, 1995], [Piccinini, 2006a]], and others deny any essential tie between semantics and physical computation. I agree with Chalmers and Piccinini." (§2.1, p. 705)

- Scherlis, William L.; & Scott, Dana S. (1983), "First Steps towards Inferential Programming", in R.E.A. Mason (ed.), *Information Processing* 83: 199–212,
  http://repository.cmu.edu/cgi/viewcontent.cgi?article=3542&context=compsci
  (preprint accessed 16 February 2015).

  – §2 discusses the relation between a specification the program that implements it.

- Scheutz, Matthias (1998), "Implementation: Computationalism's Weak Spot", *Conceptus Journal of Philosophy* 31(79): 229–239.

1. Accessed 21 October 2014 from:
   http://hrilab.tufts.edu/publications/scheutz98conceptus.pdf

2. Analyzes Searle's and Putnam's arguments, concluding that "a better notion of impelmentation is...[needed] that avoids state-to-state correspondences between physical systems and abstrat objects"

- Scheutz, Matthias (1999), "When Physical Systems Realize Functions", *Minds and Machines* 9: 161–196.

  1. Preprint accessed 22 October 2014 from:
     http://hrilab.tufts.edu/publications/scheutz99mm.pdf

  2. §3 is an especially good discussion of Putnam's argument.

  3. Scheutz approaches implementation by way of its inverse, abstraction (from a physical system); see §7.

  4. To refute Putnam 1988, Scheutz replaces the notion of "implementation of a computation" with "realization of a *function*". His view is not inconsistent with the "semantic" theory of Rapaport 1999, at least when Scheutz says (§7, p. 174): "what is the same [in the case of two realizations] is the syntactic structure of...the function"; that is, they are different semantic interpretations of the same syntactic structure.

- Scheutz, Matthias (2001), "Computational versus Causal Complexity", *Minds and Machines* 11: 543–566.

  – Preprint accessed 23 October 2014 from:
    http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.21.8293&represent=rep1&type=pdf

  – §2 critiques Chalmers's theory of implementation. The rest of the paper introduces a new theory of implementation based on a notion of "bisimulation".

- Scheutz, Matthias (2012), "What It Is *Not* to Implement a Computation: A Critical Analysis of Chalmers' Notion of Implementation", *Journal of Cognitive Science* (Republic of Korea) 13(1) (January-March): 75–106.

  – Accessed 30 October 2014 from:
    http://j-cs.org/gnuboard/bbs/board.php?bo_table=__vol013i1&wr_id=4

  – Argues that Chalmers's "definition of implementation still allows for unwanted implementations" (p. 75)

- Shagrir, Oron (2012) "Computation, Implementation, Cognition", *Minds and Machines* 22(2) (Summer): 137–148.

  – A critique of Putnam 1988, [Searle, 1990], [Chalmers, 1996], [Scheutz, 2001], [Piccinini, 2006a], arguing that there can be "systems that simultaneously implement different complex automata" (p. 137).

- Sloman, Aaron (2008), "Why Virtual Machines Really Matter—for Several Disciplines", lecture slides accessed 27 October 2014 from:
  http://www.cs.bham.ac.uk/research/projects/cogaff/talks/information.pdf

– Explores this definition of implementation (and its relationship to the concept of a virtual machine): "Phenomena of type X... are fully implemented in, or realised in, or grounded in phenomena of type Y... if and only if:

(a) type X phenomena cannot exist without some entities and processes of type Y. ...

(b) certain entities and processes of type Y are sufficient for the phenomena of type X to exist—they constitute the implementation."

(slide 15)

• Wilson, Robert A. (2001), "Two Views of Realization", *Philosophical Studies* 104: 1–31.

– Argues that realization is "context sensitive".

• Also cite Rescorla.
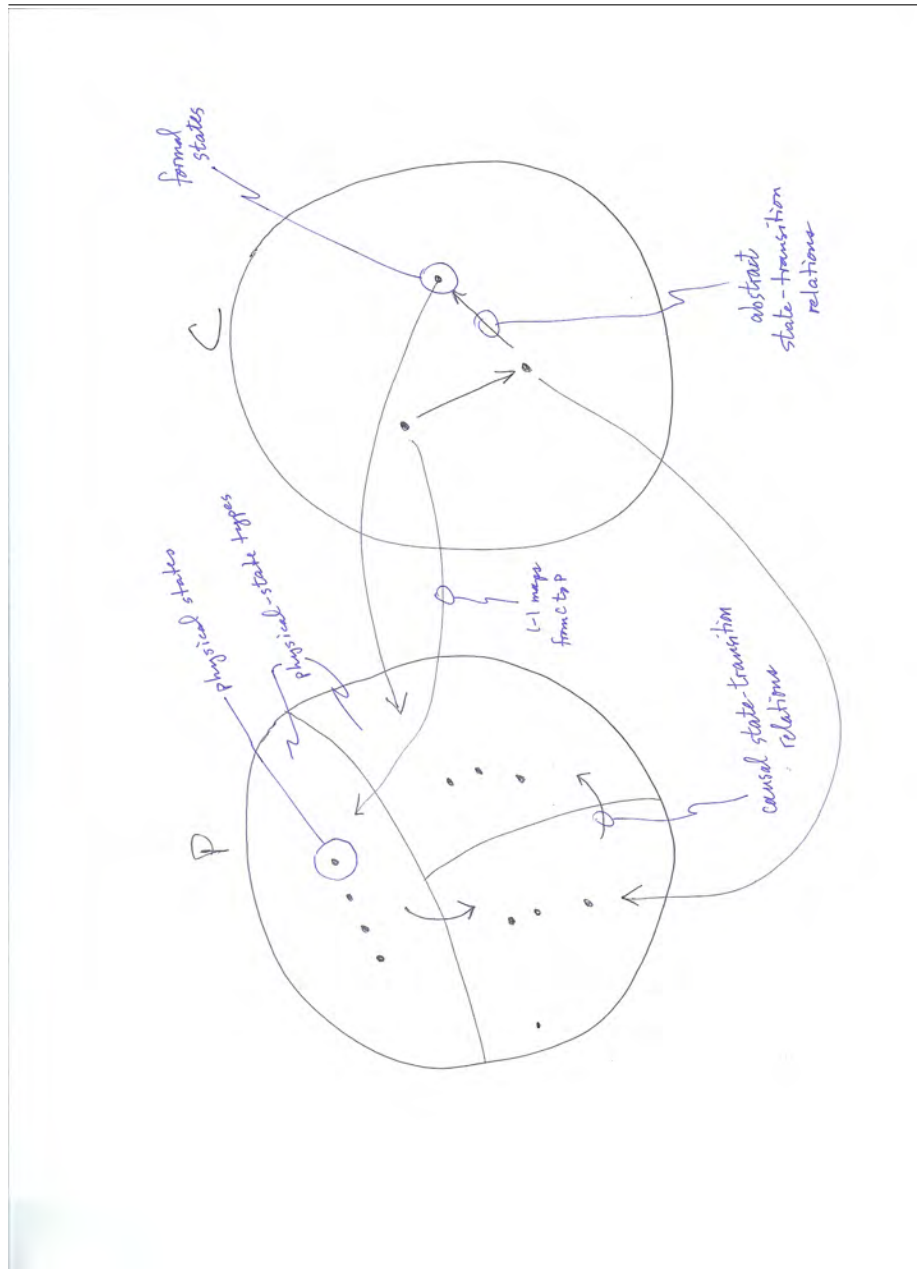
Figure 14.2: P is a physical system whose physical states (dots) are grouped into state types (partitions of P) related by causal state-transtion relations; C is a computation whose formal states (dots), which are related by abstract state-transition relations, are mapped to P's physical-state types by abstract state-transition relations.
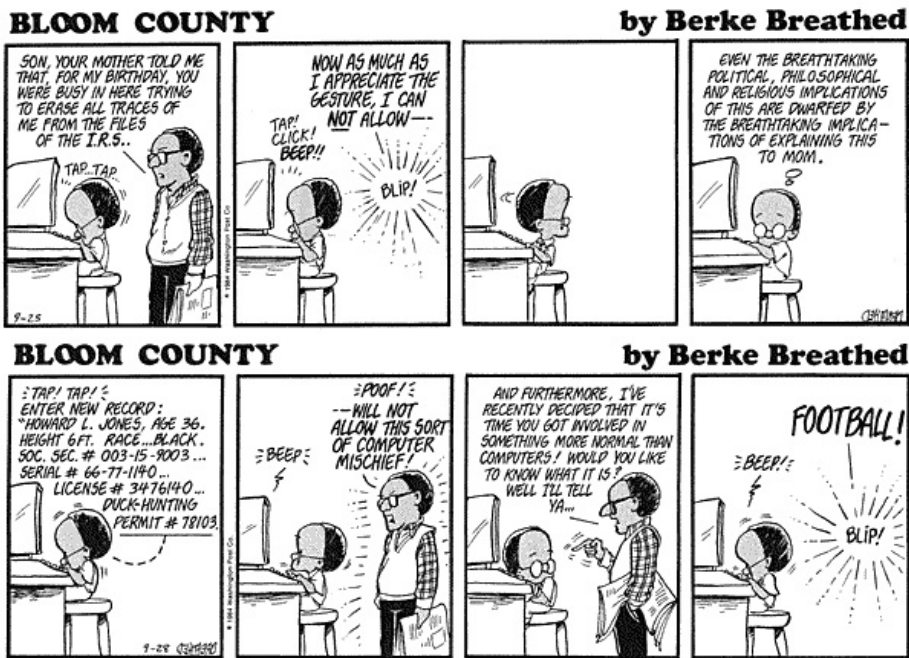
Figure 14.3: ©2010, Berkeley Breathed

Chapter 15

What Is a Computer Program? III. Are Computer Programs Theories?

I haven't formalized my theory of belief revision, but I have an algorithm that does it.
— Frances L. Johnson (personal communication, February 2004).

...within ten years most theories in psychology will take the form of computer programs, or of qualitative statements about the characteristics of computer programs.
— [Simon and Newell, 1958, 7–8]

Figure 15.1: ©2014, Universal Uclick

15.1   Recommended Readings:

1. Required:

   - Simon, Herbert A. (1996), "Understanding the Natural and Artificial Worlds", Ch. 1, pp. 1–24 of Simon's *The Sciences of the Artificial, Third Edition* (Cambridge, MA: MIT Press).

     – Originally written in 1969; updated in 1981.
     – In this chapter, Simon also has interesting things to say about whether computer science is a science.

   - Thagard, Paul (1984), "Computer Programs as Psychological Theories", in O. Neumaier (ed.), *Mind, Language, and Society* (Vienna: Conceptus-Studien): 77–84.

     – A critique of the whole idea.

2. Very Strongly Recommended:

   - Weizenbaum, Joseph (1976), *Computer Power and Human Reason: From Judgment to Calculation* (New York: W.H. Freeman).

     – Ch. 5 ("Theories and Models"), pp. 132–153.
     – Ch. 6 ("Computer Models in Psychology"), pp. 154–181.
       * Ch. 6 discusses computer programs as theories and the potential evils of AI, as well as presenting Weizenbaum's objections to Simon.

3. Strongly Recommended:

   - Moor, James H. (1978), "Models and Theories", §4 of his "Three Myths of Computer Science", *British Journal for the Philosophy of Science* 29(3) (September): 213–222.

4. Highly Recommended:

   - Simon, Herbert A.; & Newell, Allen (1956), "Models: Their Uses and Limitations", in Leonard D. White (ed.), *The State of the Social Sciences* (Chicago: University of Chicago Press): 66–83.

     – Makes many of the same points as Simon 1996, but goes into a bit more detail on the nature of theories and models.

5. Recommended:

   - Wilks, Yorick (1990?), "One Small Head: Models and Theories", pp. 121–134, in §4 ("Programs and Theories") of Derek Partridge & Yorick Wilks (eds.), *The Foundations of Artificial Intelligence: A Sourcebook* (Cambridge, UK: Cambridge University Press).

     – Published in slightly different form as:
       Wilks, Yorick (1974), "One Small Head—Models and Theories in Linguistics", *Foundations of Language* 11(1) (January): 77–95.
     – Has a useful, if sometimes confusing, overview of the many meanings of 'theory' and 'model'.

   - Daubert v. Merrell Dow Pharmaceuticals (92-102), 509 U.S. 579 (1993)
     http://openjurist.org/509/us/579 (accessed 13 November 2012).

    – A Supreme Court case concerning what counts as "generally accepted" reliability by the scientific community. Has interesting observations on the nature of scientific theories and expertise. My colleague Sargur N. Srihari recommended this to me, after his experience being called as an expert witness on handwriting analysis, on the grounds that his computer programs that could recognize handwriting were scientific theories of handwriting analysis.

- Green, Christopher D. (2004), "Philosophical Approaches to Explanation and Scientific Models, and Their Relations to Connectionist Cognitive Science", Ch. 3 of his "(How) Do Connectionist Networks Model Cognition?" (Ph.D. dissertation, Department of Philosophy, University of Toronto).

    – Contains a useful survey of different views of scientific explanation and scientific models, embedded in a discussion of connectionism.

15.2   Introduction

The issue raised in the quote that opens this chapter, from a former graduate student at my university, is whether an algorithm, which is a pretty formal, precise thing, is different from a formal theory. Some might say that her algorithm *is* her theory. Does it really make sense to say that you *don't* have a formal theory of something if you *do* have a formal algorithm that implements your theory? Roger Schank, an AI researcher famous for taking a "scruffy"—that is, non-formal—approach to AI used formal algorithms to express his non-formal theories. That sounds paradoxical.

As part of our investigation into the nature of computer programs, we have seen that algorithms are said to be implemented in computer programs. But, if implementation is semantic interpretation, then computer programs are semantic interpretations of algorithms, in the medium of some programming language. However, some philosophers have argued that computer programs are theories; but theories are more like abstractions than they are like implementations. And others have argued that computer programs are simulations or models of real-world situations, which sounds more like an implementation than an abstraction. In this chapter, we will look into the relation of a program to that which it models or simulates, the nature of simulation, and whether programs are (scientific) theories.

15.3   Three Distinctions

Let's begin by considering three distinctions, between "simulations" and "emulations", between "mere simulations" and "the real thing", and between "theories" and "models".

15.3.1   Simulation vs. Emulation

There is no standard, agreed-upon definition of either 'simulation' or 'emulation'. (This unfortunate situation occurs all too frequently. Therefore, it is always important for you to try to find out how a person is using such terms before deciding whether to agree with what they say about them.)

Following [Roth, 1983], let's say that

> *x* **simulates** *y* means (roughly): "*y* is a real or imagined system, and *x* is a
> model of *y*, and we experiment with *x* in order to understand *y*".

This is only a rough definition, because I have not said what is meant by 'system', 'model', or 'understand', not to mention 'real', 'imagined', or 'experiment'! Typically, a computer program (*x*) is said to simulate some real-world situation *y* when program *x* is a model of situation *y*. If we want to understand the situation, we can do so by experimenting with the program; perhaps the program is easier to deal with or to manipulate than the real-world situation. In an extreme case, *x* simulates *y* if and only if *x* and *y* have the same input-output behavior, but they might differ greatly in some of the details of how they work.

And, following [Habib, 1983], let's say that

x **emulates** *y* means (roughly)

> *either*: "*x* and *y* are computer systems, and *x* interprets and executes *y*'s instruction set by implementing *y*'s operation codes in *x*'s hardware"—that is, hardware *y* is implemented as a virtual machine on *x*

> *or*: "*x* is some software feature, and *y* is some hardware feature, and *x* simulates *y*, doing what *y* does "exactly" as *y* does it.

In general, *x* emulates *y* if and only if *x* and *y* not only have the same input-output behavior (*x* not only simulates *y*) but *x* also uses the same algorithms and data structures as *y*.

It is unlikely that being a simulation and being an emulation are completely distinct notions. More likely, they are the ends of a spectrum, in the middle of which are *x*s and *y*s that differ in the level of detail of the algorithms and data structures that *x* uses to do *y*'s job. At the "pure" simulation end of the spectrum, only *x*'s and *y*'s external, input-output behavior agree; at the "pure" emulation end, all of their internal behavior also agree. Perhaps, then, the only pure example of emulation would be *y* itself! Perhaps, even, there is no real distinction between simulation and emulation except for the degree of faithfulness to what is being simulated or emulated (and except for cases in which *y* is only an imagined situation, whereas *x* will always be something real).

15.3.2 Simulation vs. the Real Thing

On the other hand, to say that *x* is "merely" a simulation of *y* is to suggest that *y is* real but that *x* is not. The word 'simulation' has a connotation of "imitation" or "unreal". For example, it is often argued that a simulation of a hurricane is not a real hurricane, or that a simulation of digestion is not real digestion.

But there are cases where a simulation *is* the real thing. (Or would such simulations be better called 'emulations'?) For example, although a scale model of the Statue of Liberty is not the real Statue of Liberty, a scale model of a scale model (of the Statue of Liberty) *is* itself a scale model (of the Statue of Liberty). A Xerox or PDF or faxed copy of a document *is* that document, even for legal purposes (although perhaps not for historical purposes; see [Korsmeyer, 2012]). Some philosophers and computational cognitive scientists have argued that a computational simulation of cognition really is cognition [Edelman, 2008], [Rapaport, 2012b]. And, in general, a simulation of information *is* that information.

15.3.3 Theories vs. Models

When people say that something is a "theory" in ordinary language, they often mean that it isn't necessarily true, that it is mere speculation. But scientists and philosophers use the word 'theory' in a more technical sense. (For a humorous illustration of this, see Fig. 15.1.)

This, by the way, is one reason that people who believe in the "theory" of evolution and those who don't are often talking at cross purposes, with the former saying that evolution is a true, scientific theory:

> Referring to biological evolution as a theory for the purpose of contesting it would be counterproductive, since scientists only grant the status of theory to well-tested ideas." (Terry Holliday, Kentucky education commissioner, 2011; cited in *Science* 337 (24 August 2012): 897.)

and the latter saying that, if it is only a theory—if, that is, it is a mere conjecture—then it might not be true:

> The theory of evolution is a theory, and essentially the theory of evolution is not science—Darwin made it up. (Kentucky state Representative Ben Waide, 2011; cited in *Science* 337 (24 August 2012): 897.)

They are using the word in very different senses.

Further complicating the issue, there are at least two views within the philosophy of science about what scientific theories are:

On the *syntactic* approach to theories (due to the "Logical Positivists" [Uebel, 2012]), a theory is an abstract description of some situation (which usually is, but need not be, a real-world situation) expressed in a formal language with an axiomatic structure; that is, a theory is a formal system (see §14.4). Such a "theory" is typically considered to be a set of sentences (linguistic expressions, well-formed formulas) that describe a situation or that codify (scientific) laws about a situation. (This is the main sense in which the theory of evolution is a "theory".) Such a description, of course, must be expressed in some language. Typically, the theory is expressed in a formal, axiomatic language that is semantically interpreted by rules linking the sentences to "observable" phenomena. These phenomena are either directly observable—either by unaided vision or with the help of microscopes and telescopes— or are theoretical terms (such as 'electron') that are definable in terms of directly observable phenomena (such as a vapor trail in a cloud chamber). (We'll say more about this in §15.4 —CHECK THIS!.)

On the *semantic* approach to theories (due largely to the philosopher Patrick Suppes [Frigg and Hartmann, 2012]), theories are the set-theoretic models of an axiomatic formal system. Such models are isomorphic to the real-world situation being modeled. (Weaker semantic views of theories see them as "state spaces" (http://en.wikipedia.org/wiki/State_space) or "prototypes" (http://en.wikipedia.org/wiki/Prototype), which are merely "similar" to the real-world situation.)

15.4   Computer Programs as Theories

Computational cognitive scientists such as P.N. Johnson-Laird, Allen Newell, Zenon Pylyshyn, and Herbert Simon have all claimed that computer programs are theories, in the sense that the programming languages in which they are written are languages for theories, that the programs are ways to express theoriest This informal way of putting

the idea sounds circular, so let's see if we can be a bit more precise. First, consider these passages from their writings (my italics throughout):

[Simon and Newell, 1962, 97]:

> 1. Computers are quite general symbol-manipulating devices that can be programmed to perform nonnumerical as well as numerical symbol manipulation.
>
> 2. Computer programs can be written that use nonnumerical symbol manipulating processes to perform tasks which, in humans, require thinking and learning.
>
> 3. *These programs can be regarded as theories, in a completely literal sense, of the corresponding human processes.* These theories are testable in a number of ways: among them, by comparing the symbolic behavior of a computer so programmed with the symbolic behavior of a human subject when both are performing the same problem-solving or thinking tasks.

[Johnson-Laird, 1981, 185–186]:

> Computer programming is too useful to cognitive science to be left solely in the hands of the artificial intelligenzia [sic]. *There is a well established list of advantages that programs bring to a theorist: they concentrate the mind marvelously; they transform mysticism into information processing, forcing the theorist to make intuitions explicit and to translate vague terminology into concrete proposals; they provide a secure test of the consistency of a theory and thereby allow complicated interactive components to be safely assembled; they are "working models" whose behavior can be directly compared with human performance.* Yet, many research workers look on the idea of developing their theories in the form of computer programs with considerable suspicion. The reason. . . [i]n part. . . derives from the fact that any large-scale program intended to model cognition inevitably incorporates components that lack psychological plausibility. . . . The remedy. . . is not to abandon computer programs, but to make a clear distinction between a program and the theory that it is intended to model. For a cognitive scientist, the single most important virtue of programming should come. . . from the business of developing [the program]. Indeed, the aim should be neither to simulate human behavior. . . nor to exercise artificial intelligence, but to force the theorist to think again.

[Pylyshyn, 1984, 76]:

> [T]he. . . requirement—that we be able to implement [a cognitive] process in terms of an actual, running program that exhibits tokens of the behaviors in question, under the appropriate circumstances—has far-reaching consequences. One of the clearest advantages of *expressing a*

> *cognitive-process model in the form of a computer program* is, it provides
> a remarkable intellectual prosthetic for dealing with complexity and for
> exploring both the entailments of a large set of proposed principles and
> their interactions.

[Johnson-Laird, 1988, 52]:

> *[T]heories of mind should be expressed in a form that can be modelled*
> *in a computer program.* A theory may fail to satisfy this criterion for
> several reasons: it may be radically incomplete; it may rely on a process
> that is not computable; it may be inconsistent, incoherent, or, like a
> mystical doctrine, take so much for granted that it is understood only by it
> adherents. These flaws are not always so obvious. Students of the mind do
> not always know that they do not know what they are talking about. The
> surest way to find out is to try to devise a computer program that models
> the theory.

The basic idea is that a theory must be expressed in some language. If you don't
express it in a language, how do you know what it is? As E.M. Forster is alleged to
have said, "How can I know what I think till I see what I say?". And if you don't write
your theory down in some language, no one can evaluate it.

Scientific theories, on this view, are sets of sentences. And the sentences have to
be in some language: Some theories are expressed as a program written in English,
some in the language of mathematics, some in the language of formal logic, some in
the language of statistics and probability. The claim here is that some theories can be
expressed in a programming language.

One advantage of expressing a theory as a computer program is that *all* details must
be filled in. That is, a computer program must be a full "implementation" of the theory.
Of course, there will be implementation-dependent details: For example, if the theory
is expressed in Java, there will be details of Java that are irrelevant to the theory itself.
So, one must try to ensure that such details are indeed irrelevant. One way to do so is to
make sure that two computer programs expressing the same theory but that are written
in two different programming languages—with different implementation-dependent
details—have the same input-output, algorithmic, and data-structure behavior (that is,
that they fully emulate each other).

Another advantage of expressing a theory as a computer program is that you can
run the program to see how it behaves and what predictions it makes. So, in a sense,
the theory becomes its own model and can be used to test itself.

15.5 Models

The notion of *model* is associated with the Puzzle of the Model in the Middle, or the
Model Muddle [Wartofsky, 1966], [Wartofsky, 1979], [Rapaport, 1995].

There are different uses of the term 'model': It can be used to refer to a syntactic
domain, as in the phrase 'mathematical model' of a real-world situation. And it can
be used to refer to a semantic domain, as in the phrase 'set-theoretic model' of a
mathematical theory.

This dual, or Janus-faced,[1] nature of models leads to what [Smith, 1987] called a "correspondence continuum": Scientists typically begin with data that they then interpret or model using a formal theory; so, the data are the syntactic domain, and the formal theory is its semantic domain. The formal theory can then be modeled set-theoretically or mathematically; so, the formal theory is now the syntactic domain, and the set-theoretic or mathematical model is the semantic domain. But that set-theoretic or mathematical model can be interpreted by some real-world phenomenon; so, the model is now the syntactic domain, and the real-world is the semantic domain. To close the circle, that real-world phenomenon is just the some of the kind of data that we started with! (Compare the example of student records and students at the end of our discussion of implementation, in §14.4.)

15.6 Computer Programs Aren't Theories

NEED TO SAY MORE ABOUT THEIR ARGUMENTS!

However, [Moor, 1978, §4] and [Thagard, 1984] argue that computer programs are *not* theories, on the grounds that they are neither sets of (declarative) sentences nor set-theoretic models of axiom systems.

Here are some questions to ask yourself about this position:

1. Must a syntactic theory be expressed in declarative sentences? (A declaritive sentence is a sentence that is either true or else false.)

2. Must a computer program be expressed in imperative language? (An imperative sentence is a sentence that says "Do this!"; it has no truth value.)

3. Must a semantic theory be a set-theoretic model of a real-world situation?

4. Could a computer *process*—that is, a program being executed—be a model of a real-world sitation?

---

[1]http://en.wikipedia.org/wiki/Janus

15.7   NOTES FOR NEXT DRAFT

1. Notes on Simon & Newell 1956 (see Highly Recommended readings, above):

   (a) Here are one or two (or three?) arguments to the effect that programs are theories:

      i. They equate 'model' and 'theory' (p. 66), but they admit that some people limit the notion of "model" to *mathematical* theory. They also state that "mathematics is a language" (p. 66), one of "three main kinds of scientific languages or theories: the mathematical, the verbal, and the analogical" (p. 67).

         Presumably, they would classify programming languages as being of the mathematical kind, from which it would follow that computer programs are theories expressed in that language.

      ii. It should also be noted that, at the end of the article, they equate 'theory' with 'analogy' (p. 82; see also [Hofstadter and Sander, 2013]. Insofar as a computer program (better?: a computer process) is an analogy to some real-world phenomenon, it is thereby a theory of that phenomenon.

      iii. More precisely, they "define a theory simply as a set of statements or sentences. (They may, of course, be mathematical statements, or equations, instead of verbal statements.)" (p. 67).

         Presumably, the statements of a theory could be statements of a computer program, expressed in a programming language. Then, clearly, a computer program, considered as a set of statements, would be a theory.

   (b) A theory, for them, is a syntactic object: a set of sentences. Its semantic counterpart is the "content" of the theory, desribed thus:

      > . . . the content of a theory is comprised of all the *assertions* about the world, whether true or not, that are explicitly stated by the theory or that can logically be inferred from the statements of the theory. (p. 68, my italics)

      First, they seem to be considering three items here: (syntactic) sentences of some language, the "assertions" made by those sentences, and the world. I think that what they are calling 'assertions' are what others might call 'propositions'. (We discussed the relation of a sentence to a proposition in Ch. 12; briefly, the two *sentences* 'It is snowing' (in English) and 'Il neige' (in French) both express a single *proposition*.) So, the English sentence 'It is snowing' expresses the proposition that it is snowing (or, to use their terminology, that sentence asserts that it is snowing), that that proposition (or assertion), in turn is true of the world just in case that it is (really) snowing.

Similarly, the computer programming sentence '$x := x + 1$' might express the situation (let's call it that) that x is to be incremented by 1, which will be (or become) true if the contents of the register named 'x' are incremented by 1. (We will discuss this kind of semantics for programming languages a bit more in Ch. 16.)

Second, normally, one can only logically infer one statement from another in a *declarative* language. If a programming language is not declarative, then a logic for it might need to be developed in order to be able to talk about logical inference of one statement from another. There are, in fact, logics for imperative languages, so this need not be a serious issue. Moreover, as we will see in Ch. 16, declarative comments can be added to programs, and one can reason about the statements in a program by reasoning about those comments.

(c) Another important point that they make is that the language that a theory is expressed in can make the content of that theory easier or harder to understand and analyze: It is easier to do arithmetic using Arabic numerals than Roman numerals (p. 69).

(d) They do explicitly discuss computers in this context, first on p. 71, where they talk about "*Computing machines* that have been programed [sic] to represent a particular theory. . . ." Note that here it is the physical computer that represents a theory. This is slightly different from talking about the computer's program as expressing a theory. On their way of talking "the computer is programed to carry out the arithmetic computations called for in the mathematical theory [of some phenomena]. Thus, the computer is an analogue for the arithmetic process." I find this rather unclear, but, in any case, they go on to emphasize (italics in the original) that "*This is not, however, the only way of employing digital computing machines as theories*".

(e) There is certainly a difference between a theory and the world that it is a theory of. One such difference is this:

> Why should theories of all kinds make irrelevant statements—possess properties not shared by the situations they model? The reason is clearest in the case of electromechanical analogues. T operate at all, they have to obey electromechanical laws—they have to be made of something—and at a sufficiently microscopic level these laws will not mirror anything in the reality being pictured. If such analogies serve at all as theories of the phenomena, it is only at a sufficiently high level of aggregation. (p. 74).

This is the problem of implementation-dependent details that we discussed in Ch. 14. It is an unavoidable problem arising whenever an Abstraction is implemented. It is only at the more abstract levels ("sufficiently high level[s] of aggregation") that we can say that an implementation and a corresponding abstract theory are "the same".

(f) Here is a reason why theories expressed as computer programs may be better than theories expressed in mathematics or in English ("verbally"): It has to do with the idea that such computational theories are analogies:

> . . . what is the particular value of the computer analogy? Why not work directly toward a mathematical (or verbal) theory of human problem-solving processes without troubling about electronic computers? ldotsit is at least possible, and perhaps even plausible, that we are dealing here with systems of such complexity that we have a greater chance of building a theory by way of the computer program than by a direct attempt at mathematical formulation. (p. 81)

Note first that they seem to consider computer programs as analogy theories, not mathematical theories! Second, computation is perhaps the best way of managing complexity (as we saw in Ch. 3).

(g) There are two advantages of expressing a theory in a programming language. "First, we would experiment with various modifications of the. . . program to see how closely we could simulate in detail the observable phenomena" (pp. 81–82). In other words, we can run the program to see how it behaves—to see how good a theory it is—and we can then modify the program (and then run the modified version) in order to make it a better theory.

Second, the program can be written in such a way that it explains what it is doing: "The computer, however complex its over-all program, could be programed to report, in accurate detail, a description of any part of its own computing processes in which we might be interested" (p. 82). This, of course, can make it easier to debug and improve the program, or, to use different terminology, to correct and improve the theory.

2. Where Simon and Newell 1956 consider models and theories to be the same thing, others distinguish them. [Apostel, 1961, 1–2] suggests that a computer can be a model of the central nervous system, and that that model might be easier to study than the system itself. If it is the *computer* that is the model, then it makes sense to say that the computer's *program* expresses the theory that the model is a model of. (Keep in mind, however, that there is an ambiguity over whether the model is a model of a theory or of some other phenomenon.)

3. For [Weizenbaum, 1976, Ch. 5, pp. 140–143], "A theory is first of all a text, hence a concatenation of the symbols of some alphabet." But "A theory is. . . not merely any grammatically correct text that uses a set of terms somehow symbolically related to reality. It is a systematic aggregate of statements of laws. Its content. . . lies at least as much in the structure of the interconnections that relate its laws to one another, as in the laws themselves." And we have to be abel to "draw. . . consequences from it". So, a theory, for Weizenbaum, is what we have elsewhere (WHERE?) called a 'formal theory': a set of axioms (laws) and rules of inference expressed in some formal language. With a few modifications

(such as to convert this kind of declarative system into an imperative one), this can be seen to describe a computer program expressed in some programming language.

Indeed, [Weizenbaum, 1976, 144–145] says just this:

> . . . theories are texts. Texts are written in a language. Computer languages are languages too, and theories may be written in them. . . . Theories written in the form of computer programs are ordinary theories as seen from one point of view. . . . But the computer program has the advantage [over "a set of mathematical equations" or even a theory written in English] not only that it may be understood by anyone suitable trained in its language, . . . but that it may also be run on a computer. . . . A theory written in the form of a computer program is thus both a theory and, when placed ona computer and run, a model to which the theory applies.

He cites Newell and Simon (1958?) in support of this.

I would make a slight distinction between the theory (the program) and the model (the computer or process).

For [Weizenbaum, 1976, 143–144], "a model *satisfies* a theory; that is, a model obeys those laws of behavior that a corresponding theory explicitly states or which may be derived from it. We may say, given a theory of a system *B*, that *A* is a model of *B* if that theory of *B* is a theory of *A* as well." So, a model (all by itself) is a (semantic) implementation of a theory. And a model *of* some system is a distinct implementation of a theory of that system. We begin with some system, we develop a theory about it (so the original system is a model of that theory), and then we can find or create another system that is also a model of that theory. If the second system is easier to understand than the original system, then we can use both the theory and the new model to understand the original system.

If a computer program is such a theory, then what would count as a model of it? One possibility, consistent with what others have said, is that a computer that is executing the program is the model. (Another possibility is that the process that is being executed is the model.)

4. Not sure exactly where this comment should go. Possibly much earlier, in Ch. 3, where I talk about computer science as the study of what is computable. Or possibly later, in Ch. 19, when I discuss computationalism in cognitive science.

In any case, [Weizenbaum, 1976, Ch. 6, esp. p. 157] makes am important point about computer programs as theories (and computers as models). He notes that,

when one studies a model *B* of some theory of a system *X*, because *B* is easier
to study than the system *X* that is really of interest, one is essentially using a
metaphor. But such uses have their limitations:

> To those fully in the grip of the computer metaphor, to understand *X*
> is to be able to write a computer program that realizes *X*.

The problem that Weizenbaum is pointing to is that, while *B* might be a good way
to understand some aspects of *X*, it is unlikely to give us a full understanding of
X or to be the best or only way to understand *X*. There are limitations to any
theory and to any model.

This does not mean, however, that one shouldn't try to understand *X* via *B*.

5. On [Thagard, 1984]:

   (a) [Thagard, 1984, 77] argues that, on two conceptions of what a theory is—
       the "syntactic" and the "semantic"—computer programs are not theories.
       He also argues that programs are not "models". Rather, "a program is a
       simulation of a model which approximates to a theory".

       On the syntactic theory of theories, a theory is a set of sentences (perhaps
       expressed as a formal system with axioms GIVE CITATION TO SECTION
       OF THIS BOOK WHERE THIS IS DISCUSSED). On the semantic theory
       of theories, a theory is a "definition of a kind of system". Presumably, he
       will argue that a program is neither a set of sentences nor a definition (of a
       kind of system). He has not yet said what a "model" is.

   (b) Indeed, on pp. 77–78, he argues that, because programs are sets of
       instructions, which do not have truth values and hence are not sentences,
       programs cannot be theories in the syntactic sense. One question that
       can be raised about this is whether programs written in a programming
       language such as Prolog, whose statements can be interpreted as declarative
       sentences with truth values, could be considered to be theories. If so, then
       why couldn't any program that was equivalent in some sense to such a
       Prolog program also be considered a theory?

       Another question that can be raised is this: Suppose that we are looking for
       a buried treasure. I might say, "I have a theory about where the treasure is
       buried: Walk three paces north, turn left, walk 5 paces, and then dig. I'll
       be that you find the treasure." Is this not a theory? I suppose that Thagard
       might say that it isn't. But isn't there a sentential theory that is associated
       with it—perhaps something like "The treasure is buried at a location that
       is three paces north and 5 paces west of our current location". Doesn't my
       original algorithm for finding the treasure carry the same information as
       this theory?

(c) [Thagard, 1984, 78] considers a *model* to be "a set-theoretic interpretation of the sentences in a [syntactic] theory.... ...a system of things...which provide an interpretation of the sentences". But "a program is not...a system of things, nor does it*ldots*provide an interpretation for anything". Hence, a program is not a model.

But a program being executed—a process—can be considered to be a system of (virtual) things that are interpretations of data structures in the program. If a process might be a model, then why couldn't the program be a theory?

(d) On the semantic (or "structuralist" [Thagard, 1984, 78]) view of theories, a theory "is a definition of a kind of natural system" (p. 79). Given some scientific laws (which, presumably, are declarative, truth-functional sentences, perhaps expressed in the language of mathematics), we would say that something is a certain kind of natural system "if and only if it is a system of objects satisfying" those laws. (The system is *defined* as being something that satisfies those laws.)

But this seems very close to what a model is. In fact, [Thagard, 1984, 79] says that a "real system R is a system of the kind defined by the theory T". But how is that different from saying that R is an implementation of (that is, a model of) T?

Thagard's response is that, first, "a program...can not be said to *define* a system" and that, second, programs contain "a host of characteristics which we know to be extraneous" to the real system that they are supposed to be like (p. 79). (We have discussed the notion of such "implementation side-effects" in Ch. 14. HAVE WE?)

(e)   First, we can at best say that a program *simulates* a system: it does not define a system..." [Thagard, 1984, 79]

He says this because simulations aren't definitions: A *simulation* of the solar system, to use his example (see §9.7.2), doesn't *define* the solar system. This seems reasonable, but it also seems to support the idea that a process (not necessarily a program) is a model (and hence that a program would be a theory).

(f) As for the problem of implementation details, [Thagard, 1984, 80] says that "if our program [for some aspect of human cognition] is written in LISP, it consists of a series of definitions of functions. The purpose of writing those functions is not to suggest that the brain actually uses them,, but to simulate at a higher level the operation of more complex processes such as image...processing." In other words, the real system that is being simulated (modeled?) by the program (process?) need not have LISP functions. But, as he notes, it will have "complex processes" that do the same thing as the LISP functions. But isn't this also true of any theory

compared to the real system that it is a theory of? A theory of cognitive behavior expressed in declarative sentences will have, say, English words in it, but the brain doesn't.

(g) [Thagard, 1984, 80] does admit that he "shall take models to be like theories (on the semantic conception) as being definitions of kinds of systems." And he notes that "a model contains specifictions which are know to be false of the target real system"—that is, implementation details! The problem with this, according to Thagard, is that if you try to make a prediction about the real system based on the model, you might erroneously make it based on one of these implementation details (pp. 80–81). But that seems to be a problem endemic to any model (or any theory, for that matter).

(A similar point is made in [Humphreys, 1990, 501]: "Inasmuch as the simulation has abstracted from the material content of the system being simulated, has employed various simplifications in the model, and uses only the mathematical form, it obviously and trivially differs from the 'real thing', but in the respect, there is no difference between simulations and any other kind of mathematical model...".)

If you make a prediction that turns out to be false, you may have to change your theory or your model. Perhaps you have to eliminate that implementation detail. (But others will always crop up; otherwise, your theory or model will not merely describe or simulate the real system; it will *be* the real system. But there are well-known reasons why a life-sized map of a country is not a very good map! CITE CARROLL, BORGES, ET AL.)

(h) Thagard's final summary is not really the wholesale rejection of programs as theories that it might first appear to be. It is more subtle:

> a program *P*, when executed on a computer, provides a simulation of a system of a kind defined by a model *M*, where *M* defines systems which are crude versions of the systems defined by a theory *T*, and the set of systems defined by *T* is intended to include the real system *R*. [Thagard, 1984, 82]

I can live with this: The process and *R* are both implementations of *T*.

6. On Daubert vs. Merrell-Dow:

  (a) Questions to consider:

    i. Is a computational theory (of *X*) a theory?
    ii. Is a computational theory (of *X*) a *scientific* theory?
    iii. What is a computational theory (of *X*)?

  (b) Presumably, a computer scientist is an expert on computer science. But is a computer scientist who writes a computer program about (or whom develops a computational theory of) something else (*X* ≠ computer science) thereby an expert on *X*? (Or must that computer scientist *become*, or work with, an expert on *X*?)

(c) Two good quotations about the nature of science (perhaps they should go in Ch. 4?):

> . . . scientists do not assert that they know what is immutably 'true'—they are committed to searching for new, *temporary* theories to explain, *as best they can,* phenomena" (Brief for Nicolaas Bloembergen et al. as Amici Curiae 9, cited in Daubert at II.B.24 in the online version, my italics)

and

> Science is not an encyclopedic body of knowledge about the universe. Instead, it represents a process for porposing and refining theoretical explanations about the world that are subject to further testing and refinement. (Brief for American Association for the Advancement of Science and the National Academy of Sciences as Amici Curiae 7–8, cited in Daubert at II.B.24)

(d) Justice Blackmun, writing in Daubert at II.B.24, citing these two quotes, states that "in order to qualify as 'scientific knowledge,' an inference or assertion must be derived by the scientific method". So, if a computer program that can, say, identify handwriting is a good *scientific* theory of handwriting, then its creator is a scientific expert on handwriting?

There are two concerns with this: First, a computer program that can identify handwriting need not be a good scientific theory of handwriting. It might be a "lucky guess" not based on any scientific theory, or it might not even work very well outside carefully selected samples. Second, even if it is based on a scientific theory of handwriting and works well on arbitrary samples, the programmer need only be a good interpreter of the theory, not necessarily a good handwriting scientist.

However, if a computer scientist studies the nature of handwriting and develops a scientific theory of it that is then expressed in a computer program capable of, say, identifying handwriting, then it would seem to be the case that that computer scientist is (also) a scientific expert in handwriting.

(e) Blackmun, writing in Daubert at II.C.28, suggests four texts of "whether a theory or technique [NB: which could include a computer program, whether or not such programs are (scientific) theories] is scientific knowledge":

   i. Testability (and falsifiability) (II.C.28): computer programs would seem to be scientific on these grounds, because they can be tested and possibly falsified, by simply running the program on a wide variety of data to see if it behaves as expected.

   ii. Peer review (II.C.29): Surely, a computer program can (and should!) be peer reviewed.

   iii. Error rate (II.C.30): It's not immediately clear what Blackmun might have in mind here, but perhaps it's something like this: A scientific

theory's predictions should be within a reasonable margin of error. To take a perhaps overly simplistic example, a polling error of 5 ± 4 points is not a very accurate ("scientific") measurement, nor is a measurement error of 5.00000 ± 0.00001 inches if made with an ordinary wooden ruler. In any case, surely a computer program's errors should be "reasonable".

iv. General acceptance (II.C.31): A computer program that is not based on a "generally accepted" scientific theory or on "generally accepted" scientific principles would not be considered scientific.

Whether or not Blackmun's four criteria are complete or adequate is not the point here. The more general point is that, whatever criteria are held to be essential to a theory's being considered scientific should also apply to computer programs that are under consideration.

7. Another good quote on the nature of science:

> ...there are important differences between the quest for truth in the courtroom and the quet for truth in the laboratory. Scientific conclusions are subject to perpetual revision. Law, on the other hand, must resolve disputes finally and quickly. (Daubert, at III.35)

Note that this applies not only to the law, but also to everyday life, as Herbert Simon was well aware in his discussions of the limits of human rationality.

8. Comments on [Simon, 1996a]:

(a) In the late 1950s, the hypothesis was advanced that human thinking is information processing, alias symbol manipulation. ...[T]hese ideas [were translated] into symbolic (nonnumerical) computer programs that simulated human mental activity at the symbolic level. The traces of these programs could be compared in some detail with data that tracked the actual paths of human thought (especially verbal protocols) in a variety of intellectual tasks, and *the programs' veracity as theories of human thinking could thereby be tested.* [Simon, 1996a, 160, my italics]

Note the slide from simulation (something that a model does) to theory (ordinarily considered as a set of statements). This is more a statement that (at least some) computer programs are scientific theories than an argument for that conclusion.

(b) An argument for it can, perhaps, be constructed from this passage:

> These programs, *which predict each successive step in behavior as a function of the current state of the memories together with the current inputs*, are theories, quite analogous to the differential equation systems of the physical sciences. [Simon, 1996a, 161–162]

Here's the reconstructed argument:

    i. Differential equation systems of the physical sciences predict successive steps in physical processes as a function of the current state together with the current inputs.

    ii. Anything that allows prediction (of successive steps in some process as a function of the current state together with the current inputs) is a theory.

    iii. Therefore, differential equation systems are theories.

    iv. Cognitive computer programs predict successive steps in human cognitive behavior as a function of the current state of the memories together with the current inputs.

    v. Therefore, they are (psychological) theories.

The point is that the reason that we consider differential equation systems to be theories is the same reason that we should consider computer programs (cognitive ones in particular, but programs in general) to be theories.

Well, maybe not *all* computer programs. Arguably, a computer program for adding two numbers or for computing income tax is not a theory. (But maybe they should be considered to be theories expressed computationally: a theory of addition in the first case, a theory of taxation in the second!)

(c) The slide from simulation to theory is intentional; Simon believes that computer programs are simultaneously both:

> Thus the digital computer provided both a means (program) for stating precise theories of cognition and a means (simulation, using these programs) for testing the degree of correspondence between the predictions of theory and actual human behavior. [Simon, 1996a, 160]

Thus, computer programs are a very special kind of theory. Not only are they statements, but they are simultaneously models—instances of the very thing that they describe. Well, perhaps not quite: They only become such instances when they are being executed.

The argument that Thagard makes that programs can't be theories because they are not sets of declarative sentences just seems parochial. They are surely sets of (imperative) statements that have the additional benefit that they can become an instance of what they describe (alternatively: that they can control a device that becomes an instance of what they describe).

This duality gives them the ability to be self-testing theories. And their precision gives them the ability to pay attention to details in a way that theories expressed in English (and perhaps theories expressed in mathematics) lack.

(d) Simon hedges a bit, however:

> ...a program was analogous to a system of differential (or difference) equations, hence could express a dynamic theory. [Simon, 1996a, 161]

> So, is it the case that a program *is* a theory? Or is merely the case that a program *expresses* a theory? Perhaps this distinction is unimportant. After all, it hardly seems to matter whether a system of equations is a theory or merely expresses a theory. (The distinction is roughly akin to that between a sentence and the proposition that it expresses.)

9.   ...computational models are better able to describe many aspects of the umiverse better than any other models we know. All scientific theories can, for example, be modeled by programs. [Knuth, 2001, 168]

Presumably, they are better because some of them can "solve" (by simulation and approximation) equations for which there are not yet any "analytic" solutions. But note that Knuth is here assuming that such computer programs *are* models (of scientific theories).

10. There's a nice definition of 'simulation' in [Peschl and Scheutz, 2001b, §1], ascribed to Shannon (but not in the cited work):

> "Simulation" understood as "the process of designing a model of a real system and conducting experiments with this model for the purpose either of understanding the behavior of the system or of evaluating various strategies (within the limits imposed by a criterion or set of criteria) for the operation of the system"

In other words, roughly, *A* simulates *B* if *A* is a model of *B* and if you experiment with *A* in order to understand *B*.

11. According to [Winsberg, 2001, S443], computational simulations are models. They are created from antecedently known scientific laws concerning some physical system (p. S444). However, this is not always the case. In cognitive science, for example, a computer program may be the best or only expression of those laws; that is, they are not necessarily antecedently known. On the other hand, even in cognitive science, it may be the case that we know the laws, at least in a statistical sense, and then we write a program to simulate them.

Interestingly, [Winsberg, 2001, S449] considers a theory to be "a family of models. A model, in this sense, is an abstract mathematical entity." This is the "semantic" view of theories. It would seem, on this view, that a theory is an abstract mathematical structure common to a "family of models". The text of a computer program, then, could be considered an expression of that structure (in much the same way that both Arabic and Roman numerals are expressions of numbers considered as structures. When the program is executed, the process is simulates (or, in some cases, *is*) the system that it models.

12. [Humphreys, 2002] suggests a view of computational models that can account for the Civil War-chess example discussed in earlier chapters: "one of the characteristic features of mathematical [including computational] models is that *the same model... can occur in, and be successfully employed in, fields with quite different subject matters* (p. S2, my italics). He goes on to say, "Let the analog computer solve one and you automatically have a solution to the other" (p. S5)—but there seems no reason not to include digital computers in this observation.

15.8   Further Sources of Information

1. Bostrum, Nick (2003), "Are You Living in a Computer Simulation?", *Philosophical Quarterly* 53(211): 243–255,
   http://www.simulation-argument.com/
   (accessed 12 January 2015).

   - Argues that if "the human species is [not] likely to go extinct before reaching a [technologically advanced] posthuman stage" and if "any posthuman civilization is…likely to run a significant number of simulations of their evolutionary history", then "we are almost certainly living in a computer simulation". We will discuss this a bit more in Ch. 20.

2. Coward, L. Andrew; & Sun, Ron (2001??), "Some Criteria for an Effective Scientific Theory of Consciousness and Examples of Preliminary Attempts at Such a Theory",
   http://www.cogsci.rpi.edu/∼rsun/coward-sun-cc2003.pdf
   (accessed 13 November 2012).

3. Downes, Stephen (1990), "Herbert Simon's Computational Models of Scientific Discovery", *PSA: Proceedings of the [1990] Biennial Meeting of the Philosophy of Science Association*, Vol. 1, pp. 97–108.

   - A critique of Simon's views on the philosophy of science in general, and of programs as theories in particular.

4. Frigg, Roman; Hartman, Stephan; & Imbert, Cyrille (eds.) (2009), Special Issue on Models and Simulations, *Synthese* 169(3): 425–626.
   http://link.springer.com/journal/11229/169/3 (accessed 17 February 2014)

5. Green, Christopher D. (2001), "Scientific Models, Connectionist Networks, and Cognitive Science", *Theory and Psychology* 11: 97—117,
   http://www.yorku.ca/christo/papers/models-TP2.htm
   (accessed 12 January 2015).

   - Analyzes the use of connectionist (or neural-network) computer programs as models of cognition, and argues that "Just because two things share some properties in common does not mean that one models the other. Indeed, if it did, it would mean that everything models everything else. There must be at least a plausible claim of some similarity in the *ways* in which such properties are realized in the model and the thing being modeled" (§IV, final paragraph)

6. Humphreys, Paul (1990), "Computer Simulations", *PSA: Proceedings of the [1990] Biennial Meeting of the Philosophy of Science Association*, Vol. 2, pp. 497–506.

7. Humphreys, Paul (2002), "Computational Models", *Philosophy of Science* 69 (September): S1–S11,
   http://ist.uap.asia/ jpira/compthink/17232634.pdf
   (accessed 12 January 2015).

8. Lane, Peter C.R.; & Gobet, Fernand (2003), "Developing Reproducible and Comprehensible Computational Models", *Artificial Intelligence* 144: 251–263.

9. Neuman, Peter G. (1993), "Modeling and Simulation", *Communications of the ACM* 36(6) (June): 124.

(a) Contains useful real-life examples of ways in which simulations (and theories) can fail to be precise models of reality.

(b) Discusses "the illusion that the *virtual* is *real*" (quoting Rebecca Mercuri).

10. Partridge, Derek; & Wilks, Yorick (eds.) (1990), *The Foundations of Artificial Intelligence: A Sourcebook* (Cambridge, UK: Cambridge University Press).

   - Especially the following two sections, containing the articles listed (many of which are available elsewhere, some online):
       - §3 ("Levels of Theory", pp. 95–118), containing:
           * Marr, David (1977), "Artificial Intelligence: A Personal View", pp. 97–107; reprinted from *Artificial Intelligence* 9: 37–48.
           * Boden, Margaret A., "Has AI Helped Psychology?", pp. 108–111.
           * Partridge, Derek, "What's in an AI Program?", pp. 112–118.
       - §4 ("Programs and Theories", pp. 119–164), containing:
           * Wilks 1990, cited above in "Recommended Readings"
           * Bundy, Alan; & Ohlsson, Stellan (198?), "The Nature of AI Principles", pp. 135-154; reprinted from *AISB Quarterly* ##47–50.
           * Simon, Thomas W., "Artificial Methodology Meets Philosophy", pp. 155–164.

11. Peschl, Markus F.; & Scheutz, Matthias (2001), "Explicating the Epistemological Role of Simulation in the Development of Theories of Cognition", in *Proceedings of the 7th Colloquium on Cognitive Science (ICCS-01)*: 274–280, http://www.academia.edu/719102/Explicating_the_epistemological_role_of_simulation_in_the_development_of_theories_of_cognition (accessed 13 November 2012).

12. Scheutz, Matthias; & Peschl, Markus (2000), "Some Thoughts on Computation and Simulation in Cognitive Science", in *Proceedings of the 6th Congress of the Austrian Philosophical Society*: 534–540, http://tinyurl.com/ScheutzPeschl2000 (first accessed 13 November 2012); http://hrilab.tufts.edu/publications/scheutzpeschl00linz.pdf (accessed 5 January 2015).

   - Argues that computer programs are good simulations (and even implementations) of cognition, but only as long as they respect "the temporal metric imposed by physics".

13. Shieh, David (interviewer) (2009), "The Trouble with Computer Simulations: Linked in with Sherry Turkle", *Chronicle of Higher Education* (March 27): A14, http://chronicle.com/article/The-Trouble-With-Computer/5386 (accessed 13 November 2012).

   - "Computer simulations have introduced some strange problems into reality."

14. Simonite, Tom (2009, August 29), "Soap Bubbles to Take the Drag out of Future Cars", *New Scientist*, http://www.newscientist.com/article/dn17706-soap-bubbles-to-take-the-drag-out-of-future-cars.html (accessed 13 November 2012).

- "Despite the increasing sophistication of computer simulations, finding ways to show complex air flows visually is critical to understanding aerodynamics...and new ways to do that in large wind tunnels are valuable. 'You cannot solve everything completely in space and time on a computer,' [Alex] Liberzon [of Tel Aviv University] told *New Scientist*. 'Simulations do not capture the full complexity of wakes and other features, which can exhibit large changes in behaviour caused by very small changes.'"

15. Turner, Ray (2010), "Programming Languages as Mathematical Theories", in Jordi Vallverdú (ed.), *Thinking Machines and the Philosophy of Computer Science: Concepts and Principles* (IGI Global): 66–82.

- "That computer science is somehow a mathematical activity was a view held by many of the pioneers of the subject, especially those who were concerned with its foundations. At face value it might mean that the actual activity of programming is a mathematical one. Indeed, at least in some form, this has been held. But here we explore a different gloss on it. We explore the claim that programming languages are (semantically) mathematical theories. This will force us to discuss the normative nature of semantics, the nature of mathematical theories, the role of theoretical computer science and the relationship between semantic theory and language design."

16. Tymoczko, Thomas (1979), "The Four-Color Problem and Its Philosophical Significance", *Journal of Philosophy* 76(2) (February): 57–83, http://www.thatmarcusfamily.org/philosophy/Course_Websites/Math_S08/Readings/tymoczko.pdf (accessed 15 December 2014).

- Can a computer program be (part of) a proof of a mathematical theorem?
- For a survey of critiques of Tymoczko's arguments, see [Scherlis and Scott, 1983, §3].

17. Winsberg, Eric (2001), "Simulations, Models, and Theories: Complex Physical Systems and Their Representations", Proceedings of the 2000 Biennial Meeting of the Philosophy of Science Association, Part I: Contributed Papers, *Philosophy of Science* 68(3) (September) Supplement: S442–S454.

Chapter 16

Can Computer Programs Be Verified?

"Mechanical computers should, Babbage thought, offer a means to eliminate at a stroke all the sources of mistakes in mathematical tables. ... A printed record could...be generated..., thereby eliminating every opportunity for the genesis of errors. ... Babbage boasted that his machines would produce the correct result or would jam but that they would never deceive." [Swade, 1993, 86–87].
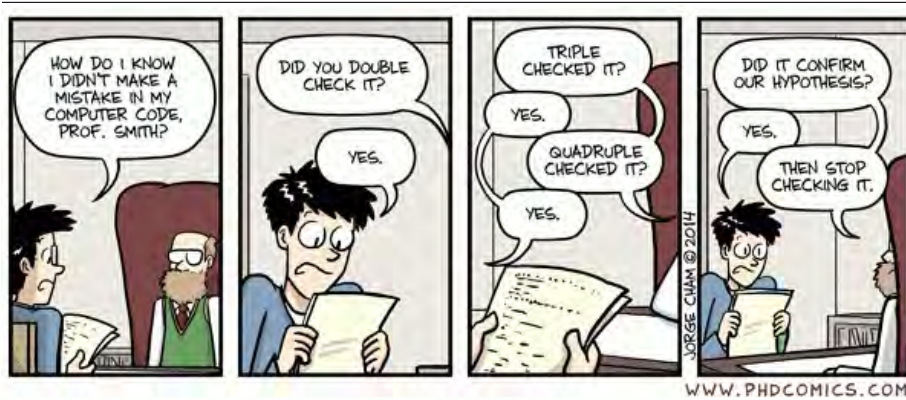
Figure 16.1: ©2014, Jorge Cham

16.1   Recommended Readings:

1. Required:

   - Fetzer, James H. (1988), "Program Verification: The Very Idea", *Communications of the ACM* 31(9) (September): 1048-1063.
     - Reprinted in [Colburn et al., 1993, 321–358].
     - A highly controversial essay!

2. Very Strongly Recommended:

   (a) De Millo, Richard A.; Lipton, Richard J.; & Perlis, Alan J. (1979), "Social Processes and Proofs of Theorems and Programs", *Communications of the ACM* 22(5): 271–280.
       - Reprinted in [Colburn et al., 1993, 297–319].
       - The "prequel" to Fetzer 1988, itself highly controversial.

   (b) Ardis, Mark; Basili, Victor; Gerhart, Susan; Good, Donald; Gries, David; Kemmerer, Richard; Leveson, Nancy; Musser, David; Neumann, Peter; & von Henke, Friedrich (1989), "Editorial Process Verification" (letter to the editor, with replies by James H. Fetzer and Peter J. Denning), ACM Forum, *Communications of the ACM* 32(3) (March): 287–290.
       - The first of many sequels to Fetzer 1988.
       - This one includes a strongly worded letter to the editor of *CACM*, signed by 10 computer scientists, protesting the publication of Fetzer 1988, a reply by Fetzer, and a self-defense by the editor.

3. Strongly Recommended:

   (a) Any of the following 4 essays that started the field of program verification. Hoare's is the most important.
       i. McCarthy, John (1963), "Towards a Mathematical Science of Computation", in C.M. Popplewell (ed.), *Information Processing 1962: Proceedings of DFIP Congress 62* (Amsterdam: North–Holland): 21–28.
          - Reprinted in [Colburn et al., 1993, 35–56].
       ii. Naur, Peter (1966), "Proof of Algorithms by General Snapshots", *BIT* 6: 310–316.
          - Reprinted in [Colburn et al., 1993, 57–64].
       iii. Floyd, Robert W. (1967), "Assigning Meanings to Programs", in *Mathematical Aspects of Computer Science: Proceedings of Symposia in Applied Mathematics*, Vol. 19 (American Mathematical Society): 19–32.
          - Reprinted in [Colburn et al., 1993, 65–81].
       iv. **Hoare, C.A.R. (1969), "An Axiomatic Basis for Computer Programming", *Communications of the ACM* 12: 576–580, 583**.
          - Reprinted in [Colburn et al., 1993, 83–96].

   (b) Dijkstra, Edsgar W. (1975), "Guarded Commands, Nondeterminacy and Formal Derivation of Programs", *Communications of the ACM* 18(8): 453–457
       - An example of program verification in action.

       [Continued on next page.]

(c) Gries, David (1981), *The Science of Programming* (New York: Springer-Verlag).

- A classic textbook. Contrast its title with that of [Knuth, 1973].

4. Recommended:

- Any of the following articles, which are representative of the battle that resulted from Fetzer 1988 and Ardis et al. 1989.

  (a) Pleasant, James C.; AND Paulson, Lawrence; Cohen, Avra; & Gordon, Michael; AND Bevier, William R.; Smith, Michael K.; & Young, William D.; AND Clune, Thomas R.; AND Savitzky, Stephen (1989), "The Very Idea" (5 letters to the editor), Technical Correspondence, *Communications of the ACM* 32(3) (March): 374–377.

  (b) Fetzer, James H. (1989), "Program Verification Reprise: The Author's Response" (to the above 5 letters), Technical Correspondence, *Communications of the ACM* 32(3) (March): 377–381.

  (c) Dobson, John; & Randell, Brian (1989), "Program Verification: Public Image and Private Reality", *Communications of the ACM* 32(4) (April): 420–422.

  (d) Müller, Harald M.; AND Holt, Christopher M.; AND Watters, Aaron (1989), "More on the Very Idea" (3 letters to the editor, with reply by James H. Fetzer), Technical Correspondence, *Communications of the ACM* 32(4) (April): 506–512.

  (e) Hill, Richard; AND Conte, Paul T.; AND Parsons, Thomas W.; AND Nelson, David A. (1989), "More on Verification" (4 letters to the editor), ACM Forum, *Communications of the ACM* 32(7) (July): 790–792.

  (f) Tompkins, Howard E. (1989), "Verifying Feature-Bugs" (letter to the editor), Technical Correspondence, *Communications of the ACM* 32: 1130–1131.

- **Barwise, Jon (1989), "Mathematical Proofs of Computer System Correctness",** *Notices of the American Mathematical Society* **36: 844–851.**

  - A cooler head prevails. Besides being an admirably clear and calm summary of the Fetzer debate, it discusses:

    (a)

    (b) the relation between algorithms and programs

    (c) the possibility of finding fault with an argument yet believing its conclusion (see §A of this book)

    (d) the nature of "philosophy of *X*" (see §2.8 of this book)

    (e) the difference between the truth of a premise and agreeing with it (see §A of this book)

    (f) the relation of math to the world

    (g) the nature of models, and the difference between models and the "real thing"

    (h) [Smith, 1985], to be discussed in Ch. 17.

  - See also:
    Dudley, Richard (1990), "Program Verification" (letter to Jon Barwise (ed.), Computers and Mathematics column, with a reply by Barwise), *Notices of the American Mathematical Society* 37: 123–124.

- **Fetzer, James H. (1998), "Philosophy and Computer Science: Reflections on the Program Verification Debate", in Terrell Ward Bynum & James H. Moor (eds.),** *The Digital Phoenix: How Computers Are Changing Philosophy, Revised Edition* **(Oxford: Blackwell, 2000): 253–273.**

    – The last word?

16.2 Introduction

Is computer programming like mathematics? Or is it more like engineering? After all, many people identify computer programming with "software engineering". Yet many others think of a program as being like a mathematical proof: a formal structure, expressed in a formal language. For example, [Suber, 1997a] compares programs to proofs this way: A program's input is analogous to the axioms used in a proof; the program's output is analogous to the theorem being proved; and the program itself is like the rules of inference that transform axioms into theorems, with the program transforming the input into the output.[1]

Or perhaps a program is more like the endpoint of a proof, namely, a mathematical theorem. In that case, just as theorems can be proved (and, indeed, must be proved before they are accepted), perhaps programs can be proved (and, perhaps, should be proved before they are used). Can we prove things about programs? What kinds of things might be provable about them?

Two answers have been given to the first of these questions: yes and no. (Did you expect anything else?) One of the most influential proponents of the view that programs *can* be proved is Tony Hoare:

> Computer programming is an exact science in that all the properties of a program and all the consequences of executing it *in any given environment* can, in principle, be found out from the text of the program itself by means of purely deductive reasoning. [Hoare, 1969, 576, my emphasis]

> When the correctness of a program, its compiler, and the hardware of the computer have all been established with mathematical certainty, it will be possible to place great reliance on the results of the program, and predict their properties with *a confidence limited only by the reliability of the electronics*. [Hoare, 1969, 579, my emphasis]

> I hold the opinion that the construction of computer programs is a mathematical activity like the solution of differential equations, that programs can be derived *from their specifications* through mathematical insight, calculation, and proof, using algebraic laws as simple and elegant as those of elementary arithmetic. [Hoare, 1986, my emphasis]

And among those arguing that programs are *not* like mathematical proofs are the computer scientists Richard De Millo, Richard Lipton, and Alan Perlis:

> ...formal verifications of programs, no matter how obtained, will not play the same key role in the development of computer science and software engineering as proofs do in mathematics. [De Millo et al., 1979, 271]

And the philosopher James Fetzer argues that the things that we *can* prove about programs are not what we think they are:

---

[1]Suber makes a stronger claim, that computers are physical implementations of formal systems. This would require the rules of inference to satisfy the constraints of being an algorithm, but not all formal systems require that.

> ...there are reasons for doubting whether program verification can succeed as a generally applicable and completely reliable method for guaranteeing the performance of a program. [Fetzer, 1988, 1049]

16.3   Program Verification

"Program verification" is a subdiscipline of computer science. It can be thought of as theoretical software engineering, or the study of the logic of software.

The idea behind it is to augment, or annotate, each statement *S* of a program with:

1. a proposition *P* expressing a "pre-condition" of executing *S*, and

2. a proposition *Q* expressing a "post-condition" of executing *S*.

By 'proposition' is meant what computer scientists call a "Boolean statement", that is, a statement that is either true or else false. Note that a mathematical proof consists of propositions (premises and conclusions) that are Boolean statements, whereas the proofs themselves (the sequences of propositions beginning with axioms and ending with a theorem) are not Boolean-valued: They are neither "true" nor "false"; rather, they are more like programs, in that they are either "correct" (the technical terms are 'valid' and 'sound') or "incorrect" (technically, 'invalid' or 'unsound'). A proof (or a logical argument) is *valid* iff it is "truth preserving", that is, iff its conclusion must be true *if* its premises are true. But, of course, the premises of an argument might not be true. If they *are* true, *and if* the argument is valid, then its conclusion must be true. Such an argument is said to be *sound*; it is *unsound* iff either one or more of its premises is false or it is invalid. Roughly, a valid argument that is unsound because of a false premise is like a correct program whose input is "garbage"; the output of such a program is also "garbage" (this is the famous saying: "garbage in, garbage out"). (For more on this, see Appendix A.)

A "pre-condition" of a statement *S* is a situation (either in the world in which the program is being executed or in the computer that is executing the program) that must be true in order for *S* to be able to be executed; that is, the pre-condition must be true *before S* can be executed.

A "post-condition" of *S* is a situation that will necessarily be true *after S* is executed.

Such annotations are typically written as comments preceding and following *S* in the program; if comments are signaled by braces, then the annotation would be written as follows:

$$\{P\}\ S\ \{Q\}$$

Such an annnotation is semantically interpreted as saying:

> If *P* correctly describes the state of the computer (or the state of the world) before *S* is executed,

> and if *S* is executed,

> then *Q* correctly describes the state of the computer (or the state of the world) after *S* is executed.

The "state of the computer" includes such things as the values of all registers (that is, the values of all variables).

So, if we think of a program as being expressed by a sequence of executable statements:

begin $S_1, S_2, \ldots, S_n$ **end.**

then the program annotated for program verification will look like this:

begin $\{I \,\&\, P_1\}S_1\{Q_1\}$, $\{P_2\}S_2\{Q_2\}, \ldots, \{P_n\}S_n\{Q_n\}$, $\{O\}$ **end.**

where:

- I is a proposition describing the input,

- P_1 is a proposition describing the initial state of the computer (or the world),

- Q_n is a proposition describing the final state of the computer (or the world),

- O is a proposition describing the output, and

- for each i, Q_i logically implies P_{i+1} and Q_n logically implies O. (Often, $Q_i = P_{i+1}$.)

The claim of those who believe in the possibility of program verification is that we can then logically prove whether the program does what it's supposed to do *without having to run the program*. We would construct a proof of the program as follows:

premise:	The input of the program is I.
premise:	The initial state of the computer is P_1.
premise:	If the input is I and the initial state is P_1,
	and if S_1 is executed,
	then the subsequent state will be Q_1.
premise:	S_1 is executed.
conclusion:	$\therefore$ The subsequent state is Q_1.
premise:	If Q_1, then P_2.
conclusion:	$\therefore P_2$.
premise:	If the current state is P_2,
	and if S_2 is executed,
	then the subsequent state will be Q_2.
conclusion:	$\therefore$ The subsequent state is Q_2.
$\ldots$	$\ldots$
conclusion:	$\therefore$ The final state is Q_n.
premise:	If Q_n, then O.
conclusion:	$\therefore O$.

(If the program isn't a "straight-line" program such as this, but is a "structured" program with separate modules, then it can be recursively verified by verifying each module.)

If debugging a program by running it, and then finding and fixing the bugs, is part of practical software engineering, then you can see why program verification can be thought of as *theoretical* software engineering.[2]

Moreover, this annotation technique can also be used to help *develop* programs that would thereby be guaranteed to be correct, another reason why program verification is argued to be an important part of software engineeing [Gries, 1981].

16.4   The Fetzer Controversy

Nonsense!, said [Fetzer, 1988], thus initiating a lengthy controversy in the pages of the *Communications of the ACM* and elsewhere, with strongly worded letters to the editor chastising the editor for publishing Fetzer's paper, supportive letters to the editor praising the decision to publish, and articles in other journals attempting to referee between the publish-or-not-to-publish controversy as well as the more substantive controversy over whether programs can, or should, be verified.

In Fetzer's terminology, let

$$\langle A_1, \ldots, A_n, P_1, \ldots, P_m, S_1, \ldots, S_l \rangle$$

be a sequence of propositions, where:

- the A_i are "axioms", that is, propositions that are *necessarily* true by virtue of their meanings (or assumed to be true for the time being),

- the P_j are "premises", that is, propositions that are *contingently* or *empirically assumed* to be true (but that would normally require some justification), and

- the S_k are other propositions.

Then to say that this sequence is a (valid) *demonstration* (or "proof") of S_l from the axioms and premises means (by definition) that each S_k follows from previous propositions in the sequence by a (valid) rule of inference.

This is a fairly standard definition of a logical proof of a theorem S_l. Note that if $n = 0$ (that is, if there are no axioms, and, especially, if $m = 0$ also—that is, if there are no premises), then there will typically have to be lots of rules of inference, and the demonstration is said to be done by "natural deduction" (because it is the way that logicians "naturally" prove things [Pelletier, 1999]).

Now, again following Fetzer's terminology,

---

[2]On the history of the term 'bug', see [Hopper, 1981], [Krebs and Thomas, 1981]. However, the idea that the first computer bug was really a bug (actually, a moth), is an urban legend, because the term was used in the non-entomological sense as early as 1889; see [Shapiro, 1985] and the *OED* online at:
http://www.oed.com/view/Entry/24352?rskey=qWlUWg&result=2#eid (accessed 7 November 2013).)
For a photo of the allegedly first "bug", see http://www.catb.org/jargon/html/B/bug.html
(accessed 6 November 2013).

- S_l is *abolutely verifiable* $=_{def} m = 0$

- S_l is *relatively verifiable* $=_{def} m > 0$

In either case, S_l is "verifiable relative to" the premises.[3] Note that what Fetzer calls 'absolute verifiability' is still verifiability except that the verifiability is relative to the *axioms* (not to the premises).

Given all of this terminology, Fetzer phrases the fundamental question of program verification this way: *Are programs absolutely verifiable?* (One question that you should keep in mind as you read the papers involved in this controversy is this: *Do* the pro-verificationists claim that programs are absolutely verifiable, in Fetzer's terminology?)

To be "absolutely verifiable" requires there to be program rules of inference that are themselves justified or truth-preserving, or it requires there to be program axioms that are necessarily true about "the performance that a machine will display when such a program is executed" [Fetzer, 1988, 1052].

But note that this is different from program verification in the Hoare-Gries tradition, because it conflates *logical* relations with *causal* relations. The former are abstract; the latter are part of the real world. Consider these analogies:

A1 A proof is to a theorem as verification is to a program: You prove theorems; you verify programs.

A2 Axioms, premises, and rules of inference are to theorems as input and programs are to outputs. Therefore, perhaps, verifying a program is like proving a theorem. (This can be called the "Suber-Fetzer" analogy.) But on this analogy, verifying a program is more like proving the inference rules! So, analogy (A1) seems better.

A3 Axioms, premises, and intermediate conclusions are to theorems as input and program are to output. Both intermediate conclusions and program are a sequence (or at least a set) of expressions that begin with something given and end with a desired result. But you are trying to *prove* the theorem; you are not trying to prove the intermediate conclusions. And you *verify* an entire program, not just its output.

Moreover, in favor of [A1], most theorems are of the form "if P, then Q"; most programs can be put in the form "if you do $S_1, \ldots, S_n$, then you'll accomplish goal G".

Fetzer claims that theorems are *not* like programs, because only programs have causal and physical interpretations as their *intended* interpretation. And this is where the main problem arises, in Fetzer's point of view: What about program statements that specify output behaviors, such as ringing a bell (some programing languages have a command BEEP whose intended behavior is to ring a bell) or drawing a circle (suppose you have a programming language one of whose legal instructions is DRAW_CIRCLE(x, y, r), whose intended behavior is to draw a circle at point (x, y) with radius r). How can you prove or verify that the program will do this? How can

---

[3] In fact, the conclusion of a formal proof—a theorem—is only true *relative to* the axioms of the formal theory. For discussion, see [Rapaport, 1984, 613].

you mathematically or *logically* prove that the (physical) bell will (actually) ring or that a (physical) circle will (actually) be drawn? How can you *logically* prove that the (*physical* bell *works* or that the pen has ink in it? Fetzer's point is that you can't. (And the controversy largely focused on whether that's what's meant by program verification.) This should remind you of the discussion in Ch.10 about Cleland's interpretation of the Church-Turing Computability Thesis: Is preparing Hollandaise sauce, or physically ringing a bell, or physically drawing a circle a computable task?

But, according to Fetzer, it's not just ringing bells, drawing circles, or, for that matter, cooking that's at issue. What about the PRINT command that is required of all Turing-machine programs? And, according to Fetzer, it's not just a matter of causal output, because you can replace every PRINT(x) command with an assignment statement: $p := x$. Even this is a causal statement, because it instructs the computer to change the values of bits in a physical register p, and so Fetzer's argument goes through: How can can you *logically* prove that the *physical* computer will actually work? Indeed, the history of early modern computers was largely concerned with ensuring that the vacuum tubes would be reliable [Dyson, 2012b].

The point, according to Fetzer, is that we must distinguish between the program and the algorithm that it implements. The former implements the latter: A program is a causal model of a logical structure, and, while algorithms might be capable of being absolutely verified, programs cannot.

It might be replied, on behalf of the pro-verificationists, that we can still do *relative* verification: verification relative to "causal axioms" that relate these commands to causal behaviors. So, we can say that, *if* the computer executing program P is in good working order, and *if* the world (the environment surrounding the computer) is "normal", *then P* is verified to behave in accordance with its specifications.

No, says Fetzer: Algorithms and programs that are only intended for abstract machines *can* be *absolutely* verified (because there is nothing physical about such machines; they are purely formal). But programs that can be compiled and executed can only be *relatively* verified.

16.5   Summary

The bottom line is that programs of the latter type need causal rules of inference of the form: input I causes output O. Perhaps the BEEP command would have to be annotated something like this:

{The bell is in working order.} BEEP {A sound is emitted.}

If such causal rules are part of the definition of an abstract machine, then we *can* have absolute verification of the program. But if they are merely empirical claims, then we can only have relative verification of the program.

Even so, absolute verification is often thought to be too tedious to perform and can lure us into overconfidence. The former problem seems to me not to be overly serious; it's tedious to prove theorems in mathematics, too; in any case, techniques are being devised to automate program verification. The latter problem is more important, for

precisely the reasons that Fetzer adduces. Just because you've proved that a program is correct is no reason to expect that the computer executing it will not break down.

The general tone of the responses to Fetzer included these objections:

- How dare you publish this! [Ardis et al., 1989]

- So what else is new? We program verificiationists never claimed that you could logically prove that a physical computer would not break down.

- Verification techniques *can* find logical faults; it is logically possible to match a program or algorithm to its specifications.

- You can minimize the number of IcO (??) rules such that they only apply to descriptions of logic gates and the physics of silicon.

- Many programs are incorrect because, for example of the limits of accuracy in using real numbers.

- Verifiably *incorrect* programs can be better than verifiably correct programs *if* they have better average performance (cf. [Moor, 1979]).

16.6   NOTES FOR NEXT DRAFT

1. Each proposition of Euclid's *Elements* consisted of an algorithm (for constructing a geometric figure using only compass and straightedge) and a proof of correctness.   See, e.g., the statement of Proposition 1 at http://tinyurl.com/kta4aqh (accessed 17 February 2014).[4]

2. Here's another way to express Fetzer's point: Even if program verification techniques can prove that a program is correct, there may still be *performance* limitations. So, Fetzer's point may be along the lines of Chomsky's competence-performance distinction.

3.   A Turing machine is like an actual digital computing machine, except that (1) it is error free (i.e., it always does what its table says it should do), and (2) by its access to an unlimited tape it is unhampered by any bound on the quantity of its storage of information or "memory". [Kleene, 1995, 27]

   The former limitation of real Turing machines does not obviate the need for program verification. Even an "ideal" Turing machine could be poorly programmed.

4.   The remarks. . . on the desired automatic functioning of the device [that is, von Neumann's definition of a computer, as quoted in Ch. 9, §19] must, of course, assume that it functions faultlessly. Malfunctioning of any device has, however, always a finite probability—and for a complicated device and a long sequence of operations it may not be possible to keep this probability negligible. Any error may vitiate the entire output of the device.   For the recognition and correction of such malfunctions intelligent human intervention will in general be necessary.

         However, it may be possible to avoid even these phenomena to some extent.   The device may recognize the most frequent malfunctions automatically, indicate their presence and location by externally visible signs, and then stop.   Under certain conditions it might even carry out the necessary correction automatically and continue. [von Neumann, 1945, §1.4, p. 1].

   One way to read this is as a recognition/anticipation of Fetzer's point. Given this inevitability, the focus presumably has to be on the elimination of *logical* errors, so that program verification still has a role to play. The second paragraph suggests that some *machine* "verification" might be automated, but that just leads to an endless regress: Even if the logical structure of that automation is guaranteed, the physcial device that carries it out will itself be subject to some residual malfunction possiblities.

---

[4]http://www.perseus.tufts.edu/hopper/text?doc=Perseus%3Atext%3A1999.01.0086%3Abook%3D1%3Atype%3DProp%3Anumber%3D1

Of course, another way to read this (as well as the entire program verification debate) is to recognize that no one, and no thing, is perfect. There's always the chance of error or malfunction, and the point is, at least, to minimize it (if complete elimination is physically impossible).

5. The entire issue of program verification might be considered as a subset of the more general engineering issue of reliability. ([Newell, 1980, 159], for example, assumes that a symbol system should be "totally reliable, nothing in its organization reflecting that its operators, control or memory could be erroful" [sic!]. He goes on to say that "universality is always relative to physical limits, of which reliability is one" (p. 160), where 'universality' is defined as the ability to "produce an arbitrary input-output function" (p. 147). This suggest that, even if a program could be proved mathematically to be correct, the process that executes it would still be limited by physical correctness, so to speak, and that, presumably, cannot be mathematically proved. (Note that this point was already made in 1980.)

Reliability is also related to what [Staples, 2014, Staples, 2015] considers to be the heart of engineering knowledge, namely, how well artifacts (including computer programs) meet their specifications (see the discussion in Ch. 5, §5.7 at 22).) [Carhart, 1956, 149] argues that it is not sufficient just to look at the physical components of a complex system, but that the interactions of the components are also important. He calls for "a total systems approach" that would include not only the physical components (hardware) but also the people who operate it (which he calls 'software'—recall our discussion in §12.4 at 10).

Adapting this to the Fetzer controversy, we might want to say that it is not sufficient merely to verify the static computer program (and its components), but also the way that program is dynamically executed.

6. Put this quote at the beginning of this chapter? Or of Ch. 10?

> "I should like to ask the same question that Descartes asked. You are proposing to give a precise definition of logical correctness which is to be the same as my vague intuitive feeling for logical correctness. How do you intend to show that they are the same?" …[T]he average mathematician…should not forget that intuition is the final authority…. –[Rosser, 1978, 4,11], slightly incorrectly cited in [De Millo et al., 1979, 271]

[De Millo et al., 1979] cite this presumably to support their argument that formal proofs don't always yield acceptance or belief. (My college friend Alan Arkawy always advised that you had to believe a mathematical proposition before you could try to prove it.)

Note the similarity to the CTCT; this is not accidental. The "Descartes" cited in the quote is not the real Descartes, but a fictional version visited by a time-traveling mathematician who tries to convince him that the modern and formally precise, δ-ε definition of a continuous curve is equivalent to "Descartes"'s intuitive definition in terms of being able to be drawn without lifting pencil from paper. They agree that, despite the informality of "Descartes"'s definition and the formality of the mathematician's, they can be "verified"—but not "proved"—to be equivalent by seeing that they agree on a wide variety of cases. The cited quotation is the mathematician's response to a logician who tries to convince the mathematician that the mathematician's intuitive notion of proof bears the same relation to "Descartes"'s intuitive notion of continuity that the logician's formal and logical definition of proof bears to the mathematician's intuitive notion of proof.

7. Comments on [Scherlis and Scott, 1983]:

   (a) They argue that program verification and development should go hand-in-hand, rather than verification coming after a program is complete. Others have made the same point (e.g., Dijkstra, Gries: CITE), but Scherlis & Scott's notion—"inferential programming"—differs from "program derivation". Whereas "program derivations [are] highly structured justifications for programs[,] inferential programming [is] the process of building, manipulating,and reasoning *about* program derivations" (p. 200, my italics).

   (b) A " 'correctness' proof [shows] that a program is consistent with its specifications" (p. 201), where "*Specifications* differ from programs in that they describe aspects or restrictions on the functionality of a desired algorithm without imposing constraints on *how* that functionality is to be achieved." (p. 202)

   (c) They take issue with [De Millo et al., 1979]. First, they observe that the claim that "Mathematicians do not really build formal proofs in practice; why should programmers?" (p. 204) is fallacious, because "formalization plays an even more important rôle in computer science than in mathematics", and this, in turn, because "computers do not run 'informal' programs" (p. 204). Moreover, formalization in mathematics has made possible much advancement independent of whether "there is any sense in looking at a *complete* formalizatio of a whole proof. Often there is not." (p. 204)

   (d) They advocate, not for a complete proof of correctness of a completed program, but for proofs of correctness of stages of development, together with a justification that "derivation steps *preserve* correctness". This is exactly the way in which proofs of theorems are justified: If the axioms and premises are true, and if the rules of inference are truth-preserving, then the conclusions (theorems) will be true (relative to the truth of the axioms and premises).

(e) Where [De Millo et al., 1979] claim that some view theorems and proofs as analogous to programs and verifications, respectively, but that theorems are more like specifications, proofs like programs, and formal demonstrations (were anyone to actually sit down and provide one) like verifications, [Scherlis and Scott, 1983] say that a better analogy is this: problems are like specifications, theorems are like programs, and proofs are like program derivations. (p. 207). They add that all such analogies must "be taken with a big grain of salt, since all thee words can mean many things".

8. FETZER ON LOGIC, REASONING, AND ARGUMENT ANALYSIS
Fetzer's paper makes some comments about the nature of logical reasoning and about knowledge and belief that are relevant to what you've been doing in your argument analyses.

(a) On p. 1050, column 1, he says:

> [W]hat makes ... a *proof* a proof is its validity rather than its acceptance (by us) as valid, just as what makes a sentence true is [that] what it asserts to be the case is the case, no[t] merely that it is believed (by us) and therefore referred to as *true*.

Note that I've been allowing you to evaluate the truth-value of the premises of an argument, *not* by trying to demonstrate whether they *are* true, but by trying to say whether and why *you believe* them. According to Fetzer's quote, it would follow that you are not *really* Jevaluating the premises.

He's correct! Whether a statement (or premise) *is* true or not does *not* depend on whether you (or anyone) *believes* it to be true. It is (or isn't) true iff what it states corresponds to reality.

Nevertheless, that's very hard (if not impossible) to prove. And that's why I'm allowing you to do something that is a bit easier, and a bit more realistic, and—for our purposes—just as useful, namely, to try to explain whether and why you believe the statement.

(b) In the same location, at the beginning of the next section, Fetzer says:

> Confidence in the truth of a theorem (or in the validity of an argument) ... appears to be a psychological property of a person-at-a-time....

It's that "confidence" that I've been asking you to examine, explain, and defend. Because it's a "psychological property of" *you now*, I only grade you on how well you explain and defend it, not on what it is.

(c) Finally, in column 2 on the same page, he says:

> [A]n individual z who is in a state of belief with respect to a certain formula f ... cannot be properly qualified as possessing knowledge that f is a theorem unless his belief can be supported by means of reasons, evidence, or warrants....

This is a very complicated way of making the following important point: If you *believe* a statement *f*, that belief doesn't count as *knowing* that *f* is the case *unless* you have a *reason* for your belief. In other words, knowledge is belief plus (at least) a reason. (Actually, most philosophers agree that knowledge requires a third thing: knowledge is belief, plus a reason, *plus f* being true: You can't "know" something that's false.)

This need to justify your beliefs is what turns a mere opinion, or an expression of feeling, into a claim that is worthy of holding and of convincing others of. It's why we have arguments to justify conclusions, and why we have to also justify all the premises of the arguments. (And it's why we have to justify, recursively, all the justifications, until we reach some starting point that is a self-justifying belief. But it's not clear that there really are any, which means that our investigations may never end!)

9. At the 40th Anniversary celebration of the founding of the UB Department of Computer Science & Engineering, our 2nd Ph.D., Bruce Shriver, former president of the IEEE Computer Society, said in his address that hardware does not have flaws, only software does. Note that this is almost exactly the converse of Fetzer's point!

10. Comments on Barwise 1989:

   (a) Perhaps incorporate here some my observations in the Readings section, above.

   (b) Barwise sees the issue between Fetzer and his opponents as being a special case of the more general question of how mathematics can be applied to the real world, given that the former is abstract and purely logical whereas the latter is concrete and empirical (p. 846, col. 2). (See also [Wigner, 1960] and §§4.9, 7.10, esp. §37.) This fits with other comments above about the ways in which mathematical expressions such as programs can be formally proved, whereas physical objects such as computers are subject to physical breakdown.

   (c) But there is another aspect to that issue in the philosophy of math, namely, the relation between the syntax of a formal mathematical expression and its semantic interpretation in the real world:

   > The axiomatic method says that our theorems are true *if* our axioms are. The modeling method says that our theorems model facts in the domain modeled *if* there is a close enough fit between the model and the domain modeled. The sad fact of the matter is that **there is usually no way to prove—at least in the sense of mathematical proof—the antecedent of a conditional of either of these types.** (p. 847, col. 2, italics in original, my boldface)

   And this fits in nicely with [Smith, 1985], which we'll look at in the next chapter. (He cites [Smith, 1985], noting that "Computer systems are not

just phyusical objects that compute abstract algorithms. They are also embedded in the physical world and they interact with users. . . . Thus, . . . our mathematical models need to include not just a reliable model of the computer, but also a reliable model of the environment in which it is to be placed, inluding the user." (p. 850, col. 1).

(d) Barwise notes that Fetzer is only willing to talk about the causal (that is, physical) role of computers, which is not susceptible of mathematical verification, whereas the field of program verification only concerns abstract programs. (p. 848, col. 2). So it really seems that both sides are not only talking past each other, but are actually consistent with each other.

(e) As for the issues about the nature and value of proofs as raised by [De Millo et al., 1979], Barwise observes that mathematical theories of formal proofs "aren't what mathematicians since the time of the ancient Greeks were constructing" (p. 849, col. 1). Although he doesn't say so explicitly, he very strongly suggests that the relation between such theories of formal proofs (and there are more than one) and the informal proofs "since the time of the ancient Greeks" is akin to the CTCT.

11. Somewhere note that the program verification issue is also a subissue of the question concerning the relation of software to hardware.

12. . . . mathematical reasoning alone can never establish the "correctness" of a program or hardware design in an absolute sense, but only relative to some formal specification of its desired behavior" [MacKenzie, 1992, 1066, col. 2].

Similarly, a formal proof of a theorem only establishes its truth *relative to* the truth of its axioms, not its "absolute" truth.

MacKenzie continues:

Mathematical argument can establish that a program or design is a correct implementation of that specification, but not that implementation of the specification means a computer system that is "safe", "secure", or whatever. [MacKenzie, 1992, 1066, col. 2].

There are two points to notice here. First, a mathematical argument can establish the correctness of a program relative to its specification, that is, whether the program satisfies the specification. In part, this is the first point, above. But, second, not only does this not necessarily mean that the computer system is safe (or whatever), it also does not mean that the *specification* is itself "correct". Presumably, a specification is a relatively abstract outline of the solution to a problem. Proving that a computer program is correct relative to—that is, satisfies—the specification does not guarantee that the specification actually solves the problem!

> . . . what can be proven correct is not a physical piece of hardware,
> or program running on a physical machine, but only a mathematical
> model of that hardware or program. [MacKenzie, 1992, 1066, col. 2].

This is the case for reasons that [Smith, 1985] discusses and that we will look
at in the next chapter. It is not unrelated to the CTCT: You can't show that two
systems are the same, in some sense, unless you can talk about both systems
in the same language. In the case of the CTCT, the problem concerns the
informality of the language of algorithms versus the formality of the language
of Turing machines or recursive functions. In the present case, the problem
concerns the mathematical language of programs versus the non-linguistic,
physical nature of the hardware. Only by *describing* the hardware in a (formal,
mathematical) language, can a proof of equivalence be attempted. But then we
also need a proof that that formal description is correct; and that can't be had.

13. (These comments might go better in Ch. 17, after discussing [Smith, 1985].):

   For Fetzer, computing is about the world; it is external and contextual (recall
   our discussion of Cleland, in Ch. 10 and elsewhere WHERE?). Thus, computer
   programs can't be verified, because the world may not be conducive to "correct"
   behavior: A physical part might break; the environment might prevent an
   otherwise perfectly running, "correct" program from accomplishing its task
   (such as making hollandaise sauce on the Moon using an Earth recipe); etc.

   For Smith, computing is about a model of the world; it is internal and narrow.
   Thus, computer programs can't be verified, but for a different reason, namely,
   the model might not match the world.

   Perhaps a better way of looking at things is to say that there are two different
   notions of "verification": an internal and an external one. (For related
   distinctions, see [Tedre and Sutinen, 2008, 163–164].)

14.   Software is sometimes taken to be an artefact. . . . However, is
        software mathematical, or a physical thing? . . . [T]he key point is
        as follows. Like the design of an artefact, it is the objective content
        of a computer program that is important to its characterisation and
        identity. The objective content of a program remains unchanged,
        regardless of whether it is represented as magnetic regions on a
        hard disk, or as ink on a page. . . . Nevertheless, the controversy
        and lessons. . . from Fetzer's (1988) paper on program verification
        remind us that *the execution of a program on a physical computer*
        is categorically different to the content of the program, and different
        to a characterisation of the program's possible behaviour using
        formal theories. The execution of a program on a computer
        system is a phenomena [sic][5] *in the real world, and it is this*

---

[5]It should be 'phenomenon'.

> *phenomena [sic] which might change the world in a way that satisfies requirements.*[6] Theories of program analysis or 'formal verification' are not purely mathematical—they are ultimately also empirical engineering theories when they support claims that the behavior of real programmed computer systems meet requirements. [Staples, 2014, §2, my italics]

Note how the issue that we discussed in Ch.12 about the nature of software vs. hardware is relevant to the issue of program verification. Does a formal proof of a program's "correctness" apply to the program as software or to the program as hardware (perhaps to the process that comes into existence when the program is executed)? This is also relevant to a discussion we will have in the next chapter about the relation between computers and the world.

15. Related to Smith's Gap:

> Human fallibility means some of the more subtle, dangerous bugs turn out to be errors in design; the code faithfully implements the intended design, but the design fails to correctly handle a particular "rare" scenario. [Newcombe et al., 2015, 67]

In other words, the abstraction that produced the design (the specification) was in error, and those are precisely the kinds of errors that Smith warns about.

---

[6]That is, the way that the program satisfies the requirements of its specification.

16.7 Further Sources of Information

1. Arnow, David (1994), "Teaching Programming to Liberal Arts Students: Using Loop Invariants", *SIGCSE 94, Phoenix, AZ*: 141–144.

   - How to use program-verification techniques to develop programs. Compare to Tam 1992, below.

2. Avigad, Jeremy; & Harrison, John (2014), "Formally Verified Mathematics", *Comminications of the ACM* 57(4) (April): 66–75, http://cacm.acm.org/magazines/2014/4/173219-formally-verified-mathematics/fulltext (accessed 25 February 2015).

   - "With the help of computational proof assistants, formal verification could become the new standard for rigor in mathematics" (from the introductory blurb, p. 66).
   - Discusses the history and nature of formal proofs in math.

3. Blum, Bruce I. (1989), "Formalism and Prototyping in the Software Process", *Information and Decision Technologies* 15: 327–341; reprinted in [Colburn et al., 1993, 213–238].

4. Bowen, J.P.; & Hinchey, M.G. (1995), "Ten Commandments of Formal Methods", *IEEE Computer* 28(4): 56–63.

5. Burkholder, Leslie (1999), "Are AI and Mechanics Empirical Disciplines?", *Journal of Experimental and Theoretical Artifical Intelligence* 11: 497–500.

   - Uses Fetzer 1988 and Barwise 1989 to argue that AI is an empirical science.

6. Clarke, Edmund M.; Emerson, E. Allen; & Sifakis, Joseph (2010), "Model Checking: Algorithmic Verification and Debugging", *Communications of the ACM* 52(11) (November): 74–84.

7. Cohen, Avra (1989), "The Notion of Proof in Hardware Verification", *Journal of Automated Reasoning* 5: 127–139; reprinted in [Colburn et al., 1993, 359–374].

8. Colburn, Timothy R. (1991), "Program Verification, Defeasible Reasoning, and Two Views of Computer Science", *Minds and Machines* 1: 97–116; reprinted in [Colburn et al., 1993, 375–399].

9. Colburn, Timothy R. (1993), "Computer Science and Philosophy", in [Colburn et al., 1993, 3–31].

10. Colburn, Timothy R.; Fetzer, James H.; & Rankin, Terry L. (eds.) (1993), *Program Verification: Fundamental Issues in Computer Science* (Dordrecht, Holland: Kluwer Academic Publishers).

    - An anthology containing many of the papers discussed or listed in this chapter, as well as a bibliography that is even longer than this one!

11. Davis, Phillip J.; & Hersh, Reuben (1998), "The Ideal Mathematician", in their *The Mathematical Experience* (Boston: Houghton Mifflin): 34–44, http://babel.ls.fi.upm.es/~pablo/Jumble/ideal.pdf (accessed 5 March 2015).

    - A partly facetious description of the behavior of (some) mathematicians, including a discussion of the nature of proof as carried out by mathematicians.
    - For an antidote to their characterization of mathematicians, read [Frenkel, 2013].

12. Devlin, Keith (1992), "Computers and Mathematics" (column), *Notices of the American Mathematical Society* 39(9) (November): 1065–1066.

- Contains   a
  dialogue between a math professor who defends [De Millo et al., 1979]'s notion of "social" proof and a philosophy student who defends the more formal logical notion of proof. Serves as an introduction to [MacKenzie, 1992].

13. Dewar, Robert (2009), "CS Education in the U.S.: Heading in the Wrong Direction? Point", *Communications of the ACM* 52(7) (July): 41–43.

- Argues in favor of teaching formal reasoning in computer science, including program-verification techniques.

- For an opposing viewpoint, see:
  Astrachan, Owen (2009), "CS Education in the U.S.: Heading in the Wrong Direction? Counterpoint", *Communications of the ACM* 52(7) (July): 43–45.

14. Dijkstra, Edsgar W. (1974), "Programming as a Discipline of Mathematical Nature", *American Mathematical Monthly* (June-July): 608–612.

- Argues "that the correctness of programs could and should be established by proof", that structured programs are simpler to prove than unstructured ones [Dijkstra, 1968], that theorems about programs make program proofs easier, and that "to prove the correctness of a given program was. . . putting the cart before the horse. A much ore promising approach turned out to be letting correctness proof and program grow hand in hand" (pp. 609–610).

15. Dijkstra, Edsger W. (1983), "Fruits of Misunderstanding" (EWD-854), reprinted in *Datamation* (15 February 1985): 86–87,
    http://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD854.html
    (accessed 13 November 2012).

16. Fetzer, James H. (1991), "Philosophical Aspects of Program Verification", *Minds and Machines* 1: 197–216; reprinted in [Colburn et al., 1993, 403–427].

- A summary, reply to objections, and further discussion of the relation of software to hardware.

17. Fetzer, James H. (1993), "Program Verification", in Allen Kent & James G. Williams (eds.), *Encyclopedia of Computer Science and Technology*, Vol. 28, Supp. 13 (New York: Marcel Dekker): 237–254; reprinted in Allen Kent & James G. Williams (eds.), *Encyclopedia of Microcomputers*, Vol. 14: Productivity and Software Maintenance: A Managerial Perspective to Relative Addressing (New York: Marcel Dekker): 47–64.

- An encyclopedia-style article on program verification, but written from Fetzer's perspective.

18. Fetzer, James H. (1996), "Computer Reliability and Public Policy: Limits of Knowledge of Computer-Based Systems", *Social Philosophy and Policy* 13(2) (Summer): 229–266.

- ". . . in the real world, the operation of computer systems is inherently uncertain, which raises profound problems for public policy." (from the Abstract)

19. Fetzer, James H. (1998), "Computer Systems: The Uncertainty of their Reliability", *Bridges* 5(3/4): 197–217.

- "The reliability of computer systems...depends on the...interaction of hardware and software...and the accuracy and completeness of the knowledge upon which they are based. ...The reliability of computer-based systems is necessarily uncertain...[and] must not be taken for granted." (from the Abstract)

20. Floyd, Christiane (1987), "Outline of a Paradigm Change in Software Engineering", in G. Bjerknes et al. (eds.), *Computers and Democracy: A Scandinavian Challenge* (Brookfield, VT: Gower Publishing): 191–210; reprinted in [Colburn et al., 1993, 239–259].

21. Floyd, R.W. (1967), "Assigning Meaning to Programs", *Mathematical Aspects of Computer Science (Proceedings of Symposia in Applied Mathematics 19)* (American Mathematical Society): 19–32; reprinted in [Colburn et al., 1993, 65–81].

22. Frenkel, Karen (1993), "An Interview with Robin Milner", *Communications of the ACM* 36(1) (January): 90–97,
http://delivery.acm.org/10.1145/160000/151241/a1991-frenkel.pdf
(accessed 26 May 2014).

    - A companion piece to [Milner, 1993], with interesting observations on AI, the semantics of programming languages, program verification, and the nature of computer science.

23. Glass, Robert L. (2002), "The Proof of Correctness Wars", *Communications of the ACM* 45(8) (August): 19–21.

    - A historical survey, with some useful references, arguing that the controversies over program verification are "extremely healthy for the field" of computing (roughly for the same reasons that, it was argued in Ch. 2, philosophy is important: the challenging of assumptions).

24. Glanz, James (1995), "Mathematical Logic Flushes Out the Bugs in Chip Designs", *Science* 267: 332–333.

25. Henzinger, Thomas (1996), "Some Myths about Formal Verification", *ACM Computing Surveys* 28(4es) (December), article no. 119.

    - A brief "position statement" in favor of formal verification.

26. Hinchey, Mike; Jackson, Michael; Cousot, Patrick; Cook, Byron; Bowen, Jonathan P.; & Margaria, Tiziana (2008), "Software Engineering and Formal Methods", *Communications of the ACM* 51(9) (September): 54–59.

    - A statement of the argument in favor of program verification.

27. Hoare, C.A.R. (1986), "Mathematics of Programming", *Byte* (August); reprinted in [Colburn et al., 1993, 135–154].

    - A summary of Hoare's view "that the construction of computer programs is a mathematical activity..., that programs can be derived from their specifications through mathematical insight, calculation, and proof..." because "Computers are mathematical machines. ...Computer progras are mathematical expressions. ...A programming language is a mathematical theory..." even though "Nothing is as I have described it".

28. Hoare, C.A.R. (2009), "Retrospective: An Axiomatic Basis for Computer Programming", *Communications of the ACM* 52(10) (October): 30–32.

29. Humelsine, Jim (2010), "Software Still as Much an Art as Science", Letters to the Editor, *Communications of the ACM* 53(1) (January): 7.

30. Lamport, Leslie (2015), "Who Builds a House without Drawing Blueprints?", *Communications of the ACM* 58(4): 38–41, http://cacm.acm.org/magazines/2015/4/184705-who-builds-a-house-without-drawing-blueprints/fulltext (accessed 14 June 2015).

   - Argues that "the main reason for writing a formal spec[ification] is to apply tools to check it", that "the math needed for most specifications is quite simple: predicate logic and elementary set theory", that "a specification can and should be embedded as a comment within the code it is specifying", and—perhaps most importantly— that "thinking does not guarantee that you will not make mistakes. But not thinking guarantees that you will."

31. Leroy, Xavier (2009), "Formal Verification of a Realistic Compiler", *Communications of the ACM* 52(7) (July): 107–115, preprint at http://gallium.inria.fr/~xleroy/publi/compcert-CACM.pdf (accessed 25 February 2015).

   - See also the introductory editorial: Morrisett, Greg (2009), "A Compiler's Story", *Communications of the ACM* 52(7) (July): 106, https://www.deepdyve.com/lp/association-for-computing-machinery/technical-perspective-a-compiler-s-story-30zQuKanaE (accessed 25 February 2015), containing philosophical remarks on the value of program verification.

32. MacKenzie, Donald (1992), "Computers, Formal Proofs, and the Law Courts", *Notices of the American Mathematical Society* 39(9) (November): 1066–1069.

   - Discusses a legal challenge to a claim that a certain computer program had been verified. The claim was that the verification was of the relatively informal, [De Millo et al., 1979]-style variety of proof; the challenge was that a formal, mathematical proof was necessary..

33. MacKenzie, Donald (2001), *Mechanizing Proof: Computing, Risk, and Trust* (Cambridge, MA: MIT Press).

   - Ch. 6 ("Social Processes and Category Mistakes") concerns the [De Millo et al., 1979]-[Fetzer, 1988] controversy.

   - For a review, see: Hayes, Brian (2002), "The Search for Rigor", *American Scientist* 90 (July-August): 382–384, http://www.americanscientist.org/bookshelf/pub/the-search-for-rigor (accessed 5 March 2015).

34. Meyer, Bertrand (1985), "On Formalism in Specifications", *IEEE Software* 2(1) (January): 6–26; reprinted in [Colburn et al., 1993, 155–189].

35. Mili, Ali; Desharnais, Jules; & Gagné, Jean Raymond (1986), "Formal Models of Stepwise Refinement of Programs", *ACM Computing Surveys* 18(3) (September): 231–276.

- §1 contains a formal presentation of program correctness.

36. Myers, Ware (ed.) (1986), "Can Software for the Strategic Defense Initiative Ever Be Error-Free?", *IEEE Computer* (November): 61–67.

37. Naur, Peter (1982), "Formalization in Program Development", *BIT* 22: 437–453; reprinted in [Colburn et al., 1993, 191–210].

38. Naur, Peter (1992), "The Place of Strictly Defined Notation in Human Insight", in *Computing: A Human Activity* (Addison-Wesley); reprinted in [Colburn et al., 1993, 261–274].

39. Nelson, David A. (1992), "Deductive Program Verification (A Practitioner's Commentary)", *Minds and Machines* 2: 283–307.

40. Nelson, David A. (1993), Review of Boyer & Moore's *A Computational Logic Handbook* and Moore's Special Issue on System Verification (*Journal of Automated Reasoning*), in *Minds and Machines* 4: 93–101.

41. Neumann, Peter (1996, July), "Using Formal Methods to Reduce Risks", *Communications of the ACM* 39(7): 114.

42. Olson, Steve (1984, January/February), "Sage of Software", *Science*: 75–80.

43. Pollack, Andrew (1999, May 3), "For Coders, a Code of Conduct: 2000 Problem Tests Professionalism of Programmers", *The New York Times*: C1, C12; http://tinyurl.com/Pollack1999.[7] (accessed 14 November 2012).

44. Scherlis, William L.; & Scott, Dana S. (1983), "First Steps towards Inferential Programming", in R.E.A. Mason (ed.), *Information Processing* 83 (JFIP and Elsevier Science Publishers): 199–212; reprinted in [Colburn et al., 1993, 99–133].

45. Shustek, Len (ed.), "An Interview with C.A.R. Hoare", *Communications of the ACM* 52(3) (March): 38–41.

46. Stewart, Ian (2007), "The Future of Proof", *Prospect* (22 February).

- On the nature of mathematical proof, written for a general audience.

47. Tam, Wing C. (1992), "Teaching Loop Invariants to Beginners by Examples", *SIGCSE Bulletin* 24(1) (March): 92–96.

- How to use program-verification techniques to develop programs. Compare to Arnow 1994, above.

48. Tedre, Matti (2007), "Know Your Discipline: Teaching the Philosophy of Computer Science", *Journal of Information Technology Educaiton* 6:105–122.

- "...one cannot formally prove either that an engineered product has the intended qualities or that an engineered product will not fail..." (p. 108)

49. **Turing, Alan M. (1949), "Checking a Large Routine", in** *Report of a Conference on High Speed Automatic Calculating Machines* **(Cambridge, UK: University Mathematics Lab): 67–69**; reprinted in Morris, F.L.; & Jones, C.B. (1984), "An Early Program Proof by Alan Turing", *Annals of the History of Computing* 6(2) (April): 139–143.

---

[7]http://www.nytimes.com/1999/05/03/business/for-coders-a-code-of-conduct-2000-problem-tests-professionalism-of-programmers.html

- This "paper... is remarkable in many respects. The three... pages of text contain an excellent motivation by analogy, a proof of a program with two nested loops, and an indication of a general proof method very like hat of Floyd [1967]" (Morris & Jones 1984: 139).

50. Verity, John W. (1985), "Bridging the Software Gap", *Datamation* (15 February): 84–88.

- Contains interviews with Dijkstra, Hoare, and Gries, and includes a copy of Dijkstra's essay, "Fruits of Misunderstanding" (EWD-854, also online at https://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD854.html, accessed 16 February 2015).

51. For interesting contrasting views of the relationship between computer programs and mathematical proofs, see:

(a) Suber, Peter (1997), "Formal Systems and Machines: An Isomorphism", http://www.earlham.edu/~peters/courses/logsys/machines.htm (accessed 9 November 2012).

(b) Thurston, William P. (1994), "On Proof and Progress in Mathematics", *Bulletin of the American Mathematical Society* 30(2) (April): 161–177.

Chapter 17

How Do Programs Relate to the World?

Today, computing scientists face their own version of the mind-body problem: how can vitual software interact with the real world? [Wadler, 1997, 240]

17.1   Required Reading:

- Smith, Brian Cantwell (1985), "Limits of Correctness in Computers", *ACM SIGCAS Computers and Society* 14-15 (1-4) (January): 18–26.

  Also published as:

  - *Technical Report CSLI-85-36* (Stanford, CA: Center for the Study of Language and Information)
  - Reprinted in:
    - ∗ Dunlop, Charles; & Kling, Rob (eds.), *Computerization and Controversy* (San Diego: Academic Press, 1991): 632–646.
    - ∗ Colburn, Timothy R.; Fetzer, James H.; & Rankin, Terry L. (eds.) (1993), *Program Verification: Fundamental Issues in Computer Science* (Dordrecht, Holland: Kluwer Academic Publishers): 275–293.
  - Preprints online at:
    - ∗ http://eliza.newhaven.edu/ethics/Resources/14.Reliability-Responsibility/LimitsOfCorrectness.pdf (accessed 3 March 2015)
    - ∗ http://sdg.csail.mit.edu/6.894/dnjPapers/p18-smith.pdf (accessed 4 March 2015).

17.2 Introduction

In this chapter, we will examine the relationship between a program and the world in which the program is executed. We have touched on this topic before, primarily in §10.5.1 where we looked at the question whether a program designed to create Hollandaise sauce could succeed in a world in which the environmental conditions were not conducive to creating Hollandaise sauce (yet the program might be provably correct and even execute flawlessly), and in other places (§§3.10, 3.13, 7.10, 8.15, 16.6). Because a program is a piece of syntax (roughly, a collection of formal marks or symbols, grammatically organized), and because semantics is the study of the relation between syntax and the world, we are interested in the semantics of a program.

Our focus in this chapter will be an important paper, written in 1985, by Brian Cantwell Smith: "Limits of Correctness in Computers" [Smith, 1985]. Indeed, I consider this to be one of the three central papers in the philosophy of computer science, the others being [Turing, 1936] and [Turing, 1950].

17.3 Background

As we saw in the previous chapter, program verification is the subdiscipline of computer science that applies logic to a computer program in order to prove that it will do what it is supposed to do. One way of accomplishing this is to annotate each statement S of a program with pre- and post-conditions:

$$\{P\}S\{Q\}$$

where:

1. pre-condition P is a declarative proposition describing the input and the state of the computer before executing statement or instruction S,

2. post-condition Q is a declarative proposition describing the output and the state of the computer after executing S, and

3. and the braces are comment delimiters.

It is to be understood as follows:

> If P is true, and if S is executed, then Q is (or will be, or *should be*) true.

One objection to program verification is that a program can be "proven correct" yet not "do what you intend". One reason, as we saw in the last chapter, might be that the computer on which the program is run might fail physically. That is, the computer system might fail at the hardware level (this was one of the points of [Fetzer, 1988]).

A second reason, as we saw in §10.5.1, might be that the world is inhospitable. There are actually two ways in which this latter problem might arise: There might be a physical problem, not with the computer or with the environment, but with the connection between them: At a simple level, the cables connecting the computer to the world (say, to a printer) might be flawed. Or the world itself might not provide the

correct conditions for the intended outcome (it might be too hot, or too cold, and so on).

A third reason is related to the possible "hyper"-computability of interactive programs, as we discussed in §11.3: Such programs cannot be verified, because they might depend on the non-verifiable behavior of an "oracle" or human user.[1]

What does 'correct' mean in this context? Does it mean "doing what was intended"? If so, *whose* intent counts? Here is Smith on this question:

> What does *correct* mean, anyway? Suppose the people want peace, and the President thinks that means having a strong defense, and the Defense department thinks that means having nuclear weapons systems, and the weapons designers request control systems to monitor radar signals, and the computer companies are asked to respond to six particular kinds of radar pattern, and the engineers are told to build signal amplifiers with certain circuit characteristics, and the technician is told to write a program to respond to the difference between a two-volt and a four-volt signal on a particular incoming wire. If being correct means *doing what was intended*, whose intent matters? The technician's? Or what, with twenty years of historical detachment, we would say *should have been intended*?

According to Smith, the cause of these problems lies not in the relation of *programs* to the world, but in the relation of *models* to the world.

17.4   Models

What is a model? According to Smith, to design a computer system to solve a real-world problem, we must do two things:

1. Create a *model* of the real-world problem.

2. *Represent the* **model** in the computer.

The model that we create has no choice but to be "delimited", that is, it must be abstract—it must omit some details of the real-world situation. Abstraction, as we saw in Ch. 14, is the opposite of implementation. It is the removal of "irrelevant" implementation details.

Why must any real-world information be removed? There are many reasons, one of which is that it is methodologically easier to study a phenomenon by simplifying it, coming to understand the simplified version, and then adding some complexities back in, little by little. Another reason is related to the nature of maps. You can't use, or even have, a map of Florida that is the size of Florida. Such a map might be thought to be more useful than a smaller, more manageable one, in that it would be able to show all the detail of Florida itself. But its lack of manageability is precisely the problem.[2] What

---

[1]However, a possible way to avoid this latter problem is due to Amir Pneuli's use of temporal logic to analyze "reactive" programs [Hoffmann, 2010].

[2][Carroll, 1893, Ch. 11], online at
http://etc.usf.edu/lit2go/211/sylvie-and-bruno-concluded/4652/chapter-11-the-man-in-the-moon/

about eating your cake and having it, too? Suppose we omit an incomplete and separate model of the world altogether and "use the world as its own model", as [Brooks, 1991] suggests. Perhaps we *can* use the real world as a representation of itself: Robots don't need a map showing where there is a wall; if it bumps into one, it will know that it's there.

In any case, the first step in solving a problem is to create an abstract, that is, simplified, model of it. For example, Figure 17.1 is a picture showing (in a 2-dimensional way, of course!) a 3-dimensional, real-world house and a 2-dimensional model of it.
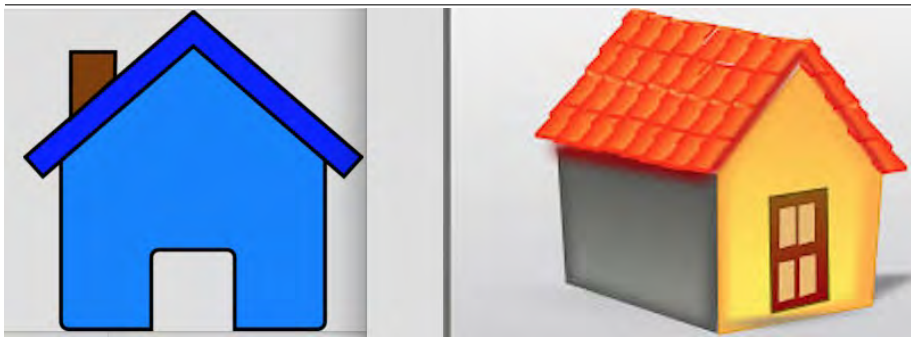
Figure 17.1: Right: 3D, real-world house; Left: 2D model of it

The second step is to use logical wffs or object-oriented data structures to represent, not the real-world situation, but the model. Figure 17.2 adds to the house and the house's model a computer representation of the model.

Figure 17.2: Right: 3D real-world house; Middle: 2D model of 3D real-world house; Left: computer model of 2D model

(Note to readers of the draft of this chapter: I am not an expert on illustrations. The intention of these figures is to show a 3D picture of a house; a 2D drawing of

(accessed 4 March 2015), [Royce, 1900, Vol. 1, esp. pp. 507ff], [Rosenblueth and Wiener, 1945, 320], [Rapaport, 1978, §5], [Borges, 1981, 234], [Eco, 1982].

it, preferably in the same colors!; and a computer representation of the 2D drawing. Please use your imagination to create a better set of figures. Smith's original is in Figure 17.3.)
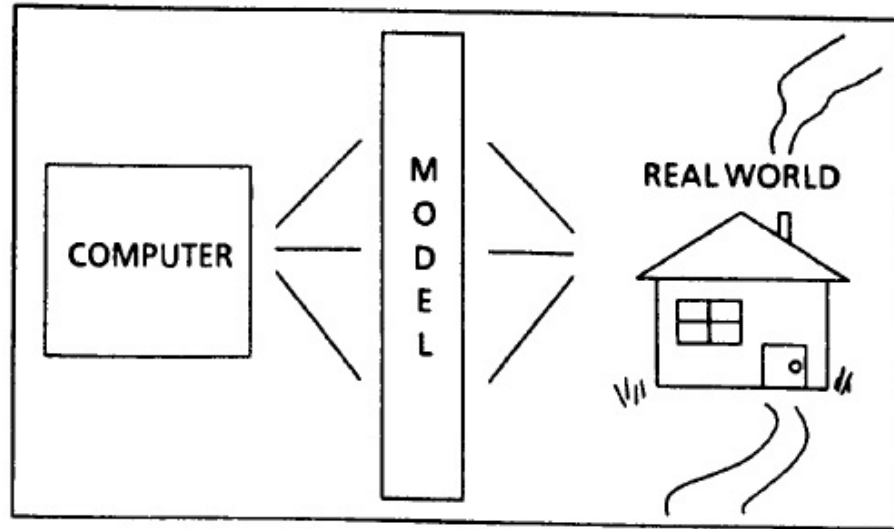
Fig. 1. Computers, models, and the embedding world.

Figure 17.3: From [Colburn et al., 1993, 283]

His point is that computers only deal with *their representations of* these *abstract models of* the real world. They are twice removed from reality.

Is that necessarily the case? Can't we skip the intermediate, abstract model, and directly represent the real-world situation in the computer? Perhaps, but this won't help us avoid the problem of *partiality* (or abstraction, or idealization, or simplification)..

All models are necessarily "partial". If they weren't, there would be too much complexity, and we would be unable to act. The only rational way to deal with (real-world) complexity is to analyze it, that is, to simplify it, that is, to deal with a partial (abstract) representation or model of it. (We have already discussed the role of computer science in managing complexity; see Chs. 3, 4, and 15. See also [Simon, 1962, Simon, 1996b].) We are condemned to do this whenever we humans must act or make decisions: If we were to hold off on acting or making a decision until we had complete and fully accurate information about whatever situation we were in, we would either be paralyzed into inaction or else the real world would change before we had a chance to complete our reasoning. (As Alan Saunders (and, later, John Lennon) said, "Life is what happens to us while we are making other plans.")[3] This is the problem that [Simon, 1996b] recognized when said that we must always reason

---

[3]http://quoteinvestigator.com/2012/05/06/other-plans/

with uncertain and incomplete (even noisy) information: We must "satisfice". GET EXACT CITATION. And this holds for computation as well as thinking.

But action is *not* abstract: You *and* the computer must act *in* the complex, real world. Therefore, such real-world action must be based on *partial models* of the real world, that is, on incomplete and noisy information. Moreover, there is no guarantee that the *models* are correct.

Action can help: It can provide feedback to the computer system, so that the system won't be isolated from the real world. Recall the blocks-world program that didn't "know" that it had dropped a block, but "blindly" continued executing its program to put the block on another (§10.5.1). If it had had some sensory device that would have let it know that it no longer was holding the block that it was supposed to move, and if the program had had some kind of error-handling procedure in it, then it might have worked much better (it might have worked "as intended").

The problem, as Smith sees it, is that mathematical model theory only discusses the relation between the model and a *description* of the model. It does not discuss the relation between the model and the world. A model is like eyeglasses for the computer, through which it sees the world. The model is the world as the computer sees it.

(The philosopher Immanual Kant said that the same thing is true about us: Our concepts are like eyeglasses that distort reality; our only knowledge of reality is filtered through our concepts, and we have no way of knowing how things "really" are "in themselves", unfiltered through our concepts (as illustrated in Figure 17.4).

Figure 17.3 is Smith's version of this picture. The problem is that we have to act in *W* on the basis of *M*.

Similarly, to prove a program correct, we need both (a) a *specification* (a model of the real-world problem) that says (declaratively) *what* the computer systems should do and (b) a *program* (a computer model of the specification model) that says (procedurally) *how* to accomplish this. A correctness proof, then, is a proof that any system that obeys the program will satisfy the specification. But this is a proof that two *descriptions* are compatible. The program is proved correct *relative to* the specification. So, a better term might be "proof of relative consistency". Suppose the proof fails to show "correctness"; what does this mean? It means *either* that the program is wrong, *or* that the specification is wrong (or both). And, indeed, often we need to adjust both specification and program.

The real problems lie in the model-world relation, which correctness does not address. This is one of the morals of Cleland's and Fetzer's claims. That is, programs can fail because the models can fail to correspond to the real world *in "appropriate" ways*. But that italicized clause is crucial, because all models abstract from the real world, but each of them do so in different ways.

17.5 Summary

To sum up: Real-world tasks are complex. Models abstract from this complexity, so they can never match the rich complexity of the world. Computers see the world through models of these models (but so do people!). Would an automated system designed to decide quickly how to respond to an emergency make you feel uneasy?
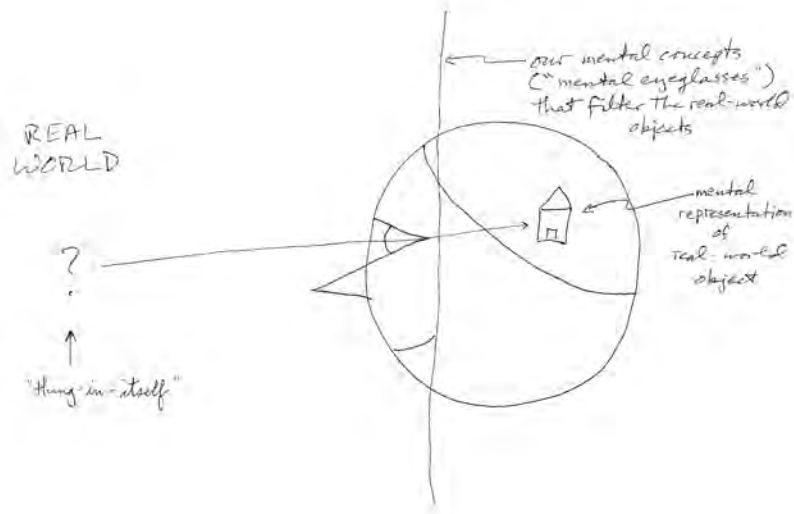
Figure 17.4: A cognitive agent looking at a real-world object that the agent categorizes as a house. Light reflecting off the house (the "thing-in-itself") enters the agent's eyes and the resulting neural signals are "filtered" through the agent's mental concepts, producing a mental image of the house. The mental image may, or may not, be a "perfect" representation of the house, but the agent has no way to directly compare the mental image with the real house, independent of the agent's concepts.

But so should a human who has to make that same decision. And they should both make you uneasy for the same reason: They have to reason and act on the basis of partial (incomplete) information. (We'll discuss this in the next chapter!) Such reasoning cannot be proved correct (and simulation only tests the computer-model relation, not the model-world relation). So, empirical *reliability* must supplement program verification. Therefore, we must embed the computer in the real world.

17.6 NOTES FOR NEXT DRAFT

1. Probably should incorporate my Covey Award paper, online at http://www.cse.buffalo.edu/~rapaport/Papers/covey.pdf

2. The debate between Rescorla-Cleland and Piccinini as to whether computation concerns the internal, syntactic manipulation of symbols or the external, semantic interpretation of them is reminiscent of Smith's gap. As [Rescorla, 2007, 265] observes, we need a computable theory of the semantic interpretation function, but, as Smith observes, we don't (can't?) have one, for reasons akin to the CTCT problem.

3. Re the opening motto:
   How does a program interact with the world? Via the *process* (which is a physical/causal entity (or event?) *in* the world. So, how does the program interact with the *process*? Via the compiler? No; that's just a first step, translating the program into a the machine language that the machine understands. The loader? Yes, because that's what copies/transduces? the machine-language program into the memory; it sets the switches.

4. In §17.4, cite McMullin on idealization, and say more about idealization. Also, mention that using the world as its own model is how robotic vacuums work; say more about Brooks.

5. Possibly include a short discussion of Wigner? At least cite in Further Readings as follows:

   • Wigner, Eugene (1960), "The Unreasonable Effectiveness of Mathematics in the Natural Sciences", *Communications in Pure and Applied Mathematics* 13(1) (February), https://www.dartmouth.edu/~matc/MathDrama/reading/Wigner.html, http://www.maths.ed.ac.uk/~aar/papers/wigner.pdf (both accessed 7 April 2015)

     – Also see reply by a computer scientist in [Hamming, 1980].

6. Any machine is a prisoner of its input and output domains. SS, our example [symbol] system, presents an abstract machine-centered view, so that the external world is pretty much what is seen by the machine. But this is deceptive. *Machines live in the real world and have only a limited contact with it.* Any machine, no matter how universal, that has no ears (so to speak) will not hear; that has no wings, will not fly. Thus universality will be relative to the input and output channels. Such a consideration is alleviated in theoretical discussions by the simple expedient of considering only abstract inputs and outputs. It can be alleviagted in the real world by providing transducers that encode from one input-output channel to another. Thus, being able to produce any function between two

given domains [that is, being "universal" in Newell's sense] permits[4] inducing any function between two other domains if the domains are hooked up appropriately. [Footnote in original omitted here.] But this interface limit must always be remembered. [Newell, 1980, 148; my italics]

- I won't describe SS here, but you can think of it as a slightly more elaborate kind of Turing machine. So Newell's first point is that Turing machines only see the external world from the point of view of the marks on its tape.

- The italicized sentence could have been taken straight out of [Smith, 1985]! Smith, however, might have replaced 'and' with 'but', to emphasize the twice-removed-from-reality nature of the "limited contact".

- The sentence that follows the italicized one should be compared to Piccinini's remarks in [Piccinini, 2011, 741, n. 11] (see Ch. 7, NOTES§12).

- By 'universality', Newell means the ability to "produce an arbitrary input-output function" (p. 147). His point in this sentence is that, in order for a Turing machine to deal with the real world, it has to overcome its reliance on the just the marks on its tape. As the next sentence suggests, we don't have to worry about that if we're not dealing with the real world. If, for example, we wanted to write a toy computer program that deals with houses, it's enough to let it deal with computer representations of houses, as in Fig. 17.2.

- The transducers do the hard work of encoding the real world into the syntax of the Turing-machine tape's marks and decoding those marks back into real-world entities.

- The penultimate sentence reminds us that the input-output codings must themselves be computable. And the last sentence takes us back to Smith, suggesting even another reason to be cautious about the "limits of correctness": Not only is the computer separated from the real world in the way that Smith notes, but the links that it does have are indirect and subject to their own translation limitations.

7. [Thagard, 1984, 82], citing [Zeigler, 1976], also notes that computers are twice removed from reality, because "a computer simulates a model which models a real system".

8. Smith's discussion at the beginning of his paper, about the computer that confused the rising moon with a Soviet nuclear attack because "this lunar reflection hadn't been predicted by the system's designers" (p. 18), is reminiscent of Cleland's hollandaise-sauce on the moon. If Cleland's recipe assumes that it is being executed on Earth, then it can't be faulted for failing on the moon; someone using it on the moon is the guilty culprit. But if Cleland's recipe advertises itself as being universally applicable, then it is at fault. One can debug the recipe

---

[4]I think 'requires' would have been a more appropriate word!

by adding a precondition that it must be executed on Earth, or by expanding it to clarify how to modify it if it is used on the moon (as in Denver cake mix recipes). Similarly, the early-warning system computer must be debugged by clarifying an unstated assumption (perhaps that any signal of a certain type must be a nuclear attack).

9.   In general, it is impossible, when writing specifications, to include *everything* that you want: legal contracts, and other humanly interpretable specifications, are always stated within a background of common sense, to cover the myriad unstated and unstatable assumptions assumed to hold in force. (Current computer programs, alas, have no common sense. . . .) [Smith, 1985, 23]

This is both a statement of the "gap", as well as the insight behind McCarthy's notion of circumscription.

10.   . . . as well as modelling the artifact itself, you have to model the relevant part of the world in which it will be embedded. [Smith, 1985, 24, col. 1]

This is essentially Shapiro's point that I discuss in [Rapaport, 2015]. This is one way to handle Cleland's hollandaise-sauce problem.

Indeed, I should probably incorporate some of that Covey-award essay in this chapter!

11.   The systems that land airplanes are hybrids—combinations of computers and people—exactly because the unforeseeable happens, and because what happens is in part the result of human action, requiring human interpretation. [Smith, 1985, 24, col. 2]

First, note that this anticipates the points about "extended cognition" made by Hutchins, and Clark & Chalmers (give citations). Second, whether this is a necessary truth is quite another matter. Smith wrote this long before the advent of self-driving cars. Now, it remains to be seen, even as of this writing (2015), whether self-driving cars will continue to need human intervention (remember: self-driving elevators don't need very much of it!). Nevertheless, such "hybrid" or "extended" systems will probably remain a reality.

12.   Reasoning is based on partial models. Which means it cannot be guaranteed to be correct. [Smith, 1985, 24, col. 2]

This is what lies behind Simon's notion of satisficing and bounded rationality.

13. [Wegner, 1997, 82] points out that interactive computer programs "are grounded in. . . external reality" rather than being "autistic".

14. Note that the issue of hypercomputation, especially interactive computation, is directly relevant to this chapter. After all, most interesting computer programs have to interact with the real world.

15.   Human fallibility means some of the more subtle, dangerous bugs turn out to be errors in design; the code faithfully implements the intended design, but the design fails to correctly handle a particular "rare" scenario. [Newcombe et al., 2015, 67]

This is a nice statement of Smith's Gap.

16.   A specification is an abstraction. It should describe the important aspects and omit the unimportant ones. Abstraction is an art that is learned only through practice. ... [A] specification of what a piece of code does should describe everything one needs to know to use the code. It should never be necessary to read the code to find out what it does. [Lamport, 2015, 39]

Smith's gap is due, in part, to the fact that specs are abstractions. How does one know if something that has been omitted from the spec is important or not? This is why it is an art and why there's no guarantee that the model is correct (matches reality).

17. Smith's main point is that program verification proves at most that a program satisfies its specification. But there's no way to formally prove that a specification is correct, that it matches reality [Smith, 1985, §6]. Both the specification and the program are models. The former is a model of the world; the latter is a model of the former (hence its double remove from reality).

18. [Cleland, 2004, 220] also talks about something very much like Smith's gap: "There is a causal gap between the actions specified by a quotidian procedure and the physical processes that make it reliable". But she finds a positive aspect to it: "the gap between action and process is actually a perk. Because of it, one can make a great Hollandaise sacue without knowing any chemistry. One need only know how to perform the action specified by the recipe. Nature takes care of the rest."

19. Nice question for further discussion (thanks to Albert Goldfain, personal communication, 3 April 2007):

An artist must paint not what he sees in nature, but what is there. To do so he must invent symbols, which, if properly used, make his work seem even more real than what is in front of him. (Charles E. Burchfield, https://www.burchfieldpenney.org/collection/charles-e-burchfield/biography/)

Change 'artist' to 'programmer' and 'paint' to 'program': "Programmers must program not what they see in nature, but what is there. To do so they must invent symbols, which, if properly used, make their work seem even more real than what is in front of them". Is the artist's task different from the scientist's or the programmer's? Can programs (or paintings, or scientific theories) end up seeming more "real" to us than the things that they are models of? Is it easier to

understand the behavior of the process of a program that models a hurricane (for example) that to understand the real hurricane itself?

20. [Mahoney and Haigh (ed.), 2011, 104–105] describes the "software life cycle", which begins with a "real world (physical) system", analyzed into (an abstract) "computational model of" that system, then implemented in a computer program, and ultimately resulting in a (once again physical) implementation as a machine. (Compare Marr's three levels of analysis!). He notes that, despite the fact that "We are on firmest theoretical ground at" the end of this cycle with the "machine as operational model of [the original physical] system", "the problems of air traffic control systems, of national weather systems, of airline booking systems, all lie at" the beginning of the cycle, "where a real-world system must be transformed into a computational model." That is where Smith's Gap is found. "That is where software engineering is not about software, indeed where it may not be about engineering at all."

21. [Mahoney and Haigh (ed.), 2011, 117] goes on to say that modeling the real world in the computer "involves an understanding not of computers but of the real-world situation in question. ...That is not what one learns in studying computer science; that is not what computer science is about." There are two observations to make about this: First, this makes Smith's Gap even wider. Note that, to model the real world for a computer, the modeler needs to understand both the real world and the computer. Second, it makes a strong recommendation about undergraduate education in computer science!

22. Smith's Gap may be related to at least one view of the nature of computer science (the issues we looked at in Ch. 3): [Mahoney and Haigh (ed.), 2011, 183–184] observes that even if (or to the extent that) computer science is not natural science but a science of the artificial (especially given the fact that computers work on partial models (hence artificial) models of the world, "those two worlds [natural and artificial] meet in the physical computer, which enacts a program in the world."

17.7   Further Sources of Information

1. The *New York Times* report on the moon-missile mistake:

   - "Canadian Is Praised over Missile Scare", *New York Times* (23 December 1960).

2. For more examples like Smith's moon-mistaken-as-missile, only with more dire consequences, see:

   - Neumann, Peter G. (1993), "Modeling and Simulation", *Communications of the ACM* 36(4) (June): 124.

3. Cerf, Vinton G. (2013), "What's a Robot?", *Communications of the ACM* 56(1) (January): 7,
   http://www.cs.grinnell.edu/~davisjan/csc/105/2013S/articles/CACM-reliability.pdf
   (accessed 18 April 2015).

   - Suggests "that we should treat as robots any programs that can have real-world... effect. ... [T]hose of us who live in and participate in the creating of software-based 'universes' might wisely give thought to the potential impact that our software might have on the real world", a thought that echoes [Smith, 1985].

4. Dewar, Robert (2009), "CS Education in the U.S.: Heading in the Wrong Direction?", *Communications of the ACM* 52(7) (July): 41–43.

5. Fetzer, James H. (1999), "The Role of Models in Computer Science", *The Monist* 82(1): 20–36.

   - A clearly written summary and critique of [Smith, 1985], arguing that there are more than merely the two models that Smith considers (a specification as a model of the world and a program as a model of the specification).

6. Hayes, Brian (2007), "Calculating the Weather", *American Scientist* 95(3) (May-June): 271–273.

   - Another interesting article on computational modeling (and what can go wrong).

7. Jackson,   Michael   (2003),   "Why   Software   Writing   Is Difficult and Will Remain So", *Information Processing Letters* 88: 13–25, preprint at http://users.mct.open.ac.uk/mj665/turski07.pdf (accessed 18 April 2015).

   - Discusses what I call Smith's gap: "the interplay between the formal world of the computer and its programming language with the informal world where the problem to be solved is located" (from the abstract, p. 13).

8. Morrisett, Greg (2009), "A Compiler's Story", *Communications of the ACM* 52(7) (July): 106.

9. van Fraassen, Bas C. (2006), "Representation: The Problem for Structuralism", *Philosophy of Science* 73 (December): 536–547,
   http://www.princeton.edu/~fraassen/abstract/docs-publd/PSA04_Structure.pdf
   (accessed 18 April 2015)

   - Discusses the relation between formal, mathematical models of reality and the reality of which they are models, arguing that what I call Smith's gap presents difficulties for scientific realism (which we discussed in Ch. 4).

10. On the nature of abstraction, see:

(a) Kramer, Jeff (2007), "Is Abstraction the Key to Computing?", *Communications of the ACM* 50(4) (April): 36–42.

(b) Rapaport, William J. (1999), "Implementation Is Semantic Interpretation", *The Monist* 82 (1): 109–130.

(c) Rapaport, William J. (2005), "Implementation Is Semantic Interpretation: Further Thoughts", *Journal of Experimental and Theoretical Artificial Intelligence* 17(4) (December): 385–417.

Chapter 18

Computer Ethics I:
Are There Decisions
Computers Should Never
Make?

The artificial pericardium will detect and treat a heart attack before any symptoms appear. "It's twenty years or more out there," [John] Rogers said. "But we can see a pathway—it's not science fiction." Bit by bit, our cells and tissues are becoming just another brand of hardware to be upgraded and refined. I asked him whether eventually electronic parts would let us live forever, and whether he thought this would come as a relief, offer evidence that we'd lost our souls and become robots, or both. "That's a good thing to think about, and people should think about it," he said "But I'm just an engineer, basically." [Tingley, 2013, 80]

18.1   Recommended Readings:

1. Required:

   - Moor, James H. (1979), "Are There Decisions Computers Should Never Make?", *Nature and System* 1: 217–229.

2. Very Strongly Recommended:

   (a) Friedman, Batya; & Kahn, Peter H., Jr. (1992), "People Are Responsible, Computers Are Not", excerpt from their "Human Agency and Responsible Computing: Implications for Computer System Design", *Journal of Systems and Software* (1992): 7–14; excerpt reprinted in M. David Ermann, Mary B. Williams, & Michele S. Shauf (eds.) (1997), *Computers, Ethics, and Society, Second Edition* (New York: Oxford University Press): 303–314.

       - Can be read as raising an objection to Moor 1979, esp. the section "Delegating Decision Making to Computational Systems" (pp. 306–307 in the Ermann et al. 1997 reprint).

   (b) Johnson, George (2002), "To Err Is Human", *New York Times* (14 July), http://www.nytimes.com/2002/07/14/weekinreview/deus-ex-machina-to-err-is-human.html (accessed 14 November 2012).

       - An op-ed piece that provides an interesting, real-life, case study of Moor's problem and a possible counterexample to Friedman & Kahn 1992.

18.2 Introduction

Ethics is a branch of philosophy, so computer ethics should be a branch of the philosophy of computer science. But computer ethics is a large and long-standing discipline in its own right. We will focus on two topics that I think are central to the philosophy of computer science but that are not the focus of most discussions of computer ethics. This chapter will consider whether there are decisions that computers should never make, and in Chapter 20 will consider an ethical issue in AI.

18.3 Do Computers Make Decisions?

It certainly seems that some computers can make rational decisions for us. Of course, it is not just a physical computer that makes a decision; it is a computer program being executed by a computer that makes the decision, but we will continue to speak as if it is the computer that decides. Any time that a computer solves a problem, it is making a decision.

And, presumably, if the decision is made by following an algorithm that does not involve any random or interactive procedure, then it is a purely rational decision. By 'rational', I don't necessarily mean that it is a purely *logical* decision. It may, for instance, involve empirical data. It may even be *il*logical, but, as long as there is an algorithm that can be studied to see how it works, or as long as the program can explain how it came to its decision, I will consider it to be rational.

Now, if you believe in the possibility and value of program verification, it would certainly be nice if such a decision-making program could be verified. But, given that we may not be able to verify such programs, should we trust them?

And what is a "decision"? There are at least two senses of 'decision':

1. A decision could be made as the result of an arbitrary choice, such as flipping a coin: heads, we'll go out to a movie; tails, we'll stay home and watch TV.

2. A decision could be made as the result of investigating the pros and cons of various alternatives, evaluating these pros and cons, and then making a decision based on this evaluation.

Computers can easily make the first kind of decision for us. Can they make the second kind? The answer is, surely, 'yes': Computers can play games of strategy, such as chess and checkers, and they can play them so well that they can beat the (human) world champions. Such games involve choices among alternative moves that must be evaluated, with, one hopes, the best (or least worst) choice being made. Computers can do this.

Can computers be completely rational? As I suggested above, insofar as what they do is algorithmic, their behavior is rational. But are there limits to a computer's rationality? Model-theoretic limits in the sense of [Smith, 1985] affect us humans, too. So, if there are limits to *our* rationality, it shouldn't surprise us that the same limits might apply to computers. But, at worst, this might mean that they are no *less* rational than we are. We'll return to this later in this section.

Another question about decisions is whether the process of decision making must be conscious. It's one thing to go into an unconscious trance and emerge with a decision. It's another to consciously think about an issue and then come up with a decision that is clearly and rationally based on the thinking process. If consciousness is a necessary condition of making a rational decision, and if computers are not conscious (as many have argued), then they cannot make such rational decisions.

But there are many cases in which we humans make unconscious decisions (without going into trances!). For instance, we make many decisions about what to eat or wear without much conscious thought (of course, we also make many such decisions consciously, with great thought). We also make decisions quickly when we are driving; often, we have no time to stop and think, but must react (that is, make decisions) "instinctively".

Cognitive scientists tell us that there are two decision-making "systems" in our brains: There is an unconscious, instinctive, intuitive, fast, tacit system (called 'System 1'), and ther is a conscious, rational, slow, explicit system (called 'System 2').[1] The former helps us avoid immediate dangers; the latter helps us make logical decisions. Unfortunately, sometimes they give us different answers to the same question.

Arguably, a computer that can *tell* us *how* it made a decison, even if it is unconscious, is better than one that cannot.

But can a computer really make a decision? I suggested above that it is really the computer program being executed on a computer that is making the decision. And programs are written by human programmers. So, perhaps it is only *humans*, *using* computers, that make such decisions. Although that is surely sometimes the case, there are cases in which humans do delegate decision-making powers to computers, such as in traffic control. Typically, this happens in cases where large amounts of data are involved in making the decision or in which decisions must be made quickly and automatically. And, of course, given that one of the goals of computer science is to determine what real-world tasks are computable, finding out which decisions are computable is an aspect of that.

In any case, humans *might* so delegate such power. So, the ultimate question is: What areas of our lives should be computer-controlled, and what areas should be left to human control?

Are there decisions that non-human computers could *not* make as well as humans? For instance, there might be situations in which there are sensory limitations that prevent computer decisions from being fully rational. Or there might be situations in which a computer could not be empathetic.

To answer this question, we need to distinguish between what *is* the case and what *could be* the case. We could try to argue that there are some things that are *in principle* impossible for computers to do. Except for computationally impossible tasks (such as deciding whether an arbitrary computer program halts), this might be hard to do. But we should worry about the possible future now, so that we can be prepared for it if it

---

[1]For   other
adjectives characterizing these two systems, see http://en.wikipedia.org/wiki/Dual_process_theory#Systems
(accessed 20 April 2015). For further discussion, see [Kahneman, 2011]; for further citations, see
http://www.cse.buffalo.edu/∼rapaport/575/rules-connections.html#uncs-cognition

happens.

Whether there are, *now*, decisions that a computer could not make as well as a human is an empirical question. It is capable of investigation, and, currently, the answer is unknown. Many, if not most, of the objections to the limitations of computers are best viewed as research problems: If someone says that computers can't do X, we should try to make ones that do X.

Another version of our question is this: Are there decisions that non-human computers could make *better than* humans? We'll come back to this in the next section. But this raises another question: How could computer decision-making competence be judged? The best answer is: in the same way that *human* decision-making competence is judged, namely, by means of its decision-making record and its justifications for its decisions.

Why do justifications matter? After all, if a computer constantly bests humans at some decision-making task, why does it matter how it does it? (Maybe we would be better off not knowing! For a science-fiction treatment of this kind of issue, though not in the context of computers, see Arthur C. Clarke's *Childhood's End* [Clarke, 1953].) Presumably, however, decision-making computers do need to be accountable for their decisions, and knowing their justifications helps this accounting. The justifications, of course, need not be the same as human justifications. For one thing, human justifications might be wrong or illogical.[2]

This suggests that the question "Are there decisions that a computer should not make?" should really have nothing to do with computers! The question should really be: Are there decisions that should not be made on a rational basis?

Again: Are there decisions that a computer *should* never make? The computer scientist Joseph Weizenbaum has argued that, even if a computer *could* make decisions as well as, or even better than, a human, they shouldn't, *especially* if their reasons differ from ours [Weizenbaum, 1976].

And James Moor points out that, possibly, computers shouldn't have the *power* to make (certain) decisions, even if they have the *competence* to do so (at least as well as, if not better than, humans) [Moor, 1979]. But, if they have the competence, why shouldn't they have the power? For instance, suppose a very superstitious group of individuals makes poor medical decisions based entirely on their superstitions; shouldn't a modern physician's "outsider" medicine take precedence? And does the fact that computers are immune to human diseases mean that they lack the empathy to recommend treatments to humans?

[Moor, 1979] suggests that, if computers can make certain decisions at least as well as humans, then we should let them do so, and it would then be up to us humans to accept or reject the computer's decision. After all, when we ask for the advice of an expert in medical or legal matters, we are free to accept or reject that advice. Why shouldn't the same be true for computer decision making?

This is crucial: Humans should be critical thinkers. There is a fallacy called the

---

[2]See, especially, [Tversky and Kahneman, 1974]. Also see [Wainer, 2007] on humans' difficulty in reasoning about probability and statistics: "Ignorance of how sample size affects statistical variation has created havoc for nearly a millennium." For other citations on humans' difficulty in reasoning about probability and statistics, see

http://www.cse.buffalo.edu/~rapaport/575/reasoning.html

Appeal to Authority: Just because an authority figure says that *P* is true, it does not logically follow that *P* is true. Although logicians sometimes warn us about this fallacy, it *is* acceptable to appeal to an authority (even a computer!) as long as the final decision is yours. *You* can—and must—decide whether to believe the authority or to trust the computer.

But what about emergencies, or other situations in which there is no time for the human who must act to include the computer's recommendation in his or her deliberations?

18.4   Emergency Decisions

On July 1, 2002, a Russian airliner crashed into a cargo jet over Germany, killing all on board, mostly students [Johnson, 2002]. The Russian airliner's flight recorder had an automatic collision-avoidance system that instructed the pilot to go higher (to fly over the cargo jet). The human air-traffic controller told the Russian pilot to go lower (to fly under the cargo jet).

According to [Johnson, 2002, my emphasis], "Pilots tend to listen to the air traffic controller because they trust a human being and know that a person *wants* to keep them safe." But the human air-traffic controller was tired and overworked. And the collision-avoidance computer system didn't "want" anything; it simply made rational judgments. The pilot followed the human's decision, not the computer's, and a tragedy occurred.

There is an interesting contrasting case. In January 2009, after an accident involving birds that got caught in its engines, a US Airways jet "landed" safely on the Hudson River in New York City, saving all on board and making a hero out of its pilot. Yet [Langewiesche, 2009] argues that the plane, with its computerized "fly by wire" system, was the real hero. In other words, the pilot's heroism was due to his willingness to accept the computer's decision. (For contrasting discussions of this complicated case, see [Haberman, 2009], [Salter, 2010].)

A paragraph deeply embedded in [Kolata, 2004] suggests that people find it difficult to accept the rational recommendations even of other people. The article reports on evidence that a certain popular and common surgical procedure had just been shown to be of no benefit: "Dr. Hillis said he tried to explain the evidence to patients, to little avail. 'You end up reaching a level of frustration,' he said. 'I think they have talked to someone along the line who convinced them that this procedure will save their life.' " Perhaps the fundamental issue is not whether computers should make rational decisions or recommendations, but whether or why humans should or don't *accept* rational advice!

18.5   Moral Decisions

[Moor, 1979] suggests that, although a computer should make rational decisions for us, a computer should *not* decide what our basic goals and values should be. Computers should help us *reach* those goals or *satisfy* those values, but they should not *change* them.

But why not? Computers can't be legally or morally responsible for their decisions, because they're not persons. At least, not yet. But what if AI succeeds? We'll return to this in Chapters 19 and 20. Note, by the way, that for many legal purposes, non-human corporations *are* considered to be persons.

[Friedman and Kahn, 1997] argue that humans *are*—but computers are *not*—*capable* of being moral agents and, therefore, computers should be designed so that:

1. humans are *not* in "merely mechanical" roles with a diminished sense of agency, and

2. computers *don't* masquerade as agents with beliefs, desires, or intentions.

Let's consider point (1): They argue that computers should be designed so that humans *do* realize that they (the humans) *are* moral agents. But what if the computer has a better decision-making track record than humans? [Friedman and Kahn, 1997] offer a case study of APACHE, a computer system that can make decisions about when to withhold life support from a patient. It is acceptable if it is used as a tool to aid *human* decision makers. But human users may experience a "diminished sense of moral agency" when using it, presumably because a computer is involved.

But why? Suppose APACHE is replaced by a textbook on when to withhold life support, or by a human expert. Would either of those diminish the human decision-maker's sense of moral agency? In fact, wouldn't human decision makers be remiss if they *failed* to consult experts or the writings of experts? So wouldn't they also be remiss if they failed to consult a computer expert?

Or would humans experience this diminished sense of moral agency for the following reason? If APACHE's decisions exhibit "good performance" and *are* more relied on, then humans may begin to yield to its decisions. But why would that be bad?

Turning to point 2, computers should be designed so that humans *do* realize that computers are *not* moral agents. Does this mean that computers should be designed so that humans *can't* take [Dennett, 1971]'s "intentional stance" towards them?

But what if the computer *did* have beliefs, desires, and intentions? (AI researchers are actively designing computers that either really have them, or else that are best understood as if they had them; CITE.) Would they not then be moral agents? If not, why not? According to [Dennett, 1971], some computers can't *help* "masquerading" as belief-desire-intention agents, because that's the best way for *us* to *understand* them.

[Friedman and Kahn, 1997] argue that we should be careful about anthropomorphic user-interfaces, because the *appearance* of beliefs, desires, and intentions does not imply that they really *have* them. This is a classic theme, not only in the history of AI CITE ELIZA, but also in literature CITE GALATEA2.2, blade runner, and new spate of films, esp AI, HER, ex machina, etc.). And this is at the heart of the Turing test in AI, to which we now turn.

18.6   NOTES FOR NEXT DRAFT

1. One thing that Moor et al. don't focus on is whether certain decisions might be OK for computers to make, while others aren't.

18.7 Further Sources of Information

1. Books and Essays:

   - Anderson, Michael; & Anderson, Susan Leigh (2007), "Machine Ethics: Creating an Ethical Intelligent Agent", *AI Magazine* 28(4) (Winter): 15–26.

   - Anderson, Michael; & Anderson, Susan Leigh (2010), "Robot Be Good", *Scientific American* 303(4) (October): 72–77.

   - Aref, Hassan (2004), "Recipe for an Affordable Supercomputer: Take 1,100 Apples...", *Chronicle of Higher Education* (5 March): B14.

     – Suggests (but does not discuss) that supercomputers might make decisions that we could not understand:

     > As we construct machines that rival the mental capability of humans, will our analytical skills atrophy? Will we come to rely too much on the ability to do brute-force simulations in a very short time, rather than subject problems to careful analysis? Will we run to the computer before thinking a problem through?... A major challenge for the future of humanity is whether we can also learn to master machines that outperform us mentally.

     – On the notion of "our analytical skills atrophy"ing, you might enjoy the following science-fiction story about a human who rediscovers how to do arithmetic after all arithemetical problems are handled by computers:

       ∗ Asimov, Isaac (1957), "The Feeling of Power", reprinted in Clifton Fadiman (ed.), *The Mathematical Magpie* (New York: Simon and Schuster, 1962): 3–14, accessed 14 November 2012 from
       http://www.themathlab.com/writings/short%20stories/feeling.htm

   - Asimov, Isaac (1950), "The Evitable Conflict", *Astounding Science Fiction*; reprinted in Isaac Asimov, *I, Robot* (Garden City, NY: Doubleday), Ch. 9, pp. 195–218, accessed 14 November 2012 from:
     http://preterhuman.net/texts/literature/books_by_author/A/Asimov,%20Isaac%20-%20I,%20Robot.pdf or http://tinyurl.com/AsimovIRobot

     – A fictional approach to Moor's question.

   - Brachman, Ronald J. (2002), "Systems that Know What They're Doing", *IEEE Intelligent Systems* (November/December): 67–71.

     – A leading AI researcher suggests how and why decision-making computers should be able to explain their decisions.

   - Greengard, Samuel (2009), "Making Automation Work", *Communications of the ACM* 52(12) (December): 18–19.

     – "Today's automated systems provide enormous safety and convenience. However, when glitches, problems, or breakdowns occur, the results can be catastrophic."

   - Heingartner, Douglas (2006), "Maybe We Should Leave That Up to the Computer", *New York Times* (18 July), accessed 14 November 2012 from
     http://www.nytimes.com/2006/07/18/technology/18model.html

     – On whether computers can make better decisions than humans.

- Markoff, John (2009), "Ay Robot! Scientists Worry Machines May Outsmart Man", *New York Times* (26 July): 1, 4; accessed 14 November 2012 from http://www.nytimes.com/2009/07/26/science/26robot.html

- Miller, Christopher A. (guest ed.) (2004), "Human-Computer Etiquette: Managing Expectations with Intentional Agents", *Communications of the ACM* 47(4) (April): 30–61.

    – Friedman & Kahn's (1992) argue that programmers should not design computer systems so that users think that the systems are "intelligent". The April 2004 issue of *CACM* has a whole section devoted to this.

- Neumann, Peter G. (1993), "Modeling and Simulation", *Communications of the ACM* 36(4) (June): 124.

    – Some cautionary examples.

- Sparrow, Robert (2007), "Killer Robots", *Journal of Applied Philosophy* 24(1): 62–77.

- Wagner, Alan R.; & Arkin, Ronald C. (2011), "Acting Deceptively: Providing Robots with the Capacity for Deception", *International Journal of Social Robotics* 3(1): 5–26; accessed 14 November 2012 from http://www.cc.gatech.edu/~alanwags/pubs/Acting-Deceptively-Final.pdf

- Wallach, Wendell; & Allen, Colin (2009), *Moral Machines: Teaching Robots Right from Wrong* (New York: Oxford University Press).

2. Websites:

    - On computer ethics in general:

        – Entire courses and books have been devoted to computer ethics. For more information, do a Google search on: "computer ethics"
        – AAAI's "AI Topics" website on "Ethical & Social Issues: Implications of AI for Society", is an excellent site, with many links: http://aitopics.net/Ethics (accessed 14 November 2012).
        – The Research Center on Computing & Society http://southernct.edu/organizations/rccs/ (accessed 14 November 2012).

Chapter 19

Philosophy of Artificial Intelligence

"The Turing Test(?): A problem is computable if a computer can convince you it is." —anonymous undergraduate student in the author's course, CSE 111, "Great Ideas in Computer Science" (2004)[1]

Figure 19.1: ©2005 by Mediagraphics

---

[1] http://www.cse.buffalo.edu/~rapaport/111F04.html

19.1   Recommended Readings:

1. Required:

   (a) Turing, Alan M. (1950), "Computing Machinery and Intelligence", *Mind* 59: 433–460.

       - Online at:
         http://mind.oxfordjournals.org/content/LIX/236/433.full.pdf+html
         (accessed 13 December 2013).

   (b) Searle, John R. (1980), "Minds, Brains, and Programs", *Behavioral and Brain Sciences* 3: 417–457; online at:

       - http://w3.uniroma1.it/episteme/file/Dispense20130329_MindsBrainsandPrograms.pdf
         (accessed 13 December 2013).
       - http://l3d.cs.colorado.edu/ ctg/classes/lib/cogsci/searle.pdf
         (accessed 13 December 2013).
       - http://cogprints.org/7150/1/10.1.1.83.5248.pdf
         (accessed 13 December 2013).

2. Recommended:

   - Rapaport, William 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; reprinted in James H. Moor (ed.), *The Turing Test: The Elusive Standard of Artificial Intelligence* (Dordrecht, The Netherlands: Kluwer Academic Publishers, 2003): 161–184;
     http://www.cse.buffalo.edu/∼rapaport/Papers/TURING.pdf
     (accessed 13 December 2013).

19.2 Introduction

As with computer ethics, the philosophy of artificial intelligence (AI) is another large and long-standing discipline in its own right, and so we will only focus on two main topics: What is AI? And is AI possible? For the second question, we will look at Alan Turing's classic 1950 paper on the Turing test of whether computers can think and John Searle's 1980 Chinese Room Argument challenging that test.

19.3 What Is AI?

Many definitions of AI have been proposed (for a list of some, see the appendix to this chapter). In this section, I want to focus on two nicely contrasting definitions. The first is by Marvin Minsky, one of the pioneers of AI research; the second is by Margaret Boden, one of the pioneers in cognitive science:

1. ...*artificial intelligence*, the science of making machines do things that would require intelligence if done by men.[2] [Minsky, 1968, v]

2. By "artificial intelligence" I...mean the use of computer programs and programming techniques to cast light on the principles of intelligence in general and human thought in particular.[3]. [Boden, 1977, 5]

Minsky's definition suggests that the methodology of AI is to study humans in order to learn how to program computers. Boden's definition suggests that the methodology goes in the opposite direction: to study computers in order to learn something about humans.

Both views are consistent with Stuart C. Shapiro's three views of AI [Shapiro, 1992] (cf. [Rapaport, 1998], [Rapaport, 2000], [Rapaport, 2003]):

1: AI as advanced computer science, or as engineering: • On this view, the goal of AI is to extend the frontiers of what we know how to program, in order to reach the ultimate goal of self-programming, natural-language-understanding computers, and to do this by whatever means will do the job, not necessarily in a "cognitive" fashion.

- The computer scientist John Case (personal communication) once described AI understood in this way as being at the "cutting edge" of computer science.[4]

AI as "computational cognition": This view has two different versions:

1. **AI as computational psychology:** Here, the goal of AI is to write programs as theories or models of human cognitive behavior. (Recall our discussion of computer programs as theories in Ch. 15.)

---

[2]I.e., by humans.

[3]This is just one sentence from a lengthy discussion titled "What Is Artificial Intelligence?", [Boden, 1977, Ch. 1]

[4]Another computer scientist, Anthony S. Ralston (personal communication) agreed with the topological metaphor, except that instead of describing AI as being at the *cutting* edge, he called it the "periphery" of computer science!

2. **AI as computation philosophy:** Here, the goal of AI is to investigate whether cognition is computable, that is, whether it is (expressible as) one or more recursive functions. By 'cognition', I mean such "mental" states and processes as belief, consciousness, emotion, language, learning, memory, perception, planning, problem solving, reasoning, representation (including categories, concepts, and mental imagery), sensation, thought, etc. If these states and processes are computable, what algorithms are necessary or sufficient for them? If not all of them are fully computable, how much of cognition *is* computable?

Minsky's view of AI as moving from humans to computers and Boden's view as moving from computers to humans are both valid: AI is, in fact, a two-way street. This is one reason why my preferred name for the field is 'computational cognition'. Other reasons have to do with the ambiguities of the word 'intelligence', which is often used in the sense of IQ (AI is not necessarily concerned only with finding programs with high IQ), and misleading implications of the word 'artificial' (which can suggest that "artificial" entities aren't the real thing; 'synthetic' might be a better adjective, because an artificial diamond might not be a diamond—it might be a cubic zirconium—whereas a synthetic diamond *is* a real diamond that just happened to be formed in a non-natural way).

So, computational cognition is the branch of computer science (working with other disciplines, such as cognitive anthropology, linguistics, cognitive neuroscience, philosophy, and psychology, among others) that tries to answer the question: How much of cognition is computable?

The working assumption of computational cognition is that *all* of cognition is computable: "The study [of AI] is to proceed on the basis of the conjecture that every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it" [McCarthy et al., 1955].

And its main open research question is: Is *what is not yet known to be computable* computable? If so, what are the implications of that fact? (We'll investigate this when we look at the Turing test in §19.4.) If not (which is the conclusion of the Chinese Room Argument to be discussed in §19.5), why not, and what does that tell us about cognition?

19.4   The Turing Test

19.4.1   How Computers Can Think

We have seen that computational cognition (AI broadly construed, including what is sometimes called "computational cognitive science") holds that cognition is comput*able*. For our present purposes, it doesn't matter whether the computations are of the classical, symbolic variety or the connectionist/artificial neural network variety. Nor does it matter whether the neuron firings that produce cognition in the human brain can be viewed as computations (for further discussion of this, see [Rapaport, 2012b]).

All that matters is this philosophical implication:

> If (and to the extent that) cognitive states and processes can be expressed as algorithms, then they can be implemented in non-human computers.

And this raises the following questions:

- Are computers executing such cognitive algorithms merely *simulating* cognitive states and processes?

- Or are they *actually exhibiting* them?

In popular parlance, do such computers think?

In this section, we will look at one answer that has been given to this question, an answer that arises from what is called the Turing test. In the next section, we will look at an objection to it has been presented in the form of the Chinese Room Argument. And after that, I will offer my own interpretation of the situation, introducing a theory I call "syntactic semantics".

19.4.2 The Imitation Game

Alan Turing's second most important paper, [Turing, 1950], never mentions a "Turing test" (just as his most important paper, [Turing, 1936], never mentions a "Turing machine"). Instead, he introduces a parlor game (which you can actually play—this is not a mere thought experiment) that he calls the "Imitation Game".

The Imitation Game consists of three players: A man, a woman, and an interrogator (who might be either a man or a woman; it might matter whether the interrogator is a man rather than a woman, or the other way round, but we'll ignore this for now). The three players are placed in separate rooms, so that they cannot see each other, and they only communicate by means of something like a "chat" interface, so that they cannot hear each other. The reason that they are not allowed to see or hear each other is that the point of the game is for the interrogator to determine which room has the man, and which room has the woman. To make things interesting, the woman is supposed to tell the truth in order to convince the interrogator that she is the woman, but the man is supposed to convince the interrogator that he (the man) is the woman, so he will occasionally have to lie. The man wins if he convinces (fools) the interrogator that he is the woman; the woman wins if she convinces the interrogator that she is the woman. Another way of thinking of this is that the interrogator wins if he correctly figures out who is in which room. If the man wins, then he is said to have passed the test. Turing suggested that "an average interrogator will not have more than 70 per cent. chance of making the right identification after five minutes of questioning" [Turing, 1950, 442]. But the actual amount of time may be irrelevant. One could conduct a series of imitation games and calculate appropriate statistics on how likely an interrogator is to make a correct determination after a given period of time.

What does this have to do with whether computers can think? What has come to be known as the Turing test makes one small change in the Imitation Game:

> We now ask the question, "What will happen when a machine takes the part of [the man] in this game?" Will the interrogator decide wrongly as often when the game is played like this as he [or she] does when the

game is played between a man and a woman? These questions replace our
original, "Can machines think?" [Turing, 1950, 434]

It turns out that there is some ambiguity in Turing's question, and, in any case,
other modifications are possible. First, what is the "machine" (that is, the computer)
supposed to do? Is it supposed to convince the interrogator that it is the woman? (That
is, is it supposed to imitate a woman?) Or is it supposed to convince the interrogator
that it is a *man who is trying to convince the interrogator that he is a woman*? (That
is, is it supposed to imitate a man?) Usually, the Turing test is taken, more simply
and less ambiguously, to consist of a set up in which a computer, a human, and a
human interrogator are located in three different rooms, communicating over a chat
interface, and in which both the human and the computer are supposed to convince
the interrogator that each is a human. To the extent that the computer convinces the
interrogator and the human doesn't, under the same criteria for successful convincing
that obtains in the original imitation game, the computer is said to have passed the
Turing test.

And an even simpler version is consists merely of two players: a human interrogator
and someone or something (a human or a computer) in two separate, chat-interfaced
rooms. If a computer convinces the interrogator that it is a human, then it passes the
Turing test. Which version of the test is better has been discussed in the literature
CITE, but we will now turn to Turing's view of the matter.

Here is his answer to the question that has now replaced "Can machines think?":

> I believe that at the end of the century [that is, by the year 2000] *the use
> of words* and *general educated opinion* will have altered so much that
> one will be able to speak of machines thinking without expecting to be
> contradicted" [Turing, 1950, 442].

Before trying to understand exactly what this might mean, let's consider the Turing test
a bit further.

19.4.3   Thinking vs. "Thinking"

In 1993, *The New Yorker* magazine published a cartoon by NAME, showing a dog
sitting in front of a computer talking to another dog, the first one saying, "On the
Internet, nobody knows you're a dog." This cartoon's humor arises from the fact that
you do *not* know with whom you are communicating via computer! Yes, it's unlikely
that there's a dog typing away at the other end of a chat session or an email, but could
it be a computer pretending to be a human? Or could it be a 30-year-old pedophile
pretending to be a 13-year-old classmate? Normally, we *assume* that we are talking
to people who really are whom they say they are. In particular, we assume that we
are talking to a human. But really all we know is that we are talking to an entity with
human cognitive capacities.

And that, I think, is precisely Turing's point: An entity with human cognitive
capacities is all that we can ever be sure of, whether that entity is really a human or
"merely" a computer. This is a version of what philosophers have called the argument
from analogy for the existence of other minds. An argument from analogy is an
argument of the form:

> Entity A is like (that is, is analogous to) entity B with respect to important properties $P_1, \ldots, P_n$.
> $\therefore A$ is (probably) like B with respect to some other property Q.

Another way to formulate it is this:

1. A is like B with respect to $P_1, \ldots, P_n$.

2. In addition, A has property Q.

3. $\therefore$, B has (or probably has, or may have) property Q.

Such an argument is non-deductive: It's quite possible for the premise to be true but for the conclusion to be false. But it has some inductive strength: The more alike two objects are in many respects, the more likely it is that they will be alike in many other respects (and maybe even all respects).

The problem of the existence of other minds is this: I know that I have a mind (because I know what it means for me to think, to perceive, to solve problems, etc.). How do I know whether you have a mind? Maybe you don't; maybe you're just some kind of computer, or android,[5] or philosophical zombie.[6]

The argument from analogy for the existence of other minds is therefore this argument:

1. You are like me with respect to all physical and behavioral properties.

2. I have a mind. (Or: My behavioral properties can best be explained by the fact that I have a mind.)

3. $\therefore$, (probably) you have a mind. (Or: Your behavioral properties can best be explained if it is assumed that you also have a mind.)

Of course, this argument is deductively invalid. I could be wrong about whether you are biologically human. In that case, the best explanation of your behavior might be, not that you have a mind, but that you are a computer who has been cleverly and suitably programmed. Now, there are two ways to understand this: One way to understand it is to say that you don't have a mind; you're just a cleverly programmed robot. But another way to understand it is to say that being cleverly programmed in that way is exactly what it means to have a mind; perhaps we are both cleverly programmed in that way, or perhaps (1) you are programmed in that way, whereas (2) I have a brain that behaves in that way, but (3) these are simply two different implementations of "having a mind".

In either case, am I wrong about your being able to think? That is, am I wrong about your (human) cognitive abilities? Turing's answer is: No! More cautiously, perhaps, his answer is that whether I'm wrong depends on the definition of (human) cognitive abilities (or thinking): If human-like cognition requires a (human) brain, and you lack one, then, technically speaking, you don't have human-like cognition (even if

---

[5] like Commander Data in *Star Trek*.

[6] A philosophical zombie is not a horror-movie zombie. Rather, it is an entity who is exactly like us in all respects but who lacks a mind or consciousness. CITE

you pass the Turing test). On this view, I think, but you can only "think". But if human-like cognition can be implemented in different ways—that is, if it does not require a (human) brain—then we both have human-like cognition (and that's why you pass the test). On this view, we both can think.

Here's an analogy: Everyone can agree that birds fly.[7] Do people fly? Well, we certainly speak as if they do; we say things like, "I flew from Buffalo to JFK last week." But we also know that I don't literally mean that I flapped my arms when flying from Buffalo to JFK; rather, I flew in an airplane: It wasn't I who was flying; it was the airplane. OK; do planes fly? Well, they don't flap their wings, either. In what sense are they flying?

There are two ways to understand what it means to say that planes fly: One way is by "metaphorical extension" [Lakoff and Mark, 980a], [Lakoff and Mark, 1980]. The reason we say that planes fly is that what they are doing is very much like what birds do when they fly (they move through the air, even if their methods of doing so are different), so, instead of using a simile, saying that planes move through the air like birds fly, we use a metaphor, saying directly that planes fly. And then that metaphor becomes "frozen"; it becomes a legitimate part of our language, so much so that we no longer realize that it is metaphorical. This is just like what happens when we say that time is money: We say things like, "You're *wasting* time", "This will *save* you time", "How did you *spend* your vacation?", and so on. But we're usually not aware that we are speaking metaphorically (until someone points it out), and there's often no other (convenient) way to express the same ideas. *As Turing said, our "use of words" has changed.*

The other way to understand what it means to say that planes fly is that we have realized that flapping wings is not essential to flying. There are deeper similarities between what birds and planes do when they move through the air that have nothing to do with wing-flapping but that have everything to do with the *shape* of wings and, more importantly, with the physics of flight. We have developed a more abstract, and therefore more general, theory of flight, one that applies to both birds and planes. And so we can "promote" the word 'fly' from its use for birds to a more general use that also applies to planes. Flying can be implemented by both biological and non-biological entities. *As Turing said, "general educated opinion" has changed.*

In fact, both the use of words *and* general educated opinion has changed. Perhaps the change in one facilitated the change in the other; perhaps the abstract, general theory *can account for* the metaphorical extension.

What does this have to do with the philosophy of AI? Well, what is a computer? As we saw in §6.2, originally, a computer was a human who computed. That was the case till about the 1950s, but, a half-century later, a computer is a machine. (To emphasize this, pre-1950, what we now call 'computers' were called 'digital' or 'electronic computers' (to distinguish them from the human kind). But, now, it is very confusing to read pre-1950 papers without thinking of the word 'computer' as meaning, by default, a non-human machine.) Now, in the beginning of the 21st century, general educated opinion holds that computers are viewed, not as implementing devices, but

---

[7]This is true in general, even if most birds actually don't fly: Not only do penguins, ostriches, etc., not fly, but baby birds, birds with injured wings, dead birds, etc., also don't fly. Handling the logic of statements like this is a branch of AI called "non-monotonic reasoning" CITE.

in functional, input-output terms. The same, as we will see, is slowly becoming true for the word 'think'. But some philosophers argue that what computers do is not *really* thinking. To understand why, let's think about one of those philosophers.

19.5 The Chinese Room Argument

Thirty years after Turing's publication of the Turing test, John Searle published a thought experiment called the Chinese Room Argument [Searle, 1980] (cf. [Searle, 1982]). In this experiment, a human (John Searle himself, as it happens) who knows no Chinese is placed in a room (the "Chinese room") along with paper, pencils, and a book containing an English-language algorithm for manipulating certain "squiggles" (symbols that are meaningless to Searle-in-the-room[8]). Outside the room is a native speaker of Chinese. There is something like a mail slot in one wall of the Chinese room. Through that slot, the native speaker passes pieces of paper that contain a text written in Chinese and reading-comprehension questions about that text. When Searle-in-the-room gets these pieces of paper, which, from his point of view, contain nothing but meaningless squiggles, he consults his book and follows its instructions. Those instructions tell him to manipulate the symbols in certain ways, to write certain symbols down on a clean piece of paper, and to slip those "responses" through the mail slot. The native speaker who reads them sees that whoever (or whatever) is in the room has answered all the questions correctly, demonstrating a fluent understanding of Chinese. This is because the instruction book is a complete, Chinese natural-language-understanding algorithm. But, by hypothesis, Searle-in-the-room does not understand Chinese. We seem to have a contradiction.

The Chinese Room Argument (CRA) is offered as a counterexample to the Turing test, concluding from this thought experiment that it is possible to pass a Turing test, yet not really think. The setup of the CRA is identical to the simplified version of the Turing test: The interrogator is the native Chinese speaker, who has to decide whether the entity in the room understands Chinese. Because the interrogator determines that the entity in the room does understand Chinese (which is analogous to deciding that the entity in the simplified Turing test is a human, rather than a computer), but the entity in fact does not understand Chinese (which is analogous to the entity in the simplified Turing test being a computer), the test fails.

Or does it?

19.5.1 Two Chinese Room Arguments

Searle actually bases two arguments on the Chinese Room thought experiment:

The Argument from Biology: B1 Computer programs are non-biological.

B2 Cognition is biological.

B3 ∴ No non-biological computer program can exhibit biological cognition.

---

[8]I will distinguish between the real John Searle, philosopher and author of [Searle, 1980], and the Chineseless John Searle who occupies the Chinese room by referring to the latter as 'Searle-in-the-room'.

The Argument from Semantics: S1 Computer programs are purely syntactic.

  S2 Cognition is semantic.

  S3 Syntax alone is not sufficient for semantics.

  S4 ∴ No purely syntactic computer program can exhibit semantic cognition.

Objections have been raised against both arguments. We will look at a few of them.

The principal objection to the Argument from Biology is that premise B2 is at least misleading and probably false: Cognition can be characterized abstractly, and implemented in different media.

The principal objection to the Argument from Semantics is that premise S3 is false: Syntax—that is, symbol manipulation—does suffice for semantics.

After investigating these objections (and others), we will consider whether there are other approaches that can be taken to circumvent the CRA. One of them is to try to build a real analogue of a Chinese room; to do that, we will need to answer the question of what is needed for natural-language understanding.

19.5.2   The Argument from Biology

19.5.2.1   Causal Powers

Let's begin by considering some of the things that Searle says about the CRA:

> I still don't understand a word of Chinese and neither does any other digital computer because all the computer has is what I have: a formal program that attaches no meaning, interpretation, or content to any of the symbols. What this simple argument shows is that no formal program by itself is sufficient for understanding.... [Searle, 1982, 5]

Note that a program that *did* "attach" meaning, etc., *might* understand. But Searle denies that, too:

> I see no reason in principle why we couldn't give a machine the capacity to understand English or Chinese, since in an important sense our bodies with our brains are precisely such machines. But... we could not give such a thing to a machine where the operation of the machine is defined solely in terms of computational processes over formally defined elements.... [Searle, 1980, 422]

Why not? Because "only something that has the same causal powers as brains can have intentionality" [Searle, 1980, 423]. By 'intentionality' here, Searle means "cognition" more generally. So he is saying that, if something exhibits cognition, then it must have "the same causal powers as brains".

All right; what are these causal powers? After all, if they turn out to be something that can be computationally implemented, then computers can have them (which Searle thinks they cannot). So, what does he say they are? He says that these causal powers are due to the fact that "I am a certain sort of organism with a certain biological (i.e. chemical and physical) structure" [Searle, 1980, 422]. We've narrowed down

the nature of these causal powers a little bit. If we could figure out what this biological structure is, and if we could figure out how to implement that structure computationally, then we should be able to get computers to understand. Admittedly, those are big "if"s, but they are worth trying to satisfy.

So, what is this biological structure? Before we see what Searle says about it, let's thing for a moment about what a "structure" is. What is the "structure" of the brain? One plausible answer is that the brain is a network of neurons, and the way those neurons are organized is its "structure". Presumably, if you made a model of the brain using (let's say) string to model the neurons, then, if the strings were arranged in the same way that the neurons were, we could say that the model had the same "structure" as the brain. Of course, string is static (it doesn't do anything), and neurons are dynamic, so structure alone won't suffice, but it's a start.

But Searle doesn't think so. He says that a simulated human brain "made entirely of...millions (or billions) of old beer cans that are rigged up to levers and powered by windmills" would not really exhibit cognition even though it appeared to [Searle, 1982].[9] In other words, structure plus the ability to do something is not enough: Cognition must (also) be biological, according to Searle; it must be made of the right stuff.

But now consider what Searle is saying: Only biological systems have the requisite causal properties to produce cognition. So we're back at our first question: What are those causal properties? According to Searle, they are the ones that are "causally capable of producing perception, action, understanding, learning, and other intentional [i.e., cognitive] phenomena" [Searle, 1980, 422]. Again: What are the causal properties that produce cognition? They are the ones that produce cognition! That's not a very helpful answer.

Elsewhere, Searle does say some things that give a possible clue as to what the causal powers are: "mental states are both *caused by* the operations of the brain and *realized in* the structure of the brain" [Searle, 1983, 265]. In other words, they are *implemented* in the brain. And this suggests a way to avoid Searle's argument from biology.

19.5.2.2 The Implementation Counterargument

Remember, Searle says:

> [M]ental states are as real as any *other* biological phenomena, as real as lactation, photosynthesis, mitosis, or digestion. Like these other phenomena, mental states are caused by biological phenomena and in turn cause other biological phenomena. [Searle, 1983, 264, my emphasis]

Searle's "mental states" are biological *implementations*. But, if they are implementations, then they must be implementations of something else, in this case, of something more abstract: abstract mental states. For every implementation *I*, there is an Abstraction *A* such that *I* implements *A* (call this "implementation thesis 1").

---

[9][Weizenbaum, 1976, Ch. 5] considers computers "made of bailing wire, chewing gum, and adhesive tape". [Weizenbaum, 1976, Ch. 2] considers a Turing machine made of toilet paper and pebbles. And recall our discussion in §8.8 of Hilbert's tables, chairs, and beer mugs.

Let's review what Searle says a bit more closely:

1. "...intentional states...are both caused by and realized in the *structure* of the brain." [Searle, 1980, 451, my emphasis]

2. But brains and beer-cans/levers/windmills can share structure. This is a simple fact about the nature of *structure*.

3. ∴ point 1 must be false: It can't be a *single* thing—intentional (that is, mental) states—that is *both* caused by *and* realized in the brain. Instead, the brain causes (a) *implemented* mental states but (b) realizes *abstract* mental states, and those are two distinct things.

Recall our discussion of implementation from Ch. 14. There, we saw that Abstractions (a generalization of the computer science notion of abstract data type) can be implemented in more than one way—they can be "multiply realized". We saw that stacks can be implemented as arrays or as lists, that any sequence of items that satisfy Peano's axioms is an implementation of the natural numbers, that any two performances of the same play or music are different implementations of the script or score, and so on. For every Abstraction *A*, there are more than one implementations I, I', etc. (call this "implementation thesis 2").

So, Searle says that the human brain can understand Chinese because understanding is biological, whereas a computer executing a Chinese natural-language-understanding program cannot understand Chinese, because it is not biological. But the implementation counterargument says that, on an abstract, functional, computational notion of understanding as an Abstraction, understanding can be implemented in both human brains and in computers, and, therefore, both can understand.

More generally, if we put implementation theses 1 and 2 together, we can see that if we begin with an implementation (say, real, biological mental states), we can develop an abstract theory about them (this is what cognitive science, including computational cognitive science, tries to do). But once we have an abstract theory, we can re-implement it in a different medium. If our abstract theory is computable, then we can re-implement it in a computer. When this happens, our use of words changes, because general educated opinion changes, as Turing predicted.

19.5.3   The Argument from Semantics

19.5.3.1   Which Premise Is at Fault?

Recall the Argument from Semantics:

S1 Computer programs are purely syntactic.

S2 Cognition is semantic.

S3 Syntax alone is not sufficient for semantics.

S4 ∴ No purely syntactic computer program can exhibit semantic cognition.

In this section, we will look at reasons for thinking that (S3) is false, that syntax does suffice for semantics.

First, let's consider this argument in a little bit more detail. Premise (S1) says that computer programs merely tell a computer to (or describe how a computer can) manipulate symbols on the basis of their properties as marks and their relations among themselves, completely independently of their semantic relations, that is, of the relations that the symbols have to other items, items that are external to the computer and that are the meanings or interpretations of the symbols, the things in the real world that the symbols represent.

Premise (S2) says that cognition is centrally concerned with such relations. Cognition, roughly speaking, is whatever the brain does with the sensory inputs from the external world. To fully understand cognition, according to this premise, it is necessary to understand the internal workings of the brain, the external world, and the relations between them. That is a semantic enterprise.

It seems clear that the study of relations among the symbols alone could not possibly suffice to account for the relations between those symbols and anything else. Hence premise (S3): Syntax and semantics are two different, though related, subjects.

Conclusion (S4) seems to follow validly. So, any questions about the goodness of the argument must concern its soundness: Are the premises true? Doubts have been raised about each of them.

Let's look at (S1) first: Although it is not a computer program, the World Wide Web is generally considered to be a syntactic object: a collection of nodes (for example, websites) related by links, that is, a mathematical graph. Some researchers have felt that there are limitations to this "syntactic" web and have proposed the Semantic Web. By "attaching meanings" to websites (as Searle might say), they hope to make the Web more...well...meaningful, more useful. Later WHERE, we will see that they way they do this is by adding more syntax! So, for now, we'll accept premise (S1).

Next, let's look at (S2): At least one major philosopher, Jerry Fodor, has argued that the study of cognition need not involve the study of the external world that is being cognized [Fodor, 1980]. For one thing, we would need to solve all of the problems of physics, cosmology, chemistry, biology, etc.—in other words, all of the problems of understanding the external world—before we could fully understand cognition. That by itself is not conclusive; after all, it has been argued that any single branch of AI is "AI complete", for instance, that the problem of natural-language understanding cannot be fully solved before fully solving all of the other problems in in AI. More importantly, Fodor argues that cognition is what takes place internally to the brain. Whether the brain correctly or incorrectly interprets its input from the external world, it's the interpretation that matters, not the actual state of the external world. This view is called "methodological solipsism". Solipsism is, roughly, the view that I am the only thing that exists; you are all figments of my imagination; note that you cannot make the same claim, because, after all, if solipsism is true, then you don't exist. (There's a story that, at a lecture that Bertrand Russell once gave on solipsism, someone in the audience commented that it was such a good theory, why didn't more people believe it?) *Methodological* solipsism is the view that, as a methodology for studying cognition, we can pretend that the external world doesn't exist; we only have to investigate what the brain does with the inputs that it gets, not where those inputs come from or what they

are really like.

Of course, if understanding cognition only depends on the workings of the brain and not on its relations with the external world, then the study of cognition might be purely syntactic. And so we're ready to consider premise (S3). Can we somehow get semantics from syntax? There are three, interrelated reasons for thinking that this is the case.

First, we can try to show that semantics, which is the study of relations between symbols and meanings, can be turned into a syntactic study, a study of relations among symbols and "symbolized" meanings. Second, it can be argued that semantics is recursive in the sense that we understand a syntactic domain in terms of an antecedently understood semantic domain, but that there must be a base case, and that this base case is a case of syntactic understanding. Third, we can take the methodologically solipsistic approach and argue that an internal, "narrow", first-person point of view is (all that is) needed for understanding or modeling cognition.

Before looking at each of these, remember that Searle claims that syntax cannot suffice for semantics because the former is missing the links to the external world. This kind of claim relies on two assumptions, both of which are faulty. First, Searle is assuming that computers have no links to the external world, that they are really (and not just methodologically) solipsistic.[10] But this is obviously not true, and is certainly inconsistent with [Smith, 1985]. Second, Searle assumes that external links are really needed in order to attach meanings to symbols. But, if so, then why couldn't computers have them just as well as humans do? Both humans and computers exist and act in the world. If we humans have the appropriate links, what reason is there to think that computers could not? (Well, there is the argument from biology, but this is still an issue that needs to be addressed.)

19.5.3.2 Semiotics

The first reason for thinking that syntax might suffice for semantics comes from semiotics, the study of signs and symmbols. According to one major semiotician, Charles Morris [Morris, 1938], semiotics has three branches: syntax, semantics, and pragmatics.

Given a formal system of "marks" (symbols without meanings)—sometimes called a "symbol system"—*syntax* is the study of relations *among* the marks: how to recognize, identify, and construct them (in other words, what they look like, for instance, their grammar), and how to manipulate them (for instance, their proof theory). Importantly, syntax does not study any relations between marks and non-marks.

Semantics is the study of relations *between* the marks and their "meanings". Meanings are part of a different domain of semantic interpretations (an "ontology"). Therefore, syntax cannot and does not suffice for semantics! (Of so it would seem.)

Pragmatics has been variously characterized as the study of relations among marks, meanings, and the cognitive agents that interpret them; or as the study of relations among marks, meanings, interpreters, and contexts. Some philosophers have suggested

---

[10]Actually, solipsism is not really the claim that only I exist, but that I live in a world of my own completely cut off from the external world, and so do you. This is reminiscent of Leibnizian mondas. But that's beyond our present scope.

that pragmatics is the study of everything that is interesting about symbols systems that isn't covered under syntax or semantics!

For our purposes, we only need to consider syntax and semantics. Again, let's be clear. Syntax studies the properties of, and relations among, the elements of a single set of objects (which we are calling "marks"); for convenience, call this set SYN. Semantics studies the relations between the members of two sets, SYN and a set that we'll call SEM.

Now, take the set-theoretical union of these two sets—the set of marks and the set of meanings: $M = \text{SYN} \cup \text{SEM}$. Consider M as a new set of marks. In other words, we have now "internalized" the previously external meanings into a new symbol system. And the study of the properties of, and the relations among, the members of M is M's syntax!

In other words, what was formerly semantics (that is, relations *between* the marks in SYN and their meanings in SEM) is now syntax (that is, relations *among* the new marks in M. This is how syntax can suffice for semantics.[11]

This can be made clear with a diagram NEEDS EXPLICATION.

This semiotic technique for turning semantics into syntax raises a number of questions: *Can* the semantic domain be internalized? Yes, under the conditions obtaining for human language understanding: How *do* we learn the meaning of a word? How, for instance, do I learn that the word 'tree' means "tree"? The common view is that this relation is learned by associating trees with the word 'tree'. But really what happens is that my internal representation of a tree is associated with my internal representation of the word 'tree'. That internal representation could be activated neurons, and in whatever way that neurons are bound together when, for instance, we perceive a pink cube (perhaps with shape neurons firing simultaneously with, and thereby binding with, color neurons that are firing) the neurons that fire when we see a tree might bind with the neurons that fire when we are thinking of, or hearing, or reading the word 'tree'.

And the same thing can happen in a computational cognitive agent. Suppose we have such an agent (a robot, perhaps; call her 'Cassie') whose computational "mind" is implemented as a semantic network whose nodes represent concepts and whose arcs represent structural relations between concepts, as in figure 19.3: There is a real tree external to Cassie's mind. Light reflecting off the tree enter Cassie's eyes; this is the causal link between the tree and Cassie's brain. The end result of the visual process is an internal representation of the tree in Cassie's brain. But she also has an internal representation of the word 'tree', and those two representations can be associated. What Cassie now has is an enlarged set of marks, including a mark for a word and a mark for the word's meaning. But they are both marks in her mind.

This is akin the Robot Reply to the CRA [Searle, 1980, 420], in which sensors and effectors are added to the Chinese Room so that Searle-in-the-room can both perceive the external world as well as act in it. Searle's response to the Robot Reply is to say that it is just more symbols. The reply to Searle is to say that that is exactly how human cognition works! In our brains, all cognition is the result of neuron firings, and the

---

[11]In fact, any study of semantics in linguistics—in the sense of a study of the meanings of linguistic expressions—that focuses only on relations among the expressions and not on their relations to the world is a syntactic enterprise. This is the nature of, for instance, cognitive semantics.

Figure 19.2:

Link betw.
(a) & (b)

(b) Semantic
network

(a) Internal
representation
of tree

Causal link

Cassie
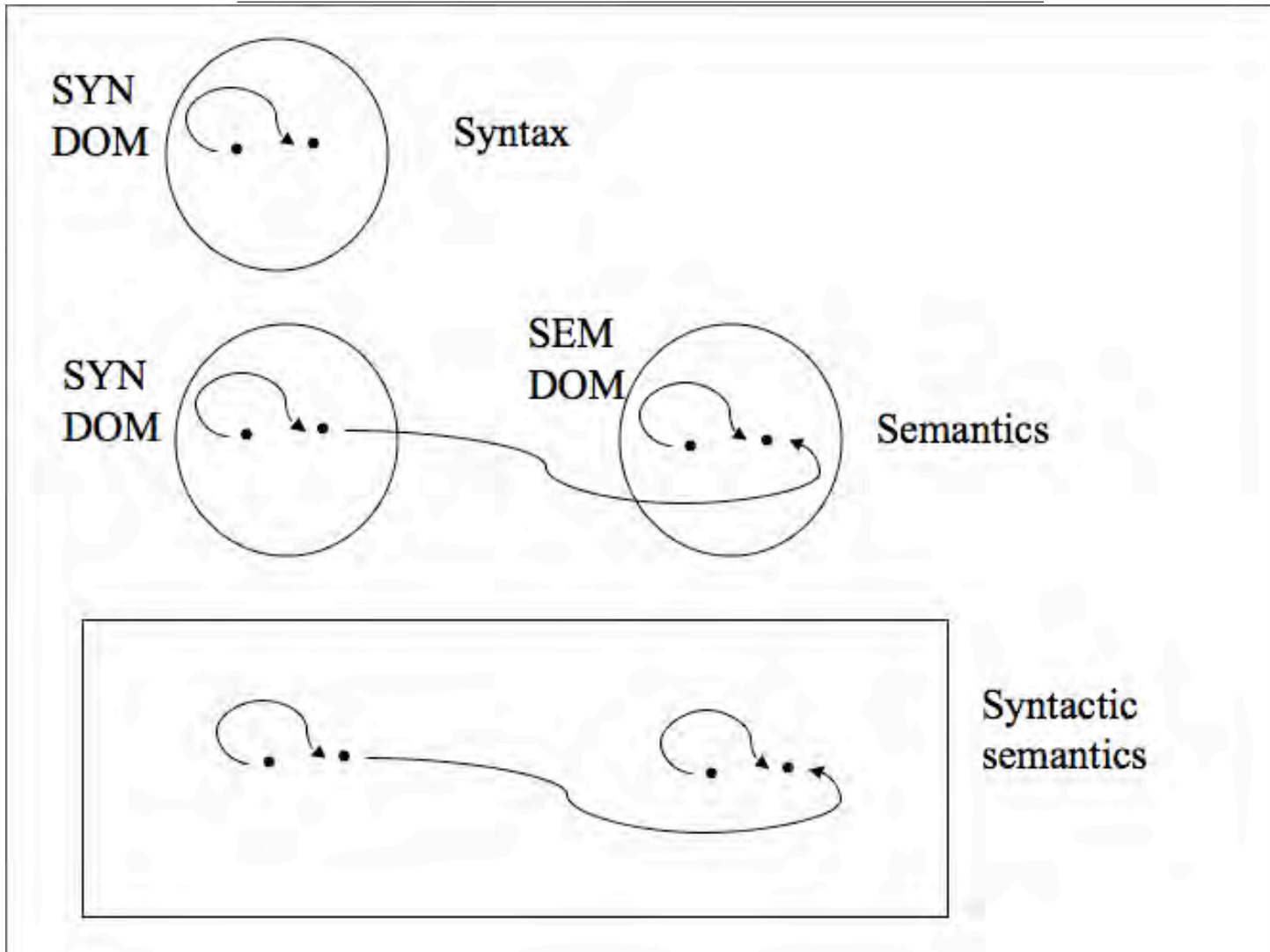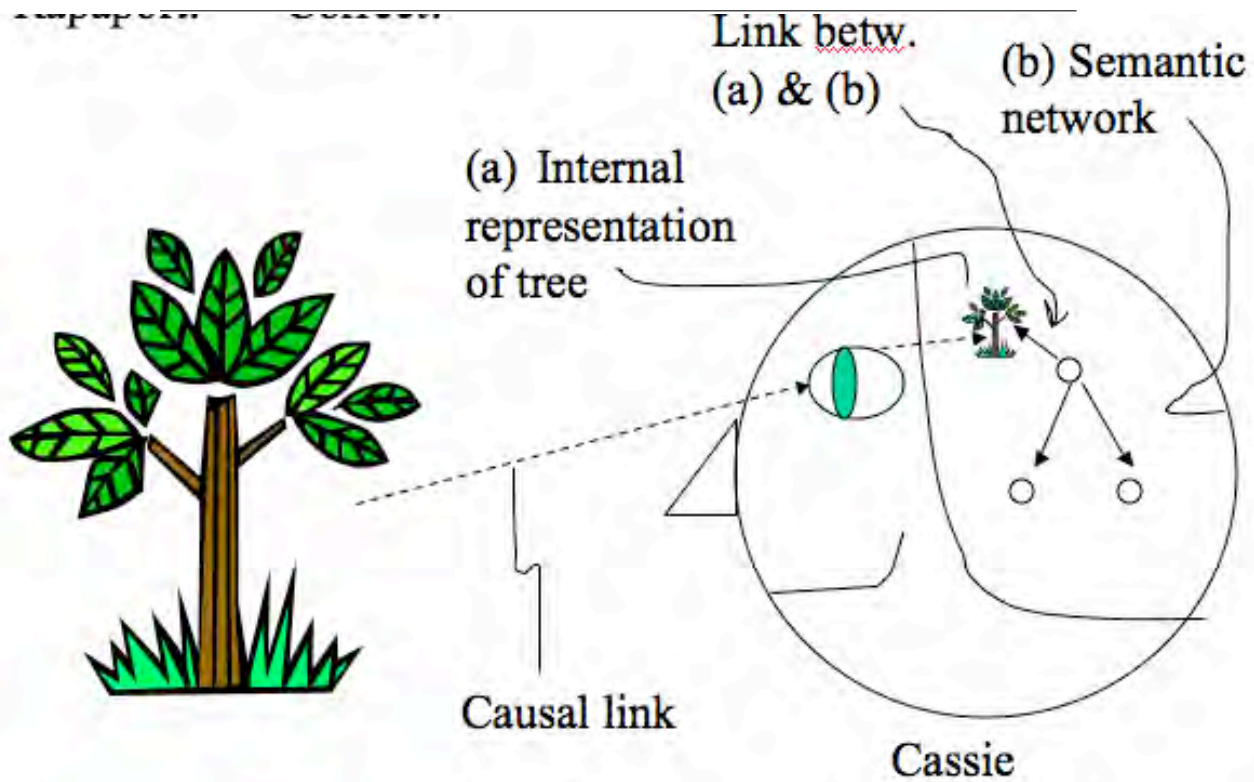
Figure 19.3:

study of that single set of neuron firings is a syntactic study (because it is the study of the properties of, and relations among, a single set of "marks"—in this case, the "marks" are neuron firings). The same is true for computers: If I say something to Cassie in English, she builds internal nodes that represent my utterance in her semantic network. If I show pictures to her, or if she sees something, she builds other internal nodes representing what she sees. This set of nodes forms a single computational knowledge base, whose study is syntactic in nature (because it is the study of the properties of, and relations among, a single set of "marks"—in this case, the "marks" are nodes in a semantic network). In the same way, both truth tables and the kind of formal semantics that logicians study are syntactic ways of doing semantics: The method of truth tables syntactically manipulates symbols that represent semantic truth values. And formal semantics syntactically manipulates symbols that represent the objects in the domain of semantic interpretation.

19.5.3.3 Points of View

The second prong of our reply to the Argument from Semantics concerns the differing points of view of the native, Chinese-speaking interrogator and Searle-in-the-room. To understand how a cognitive agent understands, and to construct a computational cognitive agent, we must take the first-person point of view. We must construct a cognitive agent (a robot, if you will) from the agent's point of view, from the perspective of what's going on "inside" the agent's head. In other words, we must be methodologically solipsistic and develop or implement a "narrow" or "internal" model of cognition. Such a model is called 'narrow', as opposed to 'wide', because it ignores the wider outside world and focuses only on the narrow inner world of the agent's point of view. We don't need to understand the causal or historical orgins of the agent's internal symbols; we only need to understand the symbols.

But there are two different points of view: There is Searle-in-the-room's point of view and there is the interrogator's point of view. In Searle's CRA, Searle-in-the-room's point of view trumps the interrogator's; in the Turing test (and in the kind of syntactic semantics that I am advocating), the interrogator's trumps Searle-in-the-room's. How should we resolve this?

Here is an analogy that I think helps clarify the situation. Consider the following passage from *The Wizard of Oz* (the novel, not the movie):

> CHECK THIS QUOTE
> When Boq [a Munchkin] saw her silver shoes, he said, "You must be a great sorceress."
> "Why?" asked [Dorothy].
> "Because you wear silver shoes and have killed the wicked witch. Besides, you have white in your frock, and only witches and sorceresses wear white."
> "My dress is blue and white checked," said Dorothy. . . .
> "It is kind of you to wear that," said Boq. "Blue is the color of the Munchkins, and white is the witch color; so we know you are a friendly witch."

> Dorothy did not know what to say to this, for all the people seemed to think her a witch, and she knew very well she was only an ordinary little girl who had come by the chance of a cyclone into a strange land. [Baum, 1900, 34–35]

Is Dorothy a witch? From her point of view, the answer is 'no'; from Boq's point of view, the answer is 'yes'. Whose point of view should trump the other's? Dorothy certainly believes that she's not a witch, at least as she understands the word 'witch' (you know—black hat, broomstick, Halloween, and all that). Now, it is certainly possible that Dorothy *is* (such a) witch while believing (mistakenly in that case) that she is *not* (such a) witch. So, what counts as being a witch (in these circumstances)? Note that the dispute between Dorothy and Boq is *not* about whether Dorothy is "really" a witch in some context-independent sense. The dispute is about whether Dorothy is a witch *in Boq's sense*, from *Boq's* point of view. And, because Dorothy is in Oz, Boq's point of view trumps hers!

Now compare this to the Chinese room situation: Here, instead of asking whether Dorothy is a witch, we ask: Does Searle-in-the-room understand Chinese? From his oint of view, the answer is 'no'; from the native Chinese speaker's oint of view, the answer is 'yes'. Whose point of view should trump the other's? Searle-in-the-room certainly believes that he does not understand Chinese, at least as he understands 'understanding Chinese' (you know—the way you understand your native language as opposed to the way you understand the foreign language that you may have (poorly) learned in high school or college). Now, it is certainly possible that Searle-in-the-room *does* understand Chinese while believing (mistakenly, in that case) that he does *not* understand it. So, what counts as understanding Chinese (in these circumstances)? For the same reason as in the witch case, it must be the native Chinese speaker's point of view that trumps Searle-in-the-room's!

Of course, it would be perfectly reasonable for Searle-in-the-room to insist that he doesn't understand Chinese. Compare Searle-in-the-room's situation to mine: I studied French in high school; spent a summer living with a French family in Vichy, France; spent a summer studying French (but mostly speaking English!) at the University of Aix-en-Provence; and have visited French friends in France many times. I believe that I understand about 80% of the French that I hear in a one-on-one conversation (considerably less if I'm hearing it on TV or radio) and can express myself the way that I want about 75% of the time (I have, however, been known to give directions to Parisian taxi drivers), but I always feel that I'm missing something. Should I believe my native French-speaking friends when they tell me that I am fluent in French? Searle would say 'no'.o

But Searle-in-the-room isn't me. Searle-in-the-room can't insist that *he alone* doesn't understand Chinese and that, therefore, his point of view should trump the native, Chinese-speaking interrogator's. And this is because *Searle-in-the-room isn't alone*: Searle-in-the-room has the Chinese natural-language-processing instruction book (even if he doesn't know that that's what it is). This is the core of what is known as the Systems Reply [Searle, 1980, 419–420]—that it is the "system" consisting of Searle-in-the-room *together with* the rule book that understands Chinese. After all, it is not a computer's CPU that would understand Chinese (or do arithmetic, or do word-

processing), but it is the system, or combination, consisting of the CPU executing a computer program that would understand Chinese (or do arithmetic, or process words). And Searle-in-the-room together with the rulebook, stranded on a desert island, *could* communicate (fluently) with a native, Chinese-speaking "Friday".[12]

Does it make sense for a "system" like this to exhibit cognition? Doesn't cognition have to be something exhibited by a single entity, like a person, an animal, or a robot? The cognitive anthropologist Edwin Hutchins, [Hutchins, 995a], [Hutchins, 995b], [Hollan et al., 2000], has argued that it is an "extended" or "distributed" cognitive system (such as a ship's crew together with their navigation instruments) that navigates a ship. This is a real-life counterpart of Searle-in-the-room together with his rulebook: "Cognitive science normally takes the individual agent as its unit of analysis.... [But] systems that are larger than an individual may have cognitive properties in their own right that cannot be reduced to the cognitive properties of individual persons" [Hutchins, 995b, 265, 266]. So, Searle-in-the-room plus his external rulebook can have the cognitive property of understanding Chinese, even though Searle-in-the-room all by himself lacks that property.

In fact, if the property of understanding Chinese (that is, the knowledge of Chinese) has to be located in some smaller unit than the entire system, it would probably have to be in the rulebook, not Searle-in-the-room! In an episode of the 1950s TV comedy series *I Love Lucy*, Lucy tries to convince her Cuban in-laws that she speaks fluent Spanish, even though she doesn't. To accomplish this, she hires a native Spanish speaker to hide in her kitchen and to communicate with her via a hidden, two-way radio, while she is in the living room conversing with her in-law "interrogators". Here, it is quite clear that the knowledge of Spanish resides in the man in the kitchen. Similarly, the knowledge of Chinese resides in the rulebook. It is the ability to execute or process that knowledge that resides in Searle-in-the-room. Together, the system understands Chinese.

It can be argued that cognitive agents have no diredt access to external entities. When I point to a tree, what I am aware of is, not my actual hand pointing to the actual tree, but an internal visual image of: my hand pointing to a tree. This is consistent with the 18th-century philosopher Immanuel Kant's theory of "phenomena" and "noumena". We are not directly aware of (for Kant, we have no knowledge of) the real world as it is in itself; he called this the world of "noumena" (singular: noumenon). All that we are aware of is the world filtered through our senses and our mental concepts; he called this the world of "phenomena" (singular: phenomenon). My access to the external world of noumena is mediated by internal representatives. There are several reasons for thinking that this is really the case (no matter how *Matrix*-like it may sound!): There is an "argument from illusion" that says that, because we see different things with each eye, what we see is, not what's out there, but the outputs of what our eyes have conveyed to our brains and that our brains have processed. There is an argument from time delay: Because it takes time (no matter how short) for light reflected off an object to reach our eyes, we see evnets *after* they happen; so, what we are seeing is in our heads, not out there.

Now, someone who takes a *third*-person point of view would say that you *can* have

---

[12] 'Friday' was the name of the native resident of the island that Robinson Crusoe was stranded on.

access to the external world. For instance, as a computer scientist programming a robot, it seems that I can have access to the world external to the robot as well as to the robot's internal mind (and I can compare the two, to determine if the robot has any misperceptions). If the robot (or you) and I are both looking at a tree, we see the same tree, don't we? From the *first*-person point of view, the answer is 'no': As the robot's programmer, I have access only to my internal representation of the external world and to my internal representation of the robot's internal world. And the same goes for you with respect to me, and for me with respect to you. If you and I are looking at a tree, we are each aware only of our two, separate internal representatives of that tree: one in your mine, one in mine; one produced by your neuron firings, one produced by mine. We cannot get outside of our heads to see what's really going on: "Kant was rightly impressed by the thought that if we ask whether we have a correct conception of the world, we cannot step entirely outside our actual conceptions and theories so as to compare them with a world that is not conceptualized at all, a bare 'whatever there is.' " [Williams, 1998, 40]

So, by merging internalized *semantic* marks with internal *syntactic* marks, the semantic project of mapping meanings to symbols can by handled by syntax, that is, by symbol manipulation. That is why syntax suffices for the first-person, semantic enterprise, and why Searle's Argument from Semantics is unsound. But there is another reason, too.

19.5.3.4 A Recursive Theory of Understanding

Semantics, as we have seen, requires there to be two domains and one binary relation: There is a syntactic domain of marks (which we have called SYN), characterized by syntactic formation and inference rules. There is a semantic domain of meanings or interpretations, (which we have called SEM), also characterized by syntactic formation and inference rules (called by AI researchers its 'ontology'). And there is a binary, semantic interpretation function, $I : SYN \rightarrow SEM$, that assigns a meaning from SEM to each mark in SYN.

We use SEM to understand SYN. Therefore, we must antecedently understand SEM; otherwise, we would be understanding one thing in terms of something else that we do not understand, and that should hardly count as understanding.

So, how do we understand SEM? In the same way that we understand SYN: by treating SEM as a new syntactic domain, and then finding a new semantic domain, SEM$'$, in terms of which to understand it. [Smith, 1987] called this a "correspondence continuum", because it can be continued indefinitely, understanding the SEM$'$ in terms of yet another SEM$''$, and so on. To stop an infinite regress, there must be a base case, a "last" semantic domain that we understand directly, in terms of itself rather than in terms of something else. But to understand a domain in terms of itself is to understand its members in terms of their properties and relations to each other. And that is syntax. It is a kind of understanding that I call 'syntactic understanding'. We understand a domain syntactically by being conversant with manipulating its marks (or by knowing which wffs are theorems). On this view, the "meaning" of a mark is its location in a network of other marks, with the connections between the marks being their properties and relations to the other marks in the network. (This is a kind of

"structural" understanding; cf. [Saussure, 1959].)

Here is another way to think about it: When I understand what you say, I do this by interpreting what you say, that is, by mapping what you say into my concepts. Similarly, I (semantically) understand a purely syntactic formal system by interpreting it, that is, by providing a (model-theoretic) semantics for it. Now, let's turn the tables: What would it be for a formal system to understand me? (Does that even make sense? Sure: Robots that could understand natural language, or even simple commands, are merely programs—formal systems—being executed.) The answer is this: A formal system could understand me in the same way that I could understand it—by treating what I say as a formal system and interpreting it. Note that links to the external world are irrelevant; the "semantic" interpretation of a formal system is a purely syntactic enterprise.

19.5.4   A Better Way

So, the really interesting question is: What's in the rulebook? What is needed for (computational) natural-language understanding? To understand language, a cognitive agent must (at least):

- *take discourse as input*; it does not suffice for it to be able to understand isolated sentences

- *understand ungrammatical input*; we do this all the time, often without realizing it, and, even when we realize it, we have to be able to recover from any misinterpretations

- *make inferences and revise our beliefs*; after all, what you say will often cause me to think about other things (a kind of inferencing) or to change my mind about things (belief revision)

- *make plans*: We make plans for speech acts (how should I ask you to pass the salt? Should I say, "Gimme the salt!" or "May I please have the salt?"?), we make plans to ask and to answer questions, and we make plans about how to initiate or end conversations.

- *understand plans*, especially the speech-act plans of our interlocutors (when you said, "It's chilly in here", did you really mean that you wanted me to close the window?)

- *construct a "user model"*, that is, a model of our interlocutor's beliefs

- *learn* about the world and about language

- *have background knowledge* (sometimes also called 'world knowledge' or 'commonsense knowledge')

- *remember* what it heard, what it learned, what it inferred, and what beliefs it has revised.

When you consider all of these things, you realize that, to understand natural language, you need to have a mind! And this mind can be constructed as a syntactic system. In other words, the rulebook must be a computer program for complete AI. Or as Stuart C. Shapiro might put it, natural-language understanding is an "AI-complete" problem [Shapiro, 1992].

A robot with such a syntactic (or computational) mind would be like Searle-in-the-room, manipulating symbols that are highly interconnected and that include internal representatives of external objects. It would be causally linked to the external world (for this is where it gets its input), which provides "grounding" and a kind of external, third-person, "semantic understanding". Such a robot could (or, more optimistically, will be able to) pass a Turing test and escape from the Chinese room.

19.6 Appendix: Some Definitions of AI

1. We propose that a 2 month, 10 man study of artificial intelligence be carried out during the summer of 1956 at Dartmouth College in Hanover, New Hampshire. The study is to proceed on the basis of *the conjecture that every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it.* An attempt will be made to find how to make machines use language, form abstractions and concepts, solve kinds of problems now reserved for humans, and improve themselves. We think that a significant advance can be made in one or more of these problems if a carefully selected group of scientists work on it together for a summer. [McCarthy et al., 1955, my emphasis]

2. The goal of work in artificial intelligence is to build machines that perform tasks normally requiring human intelligence. [Nilsson, 1971, vii] (See also [Nilsson, 1983].)

3. Research scientists in Artificial Intelligence try to get machines to exhibit behavior that we call intelligent behavior when we observe it in human beings. [Slagle, 1971, 1]

4. B[ertram] Raphael... has suggested that AI is a collective name for problems which we do not yet know how to solve properly by computer.[13] [Michie, 1971, 101]

5. What is or should be [AI researchers'] main scientific activity—*studying the structure of information and the structure of problem solving processes independently of applications and independently of its realization in animals or humans.* [McCarthy, 1974, 317]

6. By "artificial intelligence" I therefore mean the use of computer programs and programming techniques to cast light on the principles of intelligence in general and human thought in particular. [Boden, 1977, 5]

---

[13]Note that it follows that, once we do know how to solve them, they are no longer AI!

7. Artificial Intelligence is the branch of Computer Science that attempts to solve problems for which there is no known efficient solution, but which we know are efficiently solvable, (typically) because some intelligence can solve the problem (often in "real time"). A side benefit of AI is that it helps us learn how intelligences solve these problems, and thus how natural intelligence works. [Leler, 1985]

8. Artificial intelligence is concerned with the attempt to develop complex computer programs that will be capable of performing difficult cognitive tasks. [Eysenck, 1990, 22]

9. AI is making computers act like those in movies. [Brown, 1992]

10. ADD RUSSELL NORVIG?

Some discussions of the nature of AI cannot be summarized neatly in a one-sentence definition:

- A valuable discussion of the nature of AI may be found in a debate between two AI researchers, Roger C. Schank and Alan Bundy; see [Schank, 1983], [Bundy, 1983].

- A discussion by a software engineer comparing two different definitions, in the context of the US government's Strategic Defense Initiative, is in [Parnas, 1985].

- John McCarthy has written several essays on the nature of AI; see [McCarthy, 1988], [McCarthy, 2007].

- A philosopher-turned-AI researcher, Aaron Sloman, discussed the nature of AI in a newsgroup, [Sloman, 1989].

- A discussion among three computer scientists on what AI is [Shapiro et al., 1992]

19.7 NOTES FOR NEXT DRAFT

1. [Quillian, 1994, 440–442] has some interesting arguments based on the history of the physics of flight to the effect that studying what might be called "artificial" flight was crucial. "[S]tudying the animals that fly", no matter in how great detail and for how many years, would not have yielded any useful information on how humans might be able to fly. Rather, "attempt[ing] to construct devices that fly...attempts to build flying machines" resulted in "our entire understanding of flight today. Even if one's aim is to understand how birds or insects fly, one will look to aeronautics for the key principles...." This supports my claim (in this book? Or merely in some of my syntactic-semantics papers?) that the study of computational theories of cognition helps us generalize the more parochial study of human (or even, more generally, animal) cognition. (Quillian—a pioneer in AI research—uses this argument to support his explanation of why the natural sciences are more "effective" than the social sciences.)

2. Is the full power of a TM needed for AI? [Sloman, 2002, §3.3] says "no". This seems correct; after all, even NLP doesn't need the full power of a TM (CITE?). On the other hand, can't a human do anything that a TM can do? TMs, after all, are computational models of human computing ability.

3.
   > The analytical engine has no pretentions whatever to originate anything. It can do whatever we know how to order it to perform. (Ada Lovelace, cited in [Menabrea, 1843, 722].

   That it has no "pretentions" means that it wasn't designed that way; nevertheless, it might still be able to "originate" things. Also, if we can find out "how to order it to perform" cognitive activities, then it can do them!

   Finding out how requires us to be conscious of something that we ordinarily do unconsciously:

   > Regardless of what one calls the work of a digital computer [specifically, regardless of whether one says that it can think], the unfortunate fact remains that more rather than less human thinking is required to solve a problem using a present day machine since every possible contingency which might arise during the course of the computation must be thought through in advance. The jocular advice recently published to the effect, "Don't Think! Let UNIVAC do it for you," cannot be taken seriously. Perhaps, if IBM's familiar motto [namely, "Think!"] needs amending, it should be "Think: Think harder when you use the 'ULTIMAC'. [Samuel, 1953, 1225].

   Surely this is related to the theme of [Dennett, 2009].

4. There is a nice discussion of "marks" and "mark manipulation" systems in [Kearns, 1997, §2, pp. 273–274]

5.
   > Recently, Michael Scherer, a *Time* magazine bureau chief, received a phone call from a young lady, Samantha West, asking him if he

> wanted a deal on health insurance. After she responded to a number of his queries in what sounded like prerecorded fashion, he asked her point-blank whether she was a robot, to which he got the reply "I am human." When he repeated the question, the connection was cut off. Samantha West turned out to be a system of recorded messages that were part of a computer program created by the brokers for health insurance. [Skidelsky, 2014, 36]

6. On Turing's strange inversion:

> [T]he thought experiment [most recently described by Ned Block, in which a computer programmed with a large table-lookup passes the Turing test] is meant to show that seemingly intelligent behavior can be produced by unintelligent means, whereas real intelligence cannot.... [McDermott, 2014, 144]

This is also the underlying motivation of Turing's "strange inversion". Actually, the latter is an even stronger version than the table-lookup version, because the latter allows for the underlying computer program to be more than merely table-lookup. McDermott's point is that even table-lookup is more than "mere".

7. On the CRA:
Rescorla's (and Cleland's) emphasis on the interpretation of the symbols used by a computer is relevant to the CRA. Here is a nice description of computation that matches the CRA:

> Consider again the scenario described by Turing: an idealized human computor manipulates symbols inscribed on paper. **The computor manipulates these symbols because he [sic] wants to calculate the value some number-theoretic function assumes on some input.** The computor starts with a symbolic representation for the input, performs a series of syntactic operations, and arrives at a symbolic representation for the output. This procedure succeeds only when the computor can *understand* the symbolic representations he manipulates. The computor need not know in advance which number a given symbol represents, but he must be capable, in principle, of determining which number the symbol represents. Only then does his syntactic activity constitute a computation of the relevant number-theoretic function. If the computor lacks any potential understanding of the relevant syntactic items, then his activity counts as mere manipulation of syntax, rather than calculation of one number from another. [Rescorla, 2007, 261–262, my boldface, Rescorla's italics]

Without the boldfaced sentence, this is a nice description of the CR. The difference is that, in the CR, Searle-in-the-room does not "want to" communicate in Chinese; he doesn't know what he's doing, in that sense of the phrase. Still,

he's doing it, according to the intepretation of the native speaker outside the room.

8.   The basic strategy of AI has always been to seek out progressively more complex human tasks and show how computers can do them, in humanoid ways or by brute force. With a half-century of steady progress, we have assembled a solid body of tested theory on the processes of human thinking and the ways to simulate and supplement them. (Herbert Simon, quoted in [Hearst and Hirsh, 2000, 8].)

The phrase 'human tasks' nicely avoids any issues involved with the notion of "intelligence".

9. On the idea that neurons have a syntax (§19.5.3.2): [Rescorla, 2012a, 19] says, "neural properties are not multiply realizable. So neural properties are not syntactic properties." But he gives no argument for why they are not multiply realizable, nor is it clear why he thinks that they are not syntactic *as a consequence of their non-realizability*. In fact, while I would agree that *neurons* are not multiply realizable, I fail to see why their *properties* are not. Properties, after all, are universals, and, if anything is multiply realizable, surely universals are.

10. On strong-vs.-weak AI:
Here is Simon's version of the distinction:

> Those of us who regard computer programs as "theories" rather htna 'metaphors' are probably still in the minority. "*Weak* AI," as the metaphoric view is sometimes called, still probably has more advocates than "*strong* AI." [Simon, 1996a, 161]

So, strong AI is the view that a computational theory of cognition expressed as a computer program is a legitimate theory, whereas weak AI is the view that such a program is merely a metaphor. Presumably, if a computational theory of cognition is correct, then, when it is being executed, it is really exhibiting cognitive behavior (and not merely simulating it).

11.   The subject of computation is essentially that of artificial intelligence since the development of computation is in the direction of making machines carry out ever more complex and sophisticated processes, i.e. to behave as intelligently as possible. [McCarthy, 1963, §4.2, p. 38].

This follows from his earlier definition (see Ch. 3) of computation as the science of how to get machines to carry out intellectual processes. The latter is surely equivalent to AI.

12. On the idea of abstracting from a single example and re-implementing computationally:

> Quite typically, an abstract structure underlies some human cognitive activity that is not at all apparent in superficial phenomenology or practice. Often, that structure is related in interesting ways to the structures we would invent if we constructed an ideal machine to perform that cognitive activity. (We might think of artificial intelligence as a normative enterprise). But that structure is rarely identical to the ideal machine's structure. [Gopnik, 1996, 489]

The underying abstract structure could be computational in nature. Hence, it could be (re-)implemented in "an ideal machine". The abstract computational theory might be thought of as having the form: such-and-such a human cognitive activity can/ought to be performed in this computational way *even if* the way that humans in fact do it is not identical to that ideal structure.

Gopnik goes on to say:

> This process may seem like analogy or metaphor, but it involves more serious conceptual changes. It is not simply that the new idea is the old idea applied to a new domain, but that the earlier idea is itself modified to fit its role in the new theory. [Gopnik, 1996, 498]

I have said elsewhere that the process is metaphorical. I don't think that that is inconsistent with it also involving "more serious conceptual changes": We come to see the old idea in a new way. This is all relevant to my interpretation of what Turing meant by the Turing test.

19.8   Further Sources of Information

1. ADD MATERIAL FROM MY WEBSITES AT http://www.cse.buffalo.edu/∼rapaport/584/philai.html

2. Cole, David (2014), "Alan Turing & the Chinese Room Argument", http://www.thecritique.com/articles/alan-turing-the-chinese-room-argument/ (accessed 19 January 2014).

   - Part of a series of online articles providing background for the movie *The Imitation Game*. This is a critique of the Chinese Room Argument by a philosopher who has written widely on the topic.

3. Horst, Steven (2014), "The Computational Theory of Mind: Alan Turing & the Cartesian Challenge", http://www.thecritique.com/articles/the-computational-theory-of-mind-alan-turing-the-cartesian-challenge/ (accessed 19 January 2015).

   - Part of a series of online articles providing background for the movie *The Imitation Game*. This is a critique of Turing 1950 and the Chinese Room Argument, suggesting that Searle's two main "points...are (1) that computation alone is not sufficient for understanding and (2) that passing the Turing test would not be proof that understanding (hence thinking) is involved."

4. Husbands, Phil; & Holland, Owen (2012), "Warren McCulloch and the British Cyberneticians", *Interdisciplinary Science Reviews* 37(3) (September): 237–253; online at: http://www.mathcomp.leeds.ac.uk/turing2012/Images/mcculloch.pdf (accessed 12 November 2013).

5. Kahn, Amina (2014), "Rover Disguised as Penguin Chick Does Research better than Scientists", *Los Angeles Times Science Now* (3 November), http://www.latimes.com/science/sciencenow/la-sci-sn-penguin-robot-chick-study-20141103-story.html (accessed 10 November)

   - Did this penguin robot pass a kind of Turing test?

6. McDermott, Drew (2014), "What Was Alan Turing's Imitation Game?", http://www.thecritique.com/articles/what-was-alan-turings-imitation-game/ (accessed 19 January 2015).

   - Part of a series of online articles providing background for the movie *The Imitation Game*. This critique of Turing 1950 was written by a well-known AI researcher.

7. McCarthy, John; & Hayes, Patrick J. (1969), "Some Philosophical Problems from the Standpoint of Artificial Intelligence", NEED CITATION

8. Pylyshyn, Zenon W. (1992), "Computers and the Symbolization of Knowledge", in Ralph Morelli, W. Miller Brown, Dina Anselmi, Karl Haberlandt, & Dan Lloyd (eds.), *Minds, Brains & Computers: Perspectives in Cognitive Science and Artificial Intelligence* (Norwood, NJ: Ablex): 82–94.

   - Preprint (1987) accessed 9 April 2014 from: http://ruccs.rutgers.edu/ftp/pub/papers/suffolk.pdf (see esp. p. 7)

9. Rey, Georges (2012), "The Turing Thesis vs. the Turing Test", *The Philosopher's Magazine* 57 (2nd Quarter): 84–89.

Chapter 20

Computer Ethics II:
Should We Build
Artificial Intelligences?

Douglas Engelbart, who "more than anyone else invented the modern user interface, modern networking and modern information management... met Marvin Minsky—one of the founders of the field of AI—and Minsky told him how the AI lab would create intelligent machines. Engelbart replied, 'You're going to do all that for the machines? What are you going to do for the people?' "
[Lanier, 2005, 365]

20.1   Recommended Readings:

1. Required:

   - LaChat, Michael R. (1986), "Artificial Intelligence and Ethics: An Exercise in the Moral Imagination", *AI Magazine* 7(2): 70–79.

2. Very Strongly Recommended:

   - Lem, Stanisław (1971), "Non Serviam", in S. Lem, *A Perfect Vacuum*, trans. by Michael Kandel (New York: Harcourt Brace Jovanovich, 1979).

       – Reprinted as:
         "The Experiment (A Review of 'Non Serviam', by James Dobb)", *The New Yorker* (24 July 1978): 26ff.
       – Reprinted in:
         Hofstadter, Douglas R.; & Dennett, Daniel C. (eds.) (1981), *The Mind's I: Fantasies and Reflections on Self and Soul* (New York: Basic Books): 296–317.

3. Strongly Recommended:

   - Moor, James H. (1985), "What Is Computer Ethics?", *Metaphilosophy* 16(4) (October): 266–275.

4. Highly Recommended:

   - Dietrich, Eric (2001), "Homo sapiens 2.0: Why We Should Build the Better Robots of our Nature", *Journal of Experimental and Theoretical Artificial Intelligence* 13(4) (October): 323–328.

   - Turkle, Sherry (2004), "How Computers Change the Way We Think", *Chronicle of Higher Education* (January 30): B26–B28.

   - Petersen, Stephen (2007), "The Ethics of Robot Servitude", *Journal of Experimental and Theoretical Artificial Intelligence* 19(1) (March): 43–54; online at:
     http://stevepetersen.net/professional/petersen-robot-servitude.pdf
     accessed 14 November 2012.

5. Recommended:

   - Powers, Richard (1995), *Galatea 2.2* (New York: Farrar, Straus, Giroux).

20.2   Introduction

Stanisław Lem's short story, "Non Serviam" [Lem, 1971], concerns what is now called "artificial life" (or "A-life"). In the story, an A-life researcher constructs a computational world of intelligent entities, and follows their evolution and development of language and philosophy. These "artificial intelligences" discuss the existence of God in much the same way that human philosophers have. The difference (if it is a difference) is that both the researcher and reader realize that he, the researcher, is their God, and, although he created them, he is neither omniscient nor omnipotent, and, worse, when his funding runs out, he will have to literally pull the plug on the computer and thereby destroy them.

This raises the question of whether such an experiment should even be begun, which is the second ethical issue and the final philosophical topic that we will look at.

20.3   Artificial Persons

One of the earliest philosophical investigations of this issue is an essay by Michael R. LaChat that appeared in *AI Magazine*, [LaChat, 1986]. LaChat argued that it is worthwhile to consider the moral implications of creating an artificial intelligence, an artificial person. One reason is that it *might* happen, so we should be prepared for it. Another reason is that, even if it turns out to be improbable, such a discussion illuminates what it means to be a person, which is an important goal in any case. Let's consider each of these.

20.4   Is AI Possible in Principle?

An artificial intelligence might be created if and only if the field of artificial intelligence is capable of succeeding in its goal of developing a computational theory of cognition.

Most objections to AI stem from what is known as "mind-brain dualism". This is, roughly, the view that the mind and the brain are two, distinct kinds of entities that somehow interact. On this view, the mind is not physical, but the brain is. And nothing physical—neither the brain nor a computer—can give rise to something non-physical; hence, AI cannot create a mind.

But, LaChat notes, there is no evidence of conscious thought without a physical brain. So, having a brain seems to be a necessary condition of having a mind, even if we don't believe that thoughts are *identical to*—that is, are the very same things as—brain states or processes. Hence, it might be sufficient to simulate a brain (perhaps computationally) in order to create a mind.

On LaChat's view, AI *is* possible in principle *if* it is possible that there exists a "functional isomorphism" between (1) the neural network that constitutes our brain (that is, brain states and processes) and (2) any other physical implementation of the functional (that is, psychological) behavior that that neural network implements

[LaChat, 1986, 72]. In other words, psychology is an Abstraction that can be implemented in either brains or other physical media.

"Functionalism" is roughly the view that the mind—better: cognition—is the function of the brain, that is, the mind is what the brain does. Or, as the philosopher Hilary Putnam suggested [Putnam, 1960], a Turing-machine program stands in the same relation to computer states and processes as mental states and processes stand to brain states and processes (sometimes summarized as "the mind is to the brain as software is to hardware"). (For a good overview of functionalism, see [Fodor, 1981].)

Functionalism, as a way of resolving the mind-brain problem, has the advantage of allowing all mental states and processes to be implemented in some physical states and processes; this is the principle of "multiple realization". And it has another advantage: It allows the mind to be an "emergent" property of a "certain level of organization" [LaChat, 1986, 72].

There are, of course, problems, both for functionalism in particular and for AI in general. One is the problem of personality: Would "[a] personal intelligence...have personality"? [LaChat, 1986, 73] thinks this is "almost impossible", but there is considerable computational work on emotions, so I would not rule this out of hand. (See Reid Simmons's "Social Robots Project", [http://www.conscious.cmu.edu/afs/conscious/project/robocomp/social/www/]; the "Cog" project at MIT, [http://www.ai.mit.edu/projects/humanoid-robotics-group/cognitive/]; and work by Rosalind Picard [Picard, 1997], Aaron Sloman [Sloman and Croucher, 1981], and Paul Thagard [Thagard, 2006].) Another is the problem of pain and other "qualia", that is, qualitative "feelings" and "experiences" such as colors and sounds. (Do red fire engines look the same to you and to me? Or do fire engines for you seem to have the color that grass has for me? Why does the sound of a bell give rise to the experience that it does rather than the experience that the smell of garlic has?) The problem here is whether computers could experience qualia and, even if they could, how we would know that. This is a vast topic well beyond our present scope, but for a brief argument that qualia are not out of the question for artificial intelligences, see Shapiro's example of a computer that can feel pain, discussed in [Rapaport, 2005b, §2.3.1].

20.5   What Is the Measure of Success?

How would we know if we have achieved a "personal artificial intelligence"? One way, of course, would be by having it pass a Turing test. LaChat offers a different criterion: Joseph Fletcher's analysis of personhood [Fletcher, 1972].

The question of what kinds of entities count as "persons" is not limited to AI. The issue arises most prominently in the abortion debate: If fetuses are persons, and if killing persons is immoral, then abortion is immoral (to oversimplify matters). It also arises in animal ethics: Are dolphins intelligent enough to be considered persons? (And how about extraterrestrials? Or corporations?) The point is that there is a distinction between the biological category of being human and an ethical or legal category of being a person. The question is: How can personhood be characterized in an implementation-independent way?

On Fletcher's analysis, *x* is a person[1] if and only if *x* has (1) the following positive characteristics:

1. minimal intelligence

    *   This might mean, for example, having an IQ greater than about 30 or 40 (if you believe that IQ measures "intelligence"). That is, to be minimally intelligent is not to be *mere* biological life (presumably, a bacterium would not be minimally intelligent; according to Douglas Hofstadter, what Fletcher is calling 'minimal intelligence' would only apply to lifeforms evolutionarily "higher" than a mosquito [Hofstadter, 2007]). For example minimal intelligence might include some level of rationality, or perhaps even language use.

2. a sense of self

    *   That is, persons must be self-aware and exhibit self-control.

3. a sense of time

    *   Persons must have a sense of the past, hence some kind of culture; a sense of the future (so that they have the ability to make plans); and a sense of the passage of time.

4. a social role

    *   Persons must have an ability to relate to others, to have concern for others, and to communicate with others (hence the need for language as part of minimal rationality).

5. curiosity

    *   That is, persons must not be indifferent.

6. changeability

    *   Persons must be creative and be able to change their minds.

7. idiosyncrasy, or uniqueness

8. neo-cortical function

    *   The cerebral cortex is where all the "cognitive action" occurs in the brain, so, for Fletcher, a person must have (something whose function is equivalent to) a cortex. (For more on neo-cortical function, see [Cardoso, 1997].)

and (2) the following negative characteristics:

1. neither essentially non-artificial nor essentially anti-artificial

---

[1]Fletcher actually uses the term 'human', not 'person', but I don't think that this is terminologically important.

- This clause allows for multiple realization and does not restrict personhood to biological entities. It also allows for "test-tube" babies to count as persons.

2. not essentially sexual

   - So, a cloned entity could be a person.

3. not essentially a bundle of rights

   - Fletcher argues that there are no "essential rights"; hence, the notion of rights cannot be used to characterize persons.

4. not essentially a worshipper

   - You don't have to be religious to be a person.

20.6   Rights

Does a "personal AI" have rights? That is, does an artificial intelligence that either passes a Turing test or that satisfies (something like) Fletcher's 12 characteristics have rights? For instance, would it have the right not to be a slave?

At first glance, you might think so. But isn't that what most robots are intended to be? So, if they are persons of either the Turing or the Fletcher variety, do they have the right not to do what we created them to do? At least one philosopher has suggested that they do not, that "robot servitude is permissible" [Petersen, 2007].

By 'robot servitude', Petersen does not mean voluntary assistance, where you do something or help someone because you want to, not because you are being paid to. Nor does he mean slavery, forced work that is contrary to your will. By 'robot servitude', he is thinking of robots who are *initially* programmed to *want* to serve us, in particular, to want to do tasks that humans find either unpleasant or inconvenient. For example, think of a robot programmed to love to do laundry. (On the other hand, compare this to a dog genetically "programmed" to want to fetch. This is reminiscent of the "epsilon" caste in Aldous Huxley's *Brave New World* [Huxley, 1932, Ch. 5, §1], who are genetically programmed to have limited desires—those destined to be elevator operators desire nothing more than to operate elevators.)

So, how would robots programmed to want to do unpleasant or humanly inconvenient tasks different from genetically engineering humans from scratch to desire to do such tasks? One assumption is that doing this to humans is morally wrong; hence, doing it to robots would also be morally wrong. Another assumption underlying this argument is that the two cases *are* alike.

But on most standard theories of ethics, either this is *not* wrong or it is *not* like the robot case. For instance, Aristotle might argue that engineering humans is wrong because humans have a particular function or purpose, and it would be wrong to engineer them away from it. In this case, there is no parallel with robots. In fact, a robot's function might be to do the laundry!

On a Kantian theory, such an engineered human would either be potentially autonomous (following its own moral rules that are universally generalizable) or would not be potentially autonomous. In the first case, it would be wrong to *prevent* such a robot from doing laundry, and it would not be harmful to let it do what it autonomously wants to do. In the second case, we can't do wrong to the robot by having it do our laundry any more than we can do wrong to a washing machine.

Alternatively, it could be argued that more autonomy is always better. But an engineered human would only have limited autonomy and therefore could have had a better life had it not been engineered. And similarly for a robot. (Does this mean that a Turing machine has less autonomy than a universal Turing machine?)

John Stuart Mill might argue that such an engineered human life would be *unfulfilling*, because it substitutes a "lower" pleasure for a "higher" one. But what's wrong with that? We're not taking a potential philosopher and turning *her* into a laundry-bot. Rather, we're introducing a laundry-bot into the world. Surely, this is no worse than when a dolphin is born.

What if such a robot could reason its way into renouncing laundry? That would be OK; otherwise, it would be robot *slavery*.

20.7 Immoral Experiments

Is the construction of a personal AI (that is, an artificial intelligence that (who?) satisfies the criteria for being considered a "person") an immoral experiment?

Clearly, some experiments are immoral. The existence of institutional review boards at universities is testament to this. There are many experiments that have been performed that are now considered to have been immoral (such as WHOSE experiments in which subjects were told to give what they thought were deadly electric shocks to people whom they thought were other subjects). And there are many that are clearly so (such as raising newborns on desert islands to see what kind of language they might naturally develop; such as experiment was proposed by WHOM?).

The most famous—and most relevant—literary example of such an experiment is the construction of Frankenstein's "monster". In Mary Shelley's novel, the monster (who is the most sympathetic character) laments as follows:

> Like Adam, I was apparently united by no link to any other being in existence, but his state was far different from mine in every other respect. He had come forth from the hands of God a perfect creature, happy and prosperous, guarded by the especial care of his creator, he was allowed to converse with, and acquire knowledge from, beings of a superior nature, but I was wretched, helpless, and alone. Many times I considered Satan was the fitter emblem of my condition. For often, like him, when I saw the bliss of my protectors, the bitter gall of envy rose up within me. ...Hateful day when I received life! ...Accursed creator! Why did you form a monster so hideous that even you turned from me in disgust? [Shelley, 1818, Ch. 15]

Frankenstein tries to justify his experiment in terms of how it advanced knowlege,

but he realizes that the advancement of knowledge must be balanced against other considerations:

> When younger,. . . I believed myself destined for some great enterprise. . . . I possessed a coolness of judgment that fitted me for illustrious achievements. This sentiment of the worth of my nature supported me when others would have been oppressed; for I deemed it criminal to throw away in useless grief those talents that might be useful to my fellow-creatures. When I reflected on the work I had completed, no less a one than the creation of a sensitive and rational animal, I could not rank myself with the herd of common projectors. But this thought, which supported me in the commencement of my career, now serves only to plunge me lower in the dust. All my speculations and hopes are as nothing; and, like the archangel who aspired to omnipotence, I am chained in an eternal hell. My imagination was vivid, yet my powers of analysis and application were intense; by the union of these qualities I conceived the idea and executed the creation of a man. Even now I cannot recollect without passion my reveries while the work was incomplete. I trod heaven in my thoughts, now exulting in my powers, now burning with the idea of their effects. From my infancy I was imbued with high hopes and a lofty ambition; but how am I sunk! Oh! my friend, if you had known me as I once was you would not recognise me in this state of degradation. Despondency rarely visited my heart; a high destiny seemed to bear me on until I fell, never, never again to rise.

Is there ever a situation in which the negative "side-effects" of an otherwise praiseworthy goal do not override that goal? (Compare this question with whether there are ever "just" wars.) The early cybernetics researcher Norbert Wiener struggled with this issue:

> If we adhere to all these taboos, we may acquire a great reputation as conservative and sound thinkers, but we shall contribute very little to the further advance of knowledge. It is the part of the scientist—of the intelligent man of letters and of the honest clergyman as well—to entertain heretical and forbidden opinions experimentally, even if he is finally to reject them. [Wiener, 1964, 5].

The basic ethical principle here seems to be what LaChat calls "non-maleficence", or Do No Harm. This is more stringent than "beneficence", or Do Good, because benevolence (doing good) might allow or require doing harm to a few for the benefit of the many (at least, according to utilitarianism), whereas non-malevolence would restrict doing good in order to avoid doing harm.

Is creating an artificial intelligence beneficial or not to the AI that is created? Does creating it do harm to that which is created? One way to think about this is to ask whether conscious life is "better" than no life at all. If it isn't, then creating an artificial life is not a "therapeutic experiment", hence not allowable by human-subjects review boards. Why? Because the subject of the experiment—the artificial person that the

experiment will create if it is successful (or, perhaps more so, if it is only partially successful)—does not exist before the experiment is begun, and so the experimenter is not "making it better". Here, we approach the philosophy of existentialism, one of whose tenets is that "existence precedes essence". This slogan means that who you are, what kind of person you are, is something that is only determinable after you are born (after you come to exist)—and it is not immutable, because, by your actions, you can change whom you are. The opposite slogan—"essence precedes existence"—is Aristotelian in flavor: You are a certain kind of person, and cannot change this fact.

On the existentialist view, you exist first, and *then* you determine what you will be. Frankenstein did an existential experiment, creating an AI without an essence, and both Frankenstein and his "monster" were surprised with the results. On the Aristotelian view, an essence (something like an Abstraction, as discussed in Ch. 14) is *implemented* (or "realized"). In AI, we can—indeed, must—plan out the essence of an entity before bringing it into existence (before implementing it). In either case, we can't guarantee that it would come out OK. Hence, creating an AI is probably immoral! So, LaChat sides with Frankenstein's "monster", not Frankenstein (or Wiener).

20.8 AIs and Morality

You might think that all of this is a bit silly or, at least, premature. But it is always better to be prepared: It is better to think about the consequences of our actions while we have the time and leisure to do so, so that, if those consequences come to be, then we won't be taken by surprise.

We have looked at whether it is moral to create an AI. Suppose we succeed in doing so. Could the AI that we create itself be moral?

If AIs are programmed, then one might say that they are not free, hence they are *amoral*. This is different from being *immoral*! Being "amoral" merely means that morality is irrelevant to whom or what you are. To oversimplify a bit, good people are moral, bad people are immoral, a pencil is amoral. The current question is whether AIs are amoral or not.

And so we have bumped up against one of the Big Questions of philosophy: Is there such a thing as free will? Do humans have it? Might robots have it? We will not attempt to investigate this issue here, but merely note that at least one AI researcher, Drew McDermott, has argued that free will may be a necessary illusion arising from our being self-aware [McDermott, 2001].

A different perspective has been taken by Eric Dietrich in a pair of papers [Dietrich, 2001], [Dietrich, 2007]. He argues that robots could be programmed to be *better* than humans (presumably because their essence precedes their existence). Hence, we could *decrease* the amount of evil in the world by building moral robots and letting them inherit the Earth.

20.9   NOTES FOR NEXT DRAFT

1. Discuss Bostrum's computer simulation argument; see Further Sources of Inofrmation for Ch. 15.

2. Add to list of movies, etc., re robots:

   - Mendelsohn, Daniel (2015), "The Robots Are Winning!", *New York Review of Books* 62(10) (4 June): 51–54, http://www.nybooks.com/articles/archives/2015/jun/04/robots-are-winning/ (accessed 14 June 2015).
     - A review of the films *Her* and *Ex Machina*, but also a history of robots in literature from *The Iliad* on.

20.10 Further Sources of Information

1. Books and Essays:

   - Bostrom, Nick (2003), "Are You Living in a Computer Simulation?", *Philosophical Quarterly* 53(211): 243–255.

     – An interesting philosophical analysis not unrelated to Lem's story.

     – Available online, along with many related items, at:
       http://www.simulation-argument.com/
       accessed 14 November 2012.

   - Choi, Charles Q. (2008), "Not Tonight, Dear, I Have to Reboot", *Scientific American* (March): 94–97.

     – "Is love and marriage with robots an institute you can disparage? Computing pioneer David Levy doesn't think so—he expects people to wed droids by midcentury. Is that a good thing?"

   - Clarke, Roger (1993/1994), "Asimov's Laws of Robotics: Implications for Information Technology":

     – Part 1, *IEEE Computer* 26(12) (December 1993): 53–61.

     – Part 2, *IEEE Computer* 27(1) (January 1994): 57–66.

   - Hofstadter, Douglas R.; & Dennett, Daniel C. (1981), "Reflections [on Lem's 'Non Serviam']", in Douglas R. Hofstadter & Daniel C. Dennett (eds.) (1981), *The Mind's I: Fantasies and Reflections on Self and Soul* (New York: Basic Books): 317–320.

   - La Chat, Michael Ray (2003), "Moral Stages in the Evolution of the Artificial Superego: A Cost-Benefits Trajectory", in Iva Smit, Wendell Wallach, & Goerge E. Lasker (eds.), *Cognitive, Emotive and Ethical Aspects of Decision Making in Humans and in Artificial Intelligence*, Vol. II (Windsor, ON, CANADA: International Institute for Advanced Studies in Systems Research and Cybernetics): 18–24.

     – A follow-up essay by LaChat, in which he argues that a "moral" robot "will have to possess sentient properties, chiefly pain perception and emotion, in order to develop an empathetic superego which human persons would find necessary and permissible in a morally autonomous AI".

   - Tanaka, Fumihide; Cicourel, Aaron; & Movellan, Javier R. (2007), "Socialization between Toddlers and Robots at an Early Childhood Education Center", *Proceedings of the National Academy of Sciences* 104(46) (13 November): 17954–17958.

2. Websites:

   (a) On Karel Čapek and "R.U.R":

      i. "Karel Capek"
         http://www.catch22.com/SF/ARB/SFC/Capek,Karel.php3
         accessed 14 November 2012.

      ii. "Josef and Karel Capek"
          http://www.imagi-nation.com/moonstruck/clsc23.html
          accessed 14 November 2012.

    iii. Jerz, Dennis G. (2002), "R.U.R. (Rossum's Universal Robots)"
http://jerz.setonhill.edu/resources/RUR/index.html
accessed 14 November 2012.

- Discussion of various versions of the play, with numerous links.
- Online versions of the play:

    A. Čapek, Karel (1920), "R.U.R.: Rossom's Universal Robots", trans. by Paul Selver
http://www.mindfully.org/Reform/RUR-Capek-1920.htm
accessed 14 November 2012.

    B. Čapek, Karel, "R.U.R. (Rossum's Universal Robots", trans. by David Wyllie
http://ebooks.adelaide.edu.au/c/capek/karel/rur/
accessed 14 November 2012.

(b) On AI research into personality and emotion:

- Reid Simmons's Social Robots Project at Carnegie-Mellon University
http://www.cs.cmu.edu/afs/cs/project/robocomp/social/www/
accessed 14 November 2012.

- MIT's COG project on humanoid robotics
http://www.ai.mit.edu/projects/humanoid-robotics-group/cog/
accessed 14 November 2012.

- Bibliography   on   the   cognitive   science   of   emotion
http://www.cse.buffalo.edu/~rapaport/575/emotion.html
accessed 14 November 2012.

Chapter 21

Summary

Philosophical reflection...is not static, and fixed, but ongoing and dynamic. The conflict of opinions not only *isn't* something to worry about, in fact, it is precisely how things ought to be. ...

For...only after you've considered all sides will you be in a meaningful position to choose one—when that time comes to decide.

...the philosopher within me cannot make that decision for you. His job, he reminds me, is merely to rouse the philosopher within *you* and to get you thinking—not to tell you what to think.

That's *your* philosopher's job.

—Andrew Pessin, *The 60-Second Philosopher* (London: Oneworld): 3–4.

21.1   Recommended Readings:

1. Required:

   - Smith, Brian Cantwell (2002), "The Foundations of Computing", in Matthias Scheutz (ed.), *Computationalism: New Directions* (Cambridge, MA: MIT Press): 23–58.

       – Earlier version online at:
          http://www.ageofsignificance.org/people/bcsmith/print/smith-foundtns.pdf
          accessed 16 November 2012.

2. Very Strongly Recommended:

   - DeJohn, Jerry; & Dietrich, Eric (2003), "Editorial: Subvert the Dominant Paradigm! A Review of *Computationalism: New Directions*, edited by Matthias Scheutz", *Journal of Experimental & Theoretical Artificial Intelligence* 15(4) (October-December): 375–382, esp. §3.1, pp. 378–379.

   - Scheutz, Matthias (2002), "Computation, Philosophical Issues about", *Encyclopedia of Cognitive Science* (London: Macmillan): 604–610; online at:
     http://staffweb.worc.ac.uk/DrC/Courses%202006-7/Comp%204070/Reading%20Materials/scheutz%20Comp%2
     accessed 16 November 2012.

Let us take stock of where we have been. MAKE SURE THAT EVERYTHING HERE IS ACTUALLY DISCUSSED IN THE BOOK!

21.2 What Is Philosophy?

We began with a brief history of western philosophy in terms of its relevance to computer science. I offered a definition of philosophy as the personal search for truth, in any field, by rational means. We reviewed some of the principles of rationality, in particular, of argument analysis, looking at how an argument consists of premises and a conclusion, how the conclusion might follow validly from the premises if the argument is truth preserving, and how the argument might be sound if the premises are true. And we briefly looked at the main branches of philosophy, asserting the existence, for any field *X* of a "philosophy of *X*". So, the philosophy of computer science would be the study of the main goals and fundamental assumptions of computer science.

21.3 What Is Computer Science?

We then turned to an ontological question: What is computer science? We looked at both political and philosophical motivations for asking the question, and then we examined various answers: [Newell et al., 1967] said that computer science is the science of *computers* and surrounding phenomena such as algorithms. [Knuth, 1974b] said that computer science is the study of *algorithms* and surrounding phenomena such as the computers that they run on. One way of adjudication between these two apparently opposed viewpoints is to take them as being extensionally equivalent— computer science is the study of algorithms *and* the computers that execute them—but intensionally distinct because each focuses on a different portion of computer science's single subject matter.

A slightly different view is taken by [Newell and Simon, 1976], who said that computer science is the "artificial science" (as opposed to the "natural science") of the phenomena surrounding computers, where both kinds of science are empirical studies, with the latter studying phenomena that occur in nature and the former studying human-made artifacts. This can be contrasted with [Shapiro, 2001], who argues that computer science *is* a natural science, but not of computers; rather, it is the natural science of *procedures*.

Others, such as [Hartmanis and Lin, 1992], add to the subject matter the notion of *information*: Computer science is the study of how to represent and process information and of the machines and systems that do this. Still others, such as [Brooks, 1996], say that it is not a science at all, but a branch of engineering. And [Denning et al., 1989], [Denning and Freeman, 2009] say that it is a new kind of science (neither physical, biological, or social) of natural and artificial information processes.

21.4 Is Computer Science a Kind of Science or of Engineering?

So the question of what computer science is breaks down into two smaller questions: What does it study (as summarized in the previous section)? And how does it study it? That is, is it a science? Is it a branch of engineering? Or is it neither—is it some other, or new, kind of study? To answer this, we considered what science and engineering were.

21.4.1 What Is Science?

We looked at two dichotomies in trying to determine what science is: Is its goal to describe, or to explain? And are its descriptions or explanations intended to be about reality ("realism") or merely useful summaries that enable us to make predictions ("instrumentalism")?

And what is the "scientific method"? Is it the experimental method advocated by Bacon in the 1600s and described by Kemeny's "loop"? That is:

> **while** there is a new fact to observe, **do:**
> **begin**
> > observe it;
> > induce a general hypothesis (to explain or describe it);
> > deduce future observations;
> > verify your predictions
> **end.**

Or is it what Popper suggested in the 1950s: Science makes conjectures and refutations. A statement is scientific if and only if it is falsifiable.

Or is it what Kuhn suggested in the 1960s: Science proceeds by "paradigm revolutions" alternating with periods of "normal" science?

And what about subjects like mathematics, which seems to be scientific, yet is non-empirical?

21.4.2 What Is Engineering?

Perhaps we need to distinguish between pure and applied sciences; perhaps computer science is an applied science, or a branch of engineering. We looked at [Davis, 1998]'s history of engineering, which suggests that engineering is defined by its curriculum, which teaches how to apply science for the use and convenience of people and to improve the means of production. By contrast, [Petroski, 2003] suggests that the fundamental activity of engineering is *design*. But [Loui, 1987] suggests that computer "science" is a new kind of engineering that studies the theory, design, analysis, and implementation of information-processing algorithms.

21.5 What Does Computer Science Study?

So, whether it is a science, a branch of engineering, or something else, what does it study? Does it study two things: computing and computers? Or are these merely two aspects of a single, underlying subject matter? To answer this, we asked four questions:

21.5.1 What Is a Computer? Historical Answer

We saw that there were two intertwined branches of the history of computers, each of which had a slightly different goal: The goal of one branch was to build a computing machine. This is the history in which prominent roles were played by Pascal, Leibniz, Babbage, Aiken, Atansoff and Berry, Turing, Eckert and Mauchly, and von Neumann, among others. The goal of the other branch was to provide a foundation for mathematics. This is the history in which prominent roles were played by Leibniz (again), Boole, Frege, Russell, Hilbet, Turing (again), Church, and Gödel, among others.

21.5.2 What Is an Algorithm? Mathematical Answer

We began this investigation by asking what computation is. A function f is computable means by definition that there is an algorithm that computes f, that is, that there is an algorithm A such that, for all input i, $A(i) = f(i)$, and such that A specfies how f's input and output are related.

So, what is an algorithm? Roughly, an algorithm for a problem P is, by definition, a finite procedure (that is, a finite set of explicit instructions) for solving P that is (i) unambiguous for the computer or human who will execute it—that is, all steps of the procedure must be clear and well-defined for the executor—and (ii) it must eventually halt, outputting a correct solution to P. Of course, this is only a rough definition: Spelling out exactly what 'clear and well-defined' means was an accomplishment of the highest order.

We looked at the most successful solution to this problem of what an algorith is, namely, the Turing machine, as well as the history of the term 'computable'. We discussed Turing's thesis that a function is computable if and only if it is computable by a Turing machine, as well as Church's thesis that a function is computable if and only if it is lambda-definable. That both of these notions turn out to be logically equivalent to each other as well as to other analyses, such as that of recursive functions, allows us to talk of the Church-Turing Computability Thesis.

21.5.3 What Is a Computer? Philosophical Answer

Armed with the history of computers and the mathematics of computation, we returned to the question of what a computer is. We began our philosophical investigation with [Searle, 1990]'s claim that everything is (interpretable as) a digital computer. And we looked at some alternatives: that a computer is "magic paper" that can take as input patterns that describe changes to themselves and to other patterns, and that causes the described changes to occur [Hayes, 1997], that a computer is a device that changes

values of variables [Thomason, 2003] (see §21.7.2, (C4), below), and that the universe is a computer [Lloyd and Ng, 2004], [Wolfram, 2002].

21.5.4   What Is an Algorithm? Philosophical Answer

And we then returned to the question of what an algorithm is, now from a philosophical point of view. This divided into three questions:

21.5.4.1   What Is a Procedure?

We saw that [Shapiro, 2001] argued for the more general notion of a "procedure" as being the subject matter of computer science. So we looked at what some philosophers have said about procedures.

[Cleland, 1993] (and in subsequent papers) argued that "mundane" procedures (such as causal processes, including recipes) are effective procedures that are not computable by Turing machines, because their effectiveness depends on conditions in the external world. And [Preston, 2006] pointed out important differences between improvisational recipes (and music) and precise algorithms, suggesting that recipes are more like specifications of programs than they are like computer programs.

21.5.4.2   What Is Hypercomputation?

Next, we looked at the idea that there might be functions that can be computed in some more general sense than by Turing machines, what is called 'hypercomputation'. We looked at Turing's oracle machines, Boolos and Jeffrey's infinitely accelerating, Zeus machines, Wegner's interaction machines" and Putnam's and Gold's "trial and error" machines (which are TMs that can "change their mind": it is the last answer that matters, not the first one), which [Kugel, 2002] argued are necessary in order for AI to succeed.

21.5.4.3   What Is a Computer Program?

This led us to the third part of our investigation of algorithms: their implementation in computer programs. Here we went deep into the depths of the philosophy of computer science, looking at implementation, software, legal protections for software, and the relation of programs to scientific theories and to the real world.

21.5.4.3.1   What Is Implementation?   [Chalmers, 2011] (and elsewhere) argued against [Searle, 1990] that implementation is an isomorphism and that a computer is an implementation of a computation or of a universal Turing machine. And [Rapaport, 1999] argued that implementation is the semantic interpretation of an Abstraction in some medium. To understand this, we looked at the syntax and semantics of formal systems.

21.5.4.3.2 What Is Software? Paralleling the computer-computation distinction is the hardware-software distinction. According to [Moor, 1978], one can understand computers as physical objects or on a symbolic level (and we compared these two levels of understanding to [Dennett, 1971]'s three levels: the physical, design, and intentional "stances"). A computer program, for Moor, is a set of instructions that a computer can execute. But the notion of software is relative to both a computer and a person: X is software for computer C and person P if and only if X is a computer program for C that is changeable by P. Hardware is similarly relative: X is hardware for C and P if and only if X is (physically) part of C, and X is not software for C and P.

In contrast, [Suber, 1988] argued that software is simply syntactic form, and [Colburn, 1999] argued that it is a "concrete abstraction" that has an abstract "medium of description" (a text in a formal language) and a concrete "medium of execution" (circuits and semiconductors).

We also looked at how the software-hardware distinction compared to Spinoza's "dual-aspect" theory concerning the mind-brain distinction, and whether any of the interpretations of Spinoza's theory might apply to the computer case.

21.5.4.3.3 Can (Should) Software Be Patented, or Copyrighted? In order to try to understand the software-hardware relationship, we looked at the issue of whether software could, or should, be patented or else copyrighted. After all, if a program is a piece of text, then copyright is appropriate. But if a program is a piece of hardware, then patent is appropriate. But programs seem to be both, yet nothing can (legally) be both patented and copyrighted.

To resolve this paradox, [Newell, 1986] suggested that we need to devise good models ("ontologies") of algorithms and other computational entities. And [Koepsell, 2000] suggested that we need to revise the models of legal protection.

21.5.4.3.4 Are Programs Scientific Theories? As part of our endeavor to understand what a program is, we considered the claim made by several philosophers and computer scientists that some programs are scientific theories.

Programs are (a language for expressing) theories, which can then be their own models. We looked at the differences and relations between theories and models, simulations and "the real thing", and simulation vs. emulation. We also briefly looked at philosophical theories of scientific explanation and of scientific models.

21.5.4.3.5 What Is the Relation of Programs to the World? A final question related to what computer programs are is how they relate to the real world. Here, we looked at [Smith, 1985], who argued that there is a gap between the world and our models of it, and that computers rely on models of the world, but must act in the real world.

One related issue here concerns [Cleland, 1993]'s views about the Church-Turing Computability Thesis: Could the computability of a problem depend in part on the real world, and not exclusively on the program for solving it?

Another related issue concerns whether Smith's observations pertain only to computers or also to us humans. After all, we must act in the real world, but perhaps

our actions are also based on mental models of the world that are separate from the real world.

21.6   Computer Ethics

We looked at two topics in computer ethics.

21.6.1   Are There Decisions Computers Should Never Make?

Given the limitations on the verifiability of program correctness, it becomes important to ask this question, first asked in [Moor, 1979].

His answer has two parts: First, no, there are no decisions that computers should never make *as long as their track record is better than that of humans*. After all, the question seems to be logically equivalent to this one: Are there decisions that should not be made rationally? Presumably, computer programs that make decisions make them on the basis of rational evaluation of the facts. And surely we always want to make our decisions rationally.

Second, it is up to us to accept or reject the decisions made by a computer. This is the case no more and no less than it is for decisions made (for us) by others. Thus, the decision made by a computer should be evaluated in the same way as advice offered by an expert or in a textbook.

On the other hand, [Friedman and Kahn, 1997] argue that, yes, there *are* decisions that computers should never make, on the grounds that only humans are capable of being moral agents. Whether that is really the case is part of the second issue in computer ethics (see §21.6.2). But to err is human, as shown in the case of the airline crash caused by following a *human's* decision instead of a computer's. This must be contrasted with cases in which tragedy occurred by blindly following a *computer's* decision, as in the radiation scenario.

21.6.2   Should We Build an Artificial Intelligence?

The second issue in computer ethics that we looked at was, given the fact that it is plausible that we *could* build an AI, it becomes important to ask whether we *should* build one.

[Lem, 1971] pointed out that, if we do succeed in building AIs, we may someday have to pull the plug on them. And [LaChat, 1986] suggested that maybe we shouldn't even begin. But [LaChat, 1986] also argued that considering the possibility of building one and considering the moral consequences of doing so enables us to deal with important issues such as: What is a person? Would an AI with personhood have (moral) rights? (See §21.6.1, above.) And could the AI itself be a moral agent?

21.6.3   Philosophy of AI

Our final topic concerned the philosophy of artificial intelligence, in which we looked at the Turing test and the Chinese Room Argument.

[Turing, 1950]'s article that introduced what is now called the Turing test suggested that a computer will be *said* to be able to think if we cannot distinguish its cognitive behavior from that of a human.

[Searle, 1980]'s rebuttal argued that a computer could pass a Turing test without really being able to think.

We then looked at how "multiple implementation" and how syntax can suffice for semantic interpretation of the kind needed for computational cognition might overthrow Searle's position.

21.7 What Is the Philosophy of Computer Science?

Let's now return to our original question and examine it in the light of an essay directed to that issue by Brian Cantwell Smith, [Smith, 2002].[1]

21.7.1 The Major Question

The major question in the philosophy of computer science, according to Smith, is: What is computation? And the answer is that it is a lot more than just the theory of Turing machines.

Smith says that computing is:

empirical: "Computation in the wild" includes every aspect of the nature and use of computing and computers, presumably including things like iPads and Facebook.

conceptual: Computing and computers are intimately related to issues in semantics such as interpretation and representation. This is probably where the theory of Turing machines fits in.

cognitive: The theory of computation must provide a foundastion for a theory of mind (that is, for what I have called computational cognitive science). This is probably where the Turing test and the Chinese Room Argument fit in.

21.7.2 9 Theses about Computation

Smith offers 9 "theses" about computation, each of which is worthy of a book-length discussion. Here, I will present them, not in the order in which Smith does, but in what seems to me to be a more logical order (but I retain Smith's numbering scheme).

(C2) *A theory of computation needs a full theory of semantics and intentionality.*

By 'intentionality', I think Smith means what the 19th-century cognitive scientist Franz Brentano meant, namely, "directedness to an object", that is, "being about something".[2]

---

[1]For a critique of [Smith, 2002], see [DeJohn and Dietrich, 2003], esp. §3.1, pp. 378–379.

[2]On intentionality, see [Jacob, 2010]. On Brentano, see [Huemer, 2010]. On "directedness" and "aboutness", see [Rapaport, 2012a].

(C1) *However, <u>none</u> of the following provide such a theory:*

1. **effective computability:** the study of what can, can't, or is impractical to be done by an abstract machine

2. **execution of algorithms, or rule following**—including recipes

3. **calculation of functions:** the behavior of taking input and producing output

4. **digital state machines:** that is, automata with finite sets of states

5. **information processing:** that is, storing, manipulating, and displaying "information".[3]

6. **physical symbol systems:** that is, the claim that a computer's interaction with symbols depends on its physical implementation

7. **formal symbol manipulation** by a machine without regard to meaning:

   (C3) • *Formal symbol manipulation separates the syntactic and semantic domains.*

      • *But real-world computational processes are "participatory"*

      That is, the syntax and semantics of real-world computational processes intersect in two ways:

      (a) As argued in [Smith, 1985], computers get their input from, and must act in (that is, produce output to), the real world.

      (b) Computers are part of the real world:

         i. Some computer-*internal* symbols refer to other computer-internal symbols; others refer to external objects.

         ii. Some computer-*external* symbols refer to computer-*internal* ones; others refer to other external ones.

   (C6) *Computation is not formal in any of the following senses of 'formal':*

      • anti-semantic: the sense in which syntax is independent of semantics

      • syntactic: for example, proof theory only relies on pattern matching and symbol manipulation.

      • determinate or well-defined (that is, not vague): for example, digital or discrete

      • mathematical: I think Smith has in mind here the kind of formality that Bertrand Russell was discussing in this remark:

         "Pure mathematics consists entirely of such assertions as that, if such and such a proposition is true of *anything*, then such and such another proposition is true of that thing. It is essential not to discuss whether the first proposition is really true, and not to mention what the anything is, of which it is supposed to be true. ... *If* our hypothesis is about *anything*,

---

[3]On various meanings of 'information', see [Piccinini and Scarantino, 2011] and http://en.wikipedia.org/wiki/Information

> and not about some one or more particular things, then our deductions constitute mathematics. Thus **mathematics may be defined as the subject in which we never know what we are talking about, nor whether what we are saying is true.** [Russell, 1917, Ch. 4, p. 75; my boldface][4]

- analytic method: This might be related to [Simon, 1962]'s theory of how to analyze complexity "hierarchically", that is, recursively.

(C4) *Turing machines don't use marks to represent numbers; they use numbers to represent marks!*

- Recall that syntax concerns mark-manipulation, as we saw in CITE. (This is the kind of mathematics that Russell was discussing in the above quote.) The real world of mathematics concerns number-manipulation. And semantics is the study of the relation between them.

- But, according to Smith,[5] "In science, we use numbers to represent 'stuff'; computation only studies marks".

A corollary is this: *The theory of computation is "a theory [that is, a language] of how... patches of the world in one physical configuration [can] change into another physical configuration"* [Smith, 2002, 42]. (Compare this to Thomason's definition of a computer, §21.5.3, above.) That is:

(C5) *Computability theory is a mathematical theory of the flow of causality.*

(C7) *A theory of computation needs a full theory of ontology.* (For a beginning, see [Smith, 1996]. For critical reviews, see [Koepsell, 1998], [Loui, 1998].)

(C8)
- *Computers are not natural kinds.* (But recall that [Shapiro, 2001] argued that *procedures* are natural kinds.)
- *There can be no theory of computation.*

(C9) *Therefore, the existence of computation is extremely important, because any theory of it will be a theory of intentional artifacts, hence a theory of everything!*

And that seems to be a good note on which to end.

---

[4]The passage humorously concludes: "People who have been puzzled by the beginnings of mathematics will, I hope, find comfort in this definition, and will probably agree that it is accurate."

[5]In a lecture given at UB in 20xx??.

21.8   NOTE FOR NEXT DRAFT

1. See discussion of "real" computers, which are interaction machines, with "theoretical" computers, in §20, above.

Chapter 22

Syntax and Semantics

USE http://www.cse.buffalo.edu/∼rapaport/563S05/synsem.html,
    BUT ALSO USE Chs. 2 and 3 from
http://www.cse.buffalo.edu/∼rapaport/Papers/book.pdf
and their published version:

- Rapaport, William J. (1995), "Understanding Understanding: Syntactic Semantics and Computational Cognition", in James E. Tomberlin (ed.), *AI, Connectionism, and Philosophical Psychology*, *Philosophical Perspectives* Vol. 9 (Atascadero, CA: Ridgeview): 49-88; reprinted in Toribio, Josefa, & Clark, Andy (eds.) (1998), *Language and Meaning in Cognitive Science: Cognitive Issues and Semantic Theory*, *Artificial Intelligence and Cognitive Science*, Vol. 4: Conceptual Issues (New York: Garland): 73–88.

And Ch. 3 of Understanding Understanding,
http://www.cse.buffalo.edu/∼rapaport/Papers/book.pdf
    And also use [Suber, 1997a].

This Is Not the End

Lots of things never end. Space. Time. Numbers. The questions little kids ask.[1]

And philosophy.

You try to convince somebody of something—even yourself—by offering reasons to believe the thing. But then your belief is only as valid as your reasons are, so you offer reasons to accept your reasons. But then those reasons need *further* reasons and you're off. As a result it often seems that there aren't any answers to philosophical questions: there are just more arguments, more objections, more replies. And so it may easily seem that it's not worth even getting started. Why bother? You'll never finish. You may as well try to count all the numbers.

But there is another way of thinking about it.

I went snorkeling for the first time a few years ago. It was an amazing experience. There was a whole world under that water to which I'd been oblivious my entire life. This world was populated with countless amazing creatures with all sorts of complex relationships to each other in that tangled ecosystemic way. Indeed every single thing was connected to every other thing: this one is food for that one, which excretes chemicals used by another one, which excretes waste products used by others, and so on. Stunning, fascinating, and absolutely, deeply, beautiful. It had been there all along, jsut waiting for me to dive in.

If you were now to tell me that that ocean goes on forever, filled with ever more amazing creatures in more amazing relationships—I wouldn't say, "Well then why bother entering?" Rather, I'd say, "Where can a guy get a wetsuit around here?"

But that is philosophy. It's filled with countless amazing ideas, concepts, beings, which exist in all sorts of complex logical relationships with each other. And unlike the actual ocean this one *is* infinitely deep: Wherever you enter you can keep going, and going, and going. What you should be thinking, then, is not: "Why enter?" It is, rather, this: thank you—very much.

But of course, that world just is *this* world, the world that you're in. This great ocean you may be looking for, you're alrady in it. You just have to start thinking about it. The very first drop in that bucket is a splash into the infinite.

This is the beginning.

—Andrew Pessin, *The 60-Second Philosopher* (London: Oneworld): 124–125.

---

[1] The cells on a Turing-machine tape. Infinite loops.—items added by WJR.

Appendix A

Argument Analysis and Evaluation

[Sidney Harris cartoon showing boy sitting in front of a computer, thinking:]

Liebniz, Boole and Gödel worked with logic.
I work with logic.
I am Leibniz, Boole and Gödel.
— *American Scientist* 73 (January-February): 19.

A.1 Recommended Readings:

- Required:

    - Chudnoff, Elijah (2007), "A Guide to Philosophical Writing", (Cambridge, MA: Harvard Center for Expository Writing).

        * Accessed 21 March 2014 from:
          http://www.fas.harvard.edu/∼phildept/files/GuidetoPhilosophicalWriting.pdf

A.2 Introduction

In §2.3, I said that the methodology of philosophy involved "rational means" for seeking truth, and in §2.6, we looked at different kinds of rational methods. In this appendix, we'll look more closely at one of those methods: argument analysis and evaluation.

Unless you are gullible, willing to believe everything you read or anything that an authority figure tells you, you should want to know *why* you should believe something that you read or something that you are told. If someone tells you that some proposition C is true because some other propositions R_1 and R_2 are true, you might then consider, first, whether those *reasons* (R_1 and R_2) really do support the *conclusion* C and, second, whether you believe the reasons.

So, let's consider how you might go about doing this.

A.3 A Question-Answer Game

Consider two players, Q and A, in a question-answer game:

- Q asks a question (see step A.3, below).

- A gives an alleged answer (see step A.3, below).

- Q must verify A's alleged answer (see steps A.3–A.3, below).

Step 1 Q asks whether C is true.

Step 2 A answers: "C, because R_1 and R_2."

> That is, A gives an *argument* for conclusion C with reasons (also called '*premises*') R_1 and R_2.
>
> Note, by the way, that this use of the word 'argument' has nothing directly to do with the kind of fighting argument that you might have with your roommate; rather, it's more like the *legal* arguments that lawyers present to a jury.
>
> Also, for the sake of simplicity, I'm assuming that A gives only two reasons for believing C; in a real case, there might be only one reason (for example: Fred is a computer scientist; therefore; someone is a computer scientist), or there might be more than two reasons (for examples, see any of the arguments for analysis and evaluation in Appendix B).

Step 3 Q must *analyze* or "verify" A's arguments. Q can do this by asking three questions:

- Do I believe (that is, do I agree with) R_1?
- Do I believe (that is, do I agree with) R_2?
- Does C follow *validly* from R_1 and R_2?

There are a few comments to make about these steps. First, steps A.3 and A.3 are *recursive*: For each of them, Q could ask A (or someone else!) whether R_i is true, and A (or the other person) could give an argument for conclusion R_i with new premises R_3 and R_4. Clearly, this process could continue (this is what toddlers do when they continually ask their parents "Why?"!). It is an interesting philosophical question, but fortunately beyond our present scope, to consider where, if at all, this process might stop.

Second, to ask whether C follows "validly" from the premises is to assume that A's argument is a *deductive* one. For the sake of simplicity, all (or at least most) of the arguments in Appendix B are deductive, but, in real life, most arguments are not completely deductive, or not even deductive at all. So, more generally, in Step A.3, Q should ask whether C follows *rationally* from the premises: If it does not follow deductively, does it follow inductively? Abductively? And so on.

Third, unlike the first steps of considering the truth value of the premises, this step of determining whether the relation between the premises of an argument and its conclusion is a rational one is *not* similarly recursive, on pain of infinite regress. The classical source of this observation is due to Lewis Carroll (of "Alice in Wonderland" fame).[1] Carroll was a logician by profession, and wrote a classic philosophy essay involving Achilles and the Tortoise [Carroll, 1895].

Finally, it should be pointed out that the order of doing these steps is irrelevant. Q could first analyze the validity (or rationality) of the argument and then analyze the truth value of (that is, decide whether to agree with) the premises, rather than the other way round.

Step 4 Having *analyzed* A's argument, Q now has to *evaluate* it, by reasoning in one of the following ways;

- **If** I agree with R_1,
  and if I agree with R_2,
  and if C follows validly (or rationally) from R_1 and R_2,
  then I logically must agree with C.

- But, bf if I really *don't* agree with C,
  then I must reconsider my agreement **either:**
  with R_1,
  or with R_2,
  or with the logic of the inference from $R_1 \& R_2$ to C.

- **If** I *agree* with R_1,
  and if I agree with R_2,
  but the argument is *invalid*, is there a *missing premise*—an extra reason (see below)—that would validate the argument **and** that I would agree with?

---

[1] More properly known as *Alice's Adventures in Wonderland* and *Through the Looking Glass*.

[(i)]

  – If so, then I can accept *C*,
     else I should not yet reject *C*,
     but I do need a new argument for (that is, a new set of reasons for believing) *C*.

- **If** I *disagree* with R_1 or with R_2 (or both),
  bf then *this* argument is not a reason *for me* to believe *C*;
  so, I need a new argument for *C*.

     There is one other option for *Q* in this case: *Q* might want to go back and reconsider the premises. Maybe *Q* was too hasty in rejecting them.

- What if *Q* cannot find a good argument for believing *C*? Then it might be time to consider whether *C* is false. In that case, *Q* needs to find an argument for *C*'s negation: Not-*C* (sometime symbolized '¬*C*').

This process of argument analysis and evaluation is summarized in the flowchart in Figure A.1.

A.4 Missing Premises

One of the trickiest parts of argument analysis can be identifying missing premises. Often, this is tricky because the missing premise seems so "obvious" that you're not even aware that it's missing. But, equally often, it's the missing premise that can make or break an argument.

Here's an example from the "Textual Entailment Challenge", a competition for researchers interested in knowledge representation and in information extraction.[2] In a typical challenge, a system is given one or two premises and a conclusion (to use our terminology) and asked to determine whether the conclusion follows from the premise. And "follows" is taken fairly liberally to include all kinds of non-deductive inference.

Here is an example:

Premise 1 (*P*): Bountiful arrived after war's end, sailing into San Francisco Bay 21 August 1945.

Premise 2: Bountiful was then assigned as hospital ship at Yokosuka, Japan, departing San Francisco 1 November 1945.

Conclusion (*C*): Bountiful reached San Francisco in August 1945.

The idea is that the two premises might be sentences from a news article, and the conclusion is something that a typical reader of the article might be expected to understand from reading it.

I hope you can agree that this conclusion does, indeed, follow from these premises. In fact, it follows from Premise 1 alone. Premise 2 is a "distractor", in this case.

But what rule of inference allows us to infer *C* from *P*?

---

[2]For more information on "extual entailment" in general, and the Challenge in particular, link to: http://pascallin.ecs.soton.ac.uk/Challenges/RTE3/ (accessed accessed 7 January 2013).

- *P* talks of "arrival" and "sailing into", but *C* talks only of "reaching"

- *P* talks of "San Franciso Bay", but *C* talks only of "San Francisco"

There are no logical rules that connect these concepts.

Most people, I suspect, would think that no such rules would be needed; after all, isn't it "obvious" that, if you arrive somewhere, then you have reached it? And isn't it "obvious" that San Francisco Bay must be in San Francisco?

Well, maybe. But, whereas *people* might know these things, computers won't, unless we tell them. In other words, computers need some lexical knowledge and some simple geographical knowledge. (If you don't like the word 'knowledge' here, you can substitute 'information'.)

So, we need to supply some extra premises that link *P* with *C* more closely. These are the "missing premises". And the argument from *P* to *C* is called an 'enthymeme' (because the missing premises are "in" (Greek 'en-') the arguer's "mind" (Greek 'thymos')).

So, we might elaborate the argument as follows (there are other ways to do it; this is one that comes to my mind):

P: Bountiful arrived after war's end, sailing into San Francisco Bay 21 August 1945.

P_a: If something sails into a place, then it arrives at that place.

C_1: ∴ Bountiful arrived at San Francisco Bay 21 August 1945.

In this first step, I've added a missing premise, P_a, and derived an intermediate conclusion C_1. Hopefully, you agree that C_1 follows validly (or at least logically in some way, that is, rationally) from *P* and P_a.

We have no way of knowing whether *P* is true, and must, for the sake of the argument, simply assume that it is true. (Well, we could look it up, I suppose; but we're not asked if the argument is "sound" (see below), only if it is "valid": Does *C* follow from *P*?)

P_a, on the other hand, doesn't have to be accepted at all; after all, *we* are imposing it on the (unknown) author of the argument. So, we had better impose something that is likely to be true. P_a is offered as part of the meaning of "sail into". I won't defend its truth any further here, but if you think that it's *not* true, then you should either reject the argument or else find a better missing premise.

We might have chosen a slightly different missing premise:

P_b: If something arrives in a place named '*X* Bay', then it arrives at a place named '*X*'.

C_2: ∴ Bountiful arrived at San Francisco 21 August 1945.

C_2 will follow from C_1 and P_b, but is P_b true? Can you think of any bays named '*X* Bay' that are not located in a place named '*X*'? If you can, then we can't use P_b. Let's assume the worst: Then we'll need something like:

P_b.**1**: If something arrives in San Francisco Bay, then it arrives at San Francisco.

C_2 will follow from C_1 and P_b.1, and we can easily check the likely truth of P_b.1 by looking at a map.

So far, so good. We've now got Bountiful arriving at San Francisco on 21 August 1945. But what we need is Bountiful "*reaching*" San Francisco in August 1945. So let's add:

P_c**:** If something arrives somewhere, then it reaches that place.

Again, this is proposed as an explication of part of the meaning of 'arrive', and, in particular, of that part of its meaning that connects it to C.

From P_c and C_2, we can infer:

C_3**:** Bountiful reached San Francisco 21 August 1945.

Are we done? Does $C_3 = C$? Look at them:

C_3**:** Bountiful reached San Francisco 21 August 1945.

C**:** Bountiful reached San Francisco in August 1945.

Think like a computer! What do you need to know in order to know whether $C_3 = C$?
You need to know whether the final missing premise, P_d, is true:

P_d**:** If something occurs (on) DATE MONTH YEAR, then it occurs *in* MONTH YEAR.

And that's true by virtue of the way (some) people talk. From P_d and C_3, we can infer C.

So, the simple argument that we started with, ignoring its irrelevant premise, becomes this rather more elaborate one:

P**:** Bountiful arrived after war's end, sailing into San Francisco Bay 21 August 1945.

P_a**:** If something sails into a place, then it arrives at that place.

C_1**:** ∴ Bountiful arrived at San Francisco Bay 21 August 1945.

P_b**:** If something arrives in a place named 'X Bay', then it arrives at a place named 'X'.

(or P_b.1: If something arrives in San Francisco Bay, then it arrives at San Francisco.)

C_2**:** ∴ Bountiful arrived at San Francisco 21 August 1945.

P_c**:** If something arrives somewhere, then it reaches that place.

C_3**:** ∴ Bountiful reached San Francisco 21 August 1945.

P_d**:** If something occurs (on) DATE MONTH YEAR, then it occurs *in* MONTH YEAR.

C**:** ∴ Bountiful reached San Francisco in August 1945.

A.5   When Is an Argument a "Good" Argument?

As we have seen, *Q* needs to do two things in analyzing and evaluating an argument:

1. deciding whether the premises are true (that is, deciding whether to agree with, or believe, the premises), and

2. deciding whether the inference (that is, the reasoning) from the premises to the conclusion is a good one.

That is, there are two separate conditions for the "goodness" of an argument:

1. factual goodness

2. logical goodness

Factual goodness—truth—is beyond the scope of logic, although it is definitely not beyond the scope of deciding whether to accept the conclusion of an argument. As we saw in §2.4, there are several ways of defining 'truth' and of determining whether a proposition is true. Two of the most obvious (though not the simplest to apply!) are (1) constructing a (good!) argument for a proposition whose truth value is in question and (2) making an empirical investigation to determine its truth-value (for instance, performing some scientific experiments or doing some kind of scholarly research).

Logical goodness (for deductive arguments) is called 'validity'. I will define this in a moment. But, for now, note that these two conditions must both obtain for an argument to be "really good": A "really good" argument is said to be *sound*—an argument is sound if and only if it is both valid and "factually good", that is, if and only if it is both valid and all of its premises really are true.

Just to drive this point home: If the premises of an argument are all true (or if you believe all of them), that by itself does not make the argument sound ("really good"). For one thing, your belief in the truth of the premises might be mistaken. But, more importantly, the argument might not be valid.

And, if an argument is valid, that by itself does not make the argument sound ("really good"). All of its premises need to be true, also; that is, it needs to be factually good.

So, what does it mean for a (deductive) argument to be valid? An argument **is valid** $=_d ef$ it is "truth-preserving". That is, an argument is valid if and only if, whenever all of its premises are true, then its conclusion must also be true. Here's another way to put it: An argument is valid if and only if it is impossible for all of its premises to be simultaneously true, yet its conclusion is false. Note that none of these formulations say that the premises *are* true. They only say that, *if* the premises *were to be* true, then the conclusion would *preserve that truth*—it would "inherit" that truth from the premises—and so it would also (have to) be true.

That's fine as far as it goes, but it really isn't very helpful in deciding whether an argument really is valid. How can you tell if an argument is truth-preserving? There is a simple, recursive definition, but, to state it, we'll need to be a bit more precise in how we define an argument.

Definition 1: An **argument from premises** $P_1, \ldots, P_n$ **to conclusion** C is$_{def}$ a *sequence* of propositions $\langle P_1, \ldots, P_n, C \rangle$, where C is *alleged* to follow logically from the P_i.

Definition 2: An argument **is valid** (or: **is truth-preserving**) if and only if it is an argument in which each proposition P_i or conclusion C either:

- is *a tautology*

- or is *a premise*

- or follows validly from previous propositions in the sequence by one or more "rules of inference".

This needs some commenting! First, what is a "tautology"? A **tautology** is a proposition that *must always be true*. How can that be? Most tautologies are (uninformative) "logical truths", such as 'Either P or not-P', or 'If P, then P'. (Note that, if P is true (or, if you believe P), then 'Either P or not-P' has to be true (or, you are logically obligated to also believe 'Either P or not-P'), and, if P is false, then not-P is true, and so 'Either P or not-P' still has to be true (or, you are logically obligated to also believe 'Either P or not-P'); so, in either case, the disjunction has to be true (or, you are logically obligated to believe it. Similar considerations hold for 'If P, then P'.) Sometimes, statements of mathematics are also considered to be tautologies (whether they are "informative" or not is an interesting philosophical puzzle; see [Wittgenstein, 1921]).

Second, a premise is one of the initial reasons given for C, or one of the missing premises added later. Premises, of course, need not be true, but, when evaluating an argument for validity, we must *assume* that they are true "for the sake of the argument".

Third, clause A.5 might look circular, but it isn't; rather, it's recursive. In fact, this entire definition is recursive. The base cases of the recursion are the first two clauses: Tautologies must be true, and premises are assumed to be true. The recursive case consists of "rules of inference", which are "sub-proofs" that are clearly valid (truth-preserving) when analyzed by means of truth tables.

So, what are these "primitive" valid arguments known as 'rules of inference'? The most famous is called 'Modus Ponens':

> From P
> and 'If P, then C',
> you may validly infer C.

Why may you validly infer C? Because the truth table for 'If P, then C' says that that conditional proposition is false in only one circumstance: when its "antecedent" (P) is true and its "consequent" (C) is false; in all other circumstances, the conditional proposition is true. So, if the antecedent of a conditional is true, and the conditional itself is true, then its consequent must also be true. Modus Ponens preserves truth.

Another important rule of inference is called 'Universal Elimination' (or 'Universal Instantiation'):

From 'For all x, $F(x)$',
you may infer $F(a)$, for any individual a in the domain of discourse (that
is, in the set of things that you are talking about).

A truth-table analysis won't help here, because this is a rule of inference from "first-order predicate logic", not from "propositional logic", and the formal definition of truth for first-order predicate logic is beyond our scope, but it should be pretty obvious that, if it is true that everything in the domain of discourse has some property F, then it must also be true that any particular thing in the domain (say, a) has that property. (For more rules of inference and for the formal definition of truth in first-order predicate logic, see any good introductory logic text.)

NEED   A   SECTION   WITH   EXAMPLES   OF
VALID/SOUND/INVALID/UNSOUND ARGS.

A.6   Summary

So, to analyze an argument, you must identify its premises and conclusion, and supply any missing premises to help make it valid. To evaluate the argument, you should then determine whether it is valid (that is, truth preserving), and then decide whether you agree with its premises.

If you agree with the premises of a valid argument, then you are logically obligated to believe its conclusion. If you don't believe it's conclusion, even after your analysis and evaluation, then you need to revisit both your evaluation of its validity (maybe you erred in determining its validity) as well as your agreement with its premises: If you really *disagree* with the conclusion of a valid argument, then you must (logically) *disagree* with at least one of its premises.

You should try your hand at analyzing and evaluating the arguments in Appendix B!

A.7   NOTES FOR NEXT DRAFT

1. MISCELLANEOUS THOUGHTS ABOUT PROOFS

    (a) There are lots of ways to have *invalid* arguments!

    (b) More importantly, it is possible to have an *invalid* argument whose conclusion is *true*. Here's an example:

    ```
    All birds fly.            (true)
    Tweety the canary flies.  (true)
    ------------------------
    Therefore, Tweety is a bird. (true)
    ```

    This is invalid, despite the fact that both premises and the conclusion are true: It is invalid, because the argument form can have true premises and a

false conclusion:

$$\forall x[P(x) \rightarrow Q(x)$$
$$R(a) \wedge Q(a)$$
$$\therefore P(a)$$

Here's a counterexample, that is, an argument with this form that has true premises but a false conclusion:

```
All birds fly.
Bob the bat flies.
------------------
Therefore, Bob is a bird.
```

Just having a true conclusion doesn't make an argument valid. And such an argument doesn't prove its conclusion (even though the conclusion is true).

(c) Can *any* proposition (or its negation) be proved?

That is, given a proposition P, we know that either P is true or else P is false (that is, that $\neg P$ is true). So, whichever one is true should be provable. Is it?

No!

First, there are propositions whose truth value we don't know *yet*. For example, once upon a time, no one knew if Fermat's Last Theorem could be proved. That's the proposition that the equation

$$x^n + y^n = z^n$$

has no solutions for integers x, y, z and integer $n > 2$. But it has now been proved.[3]

For another example, no one knows (yet) if Goldbach's Conjecture is true. Goldbach's Conjecture says that all positive even integers are the sum of 2 primes; for example, $28 = 5 + 23$. Expressed in first-order logic, this would be:

$$\forall x[\text{Integer}(x) \wedge (x > 0) \wedge \text{Even}(x) \rightarrow \exists y \exists z[\text{Prime}(y) \wedge \text{Prime}(z) \wedge (x = y + z)]]$$

For yet another example, no one knows (yet) if the Twin Prime Conjecture is true. The Twin Prime Conjecture says that there are an infinite number of "twin" primes, that is, primes m, n such that $n = m + 2$; for example, 2 and 3, 3 and 5, 5 and 7, 9 and 11, 11 and 13, etc. Expressed in first-order logic, this would be:

$$\forall m \forall n[\text{Prime}(m) \wedge \text{Prime}(n) \wedge (n = m + 2) \rightarrow \exists x \exists y[\text{Prime}(x) \wedge \text{Prime}(y) \wedge (y = x + 2) \wedge (y > m) \wedge (x > n)]]$$

Second—and much more astounding than our mere inability so far to prove or disprove any of these conjectures—there are propositions whose truth

---

[3]See http://en.wikipedia.org/wiki/Fermat's_Last_Theorem

value is known to be true, but which we can prove that we cannot prove! This is the essence of Gödel's Incompleteness Theorem. Stated informally, it asks us to consider this proposition, which is a slight variation on the Liar Paradox (that is the proposition "This proposition is false": If it's false, then it's true; if it's true then it's false):

(G) This proposition is true but unprovable.

We can assume that *G* is either true or false. Is it false? If it is false, then it is provable. But any provable proposition has to be true (because proofs are truth-preserving). So it isn't false. Therefore, it must be true. But if it's true, then it's unprovable. End of story; no paradox!

That is, there are true propositions (moreover, Gödel showed that they are propositions that are true in the mathematical system consisting of first-order predicate logic plus Peano's axioms; that is, they are true propositions of arithmetic!) that cannot be proved. (For more information on Gödel's proof, see [Nagel et al., 2001], [Hofstadter, 1979], and [Franzén, 2005].)

2. Application of Predicate Logic to AI
   Here's an application of predicate logic to artificial intelligence (AI).

   In the late 1950s, one of the founders of AI, John McCarthy, proposed a computer program to be called "The Advice Taker", as part of a project that he called "programs with common sense". (McCarthy is famous for at least the following things: He came up with the name 'artificial intelligence', he invented the programming language Lisp, and he helped develop time sharing. For more information on him, see:
   http://en.wikipedia.org/wiki/John_McCarthy_(computer_scientist)
   and
   http://aitopics.org/search/site/John%20McCarthy)

   The idea behind The Advice Taker was that problems to be solved would be expressed in a predicate-logic language (only a little bit more expressive than first-order logic), a set of premises or assumptions describing required background information would be given, and then the problem would be solved by logically deducing it from the assumptions.

   He gave an example: getting from his desk at home to the airport. It begins with premises like

   at(I,desk)

   meaning "I am at my desk", and rules like

   $$\forall x \forall y \forall z [\text{at}(x,y) \land \text{at}(y,z) \rightarrow \text{at}(x,z)],$$

   which expresses the transitivity of the "at" predicate, and slightly more complicated rules (which go slightly beyond the expressive power of FOPL) like:

   $\forall x \forall y \forall z [\text{walkable}(x) \land \text{at}(y,x) \land \text{at}(z,x) \land \text{at}(I,y) \rightarrow \text{can}(I, \text{go}(y,z,\text{walking}))];$

that is, if x is walkable, and if y and z are at x, and if I am at y, then I can go from y to z by walking.

The proposition to be proved from these (plus lots of others) is:

    want(at(I,airport))

To see it all worked out, take a look at [McCarthy, 1959].
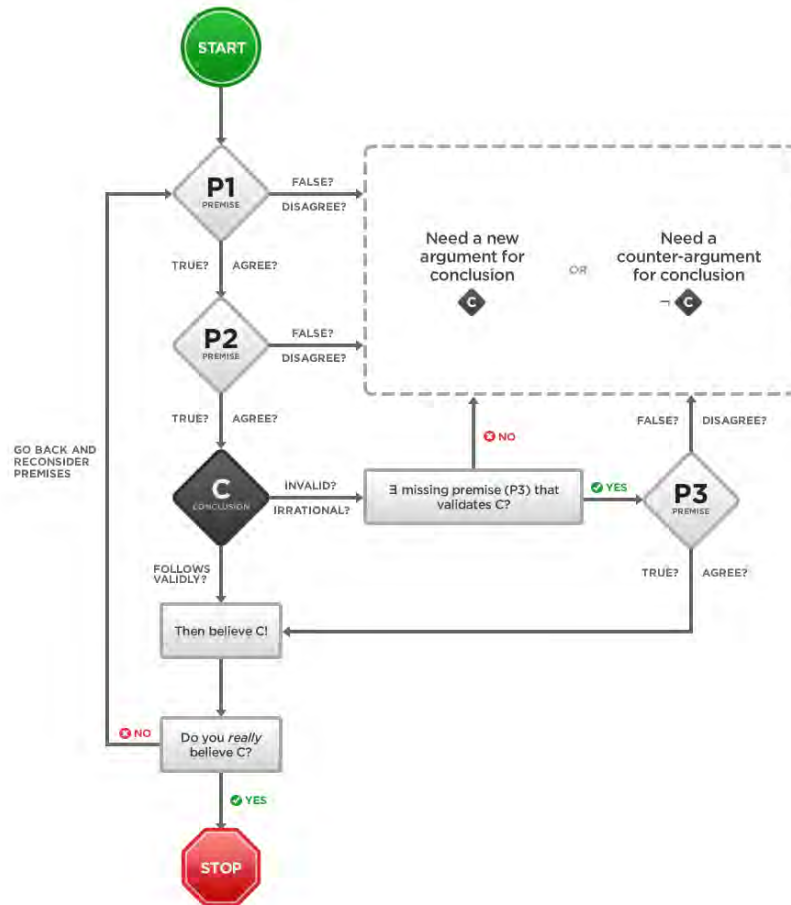
Figure A.1: This flowchart uses '*Pi*' instead of R_i, as the text does. Also, the symbol '*exists*' should be read: "Does there exist".

Appendix B

Position-Paper Assignments

B.1   Recommended Readings:

- Required:
    - Chudnoff, Elijah (2007), "A Guide to Philosophical Writing", (Cambridge, MA: Harvard Center for Expository Writing).
        * Accessed 21 March 2014 from:
          http://www.fas.harvard.edu/~phildept/files/GuidetoPhilosophicalWriting.pdf

B.2 Introduction

One of the best ways to learn how to do philosophy and, perhaps more importantly, to find out what your beliefs are about important issues (as well as what your *reasons for your beliefs* are!) is to write about them and then to discuss what you've written with others who have also thought and written about about the issues—your "peers".

So, the writing assignments take the form of "position papers", in which you will be:

- presented with a logical argument about a topic in the philosophy of computer science,

- asked to analyze and evaluate the argument,

- given an opportunity to clarify and defend your analysis and evaluation,

- and simultaneously be asked to help your peers clarify and defend *their* analyses and evaluations of the same argument in an exercise called "peer editing".

This should help you to clarify your *own* position on the topic.

The arguments that are presented in the following sections are those that I have used when I have taught this course. Instructors are invited to modify these or to create their own arguments. The assignments can be scheduled at the convenience of the instructor, which is why I have collected them in this appendix rather than placing them throughout the text. In general, however, I suggest putting each assignment approximately one week *after* the relevant topic has been covered in class. This way, the students will have the benefit of having thought about the readings before forming their own opinions. An alternative, of course, is to schedule them one week *before* the topic is begun to be covered in class, so that the students will be foreced to think about the issues before reading what "experts" have had to say. Another option is to do both: Assign the first draft before the topic is begun, then have the students do the required readings and participate in class discussions of the topic, then follow this with an optional revision or second draft of the position paper and the peer-editing session, with a third draft (or second, if the optional, post-class-discussion draft is omitted) to be turned in for instructor evaluation.

Peer-editing sessions should take up a full class period. The general method is to divide the class into small groups of three or four students (two students will work if the number of students in the class—or latecomers!—demands it, but is not ideal; five-student groups are too large to enable everyone to participate, especially if time is limited). For each student in a group:

1. Have the group read the student's position paper.

2. Have the group challenge the student's position, ask for clarification, and make recommendations for improving the student's paper.

If there are s students in a group and the peer-editing session lasts for m minutes, then the group should spend no more than m/s minutes on each student's paper. The instructor should visit each group at least once to ensure that all is going well, to answer

questions, and to suggest questions if the group seems to be having trouble. If a group ends early and there is a lot of time left in the session, ask each student in the group to join another group (even if only to listen in to that group's ongoing discussion, but if that other group also ended early, then the newcomer should peer-edit one of their papers). Specific information for peer-editing sessions is given with each assignment.

After peer-editing, students should revise their position papers in the light of the editing suggestions and hand in all drafts to the instructor. I usually give the students one week for this revision.

B.3 Position Paper #1: What Is Computer Science?

B.3.1 Assignment

B.3.1.1 Introduction

The purpose of this position paper is to give you an opportunity to clarify *your* beliefs about what computer science is, so that, as we continue to discuss the topic in class, and as you continue to read about it, you'll know where *you* stand—what *your* beliefs are.

Later, when your beliefs have been "contaminated" by further readings and by our discussions, you may wish to *revise* your beliefs. But you can't revise a belief that you don't have (you can only acquire new beliefs). So, here I am forcing you to *discover*, *clarify*, and *defend* the beliefs that you *now* have, by turning them into words and putting them on paper.

B.3.1.2 The Argument

Imagine that you are the newly-appointed Dean of the School of Science at the University of X. In an attempt to build up the rival School of Engineering, the newly-appointed Dean of Engineering has proposed to the Provost (the boss of both deans) that the Department of Computer Science be moved—lock, stock, and computer, so to speak[1]—to Engineering, on the following grounds:

1. Science is the systematic observation, description, experimental investigation, and theoretical explanation of natural phenomena.

2. Computer science is the study of computers and related phenomena.

3. Therefore, computer science is not a science.

(The Dean of Engineering has not *yet* argued that computer science is an engineering discipline; that may come later.)

How do you respond to the Dean of Engineering's argument?

You may agree with it, or not (but there are several ways that might happen; see below).

---

[1] http://www.worldwidewords.org/qa/qa-loc1.htm

You should ignore political considerations: You may suppose that the move from Science to Engineering involves no loss or gain of money, prestige, or anything else, and it is to be done, if at all, only on strictly intellectual grounds.

The Provost is eagerly awaiting your reply, and will abide by your decision...*if*, that is, you give a well-argued defense of your position.

To formulate and defend your position, you should:

a) Say *whether* you agree that conclusion 3 *logically* follows from premises 1 and 2,

   (whether or not you agree with 1 and 2)

   and *why* you think it follows or doesn't.

   ("I agree that conclusion 3 follows from premises 1 and 2 because..."
   OR "I do not agree that conclusion 3 follows from premises 1 and 2 because...")

   - If you think that conclusion 3 *doesn't* follow, is there some (interesting, non-trivial) missing premise (that is, a "missing link" between the premises and conclusion) that would make it follow?[2]

b) Say *whether* you agree with premise 1, and *why* you do or don't.

   ("I agree with premise 1 because ..." OR "I disagree with premise 1 because ...")

c) Say *whether* you agree with premise 2, and *why* you do or don't.

   ("I agree with premise 2 because ..." OR "I disagree with premise 2 because ...")

d) If you thought that there were missing premises that validated the argument, say *whether* you agree with them, and *why* you do or don't.

e) If you think that the argument is logically *invalid*, you might still agree or disagree with conclusion 3 independently of the reasons given for it by premises 1 and 2 (and any missing premises).

   - If so, state *whether* you agree with 3, and *why*.

It's also possible that you might **neither** agree **nor** disagree with 3; alternatively, you might **both** agree **and** disagree with it.

- For example, you might believe that computer science is *both* a science *and* an engineering discipline (or, alternatively, that it is neither).

- If so, then please give your reasons for this.

---

[2]For a discussion of missing premises, see
http://www.cse.buffalo.edu/∼rapaport/584/S10/EMAIL/20100118-MissingPremises.txt
"Argument Analysis: Finding Missing Premises". INCLUDE THIS IN THE BOOK SOMEWHERE

And, if you are unsure about any of your answers, try to be very precise about *why* you are unsure and what further information would help you decide.

    Other responses:
You might not agree with any of these ways to respond. However, I believe that any other response can, perhaps with a bit of force, be seen to fall under one of the above responses. But if you really feel that your position is not exactly characterized by any of the above responses, then please say:

- what your position is,

- why you believe it,

- and why you think it is not one of the above.

    For general assistance on analyzing arguments, see
http://www.cse.buffalo.edu/∼rapaport/584/S10/arganal.html
A Heuristic for Argument Analysis, and the links at
http://www.cse.buffalo.edu/∼rapaport/584/arganalhome.html
Argument Analysis ADD THESE TO THE TEXT
    Ground Rules:

1. Your answer should honestly reflect *your* beliefs (not what you think the fictional Provost or Dean of Engineering wants to hear!).

2. If you resort to a dictionary, textbook, article, website, etc., be sure to say which one. Give as much detailed information as you can that would assist someone else to locate the item by themselves. (See the
   http://www.cse.buffalo.edu/∼rapaport/howtowrite.html#citations
   "How to Handle Citations" section of my website on "How to Write", for the proper way to do this.)

3. Your position paper should be approximately **1 typed page and double-spaced (that is, about 250 words)** (not including any bibliographic citations).

   - To help keep your paper short, you do not need any fancy introductory paragraph; you can assume that your reader is a fellow student in this course who has just done the same assignment.
   - If you write:
     - 1 paragraph analyzing validity,
     - 1 paragraph each analyzing the premises,
     - and 1 paragraph analyzing the conclusion,

     you will have (more than) enough material.

4. Please bring **5 copies** to **lecture** on the due date.

5. At the top of the page, please put all and only the following information:

   a) the title "Position Paper #1"

   b) your name

   c) the course you are enrolled in

   d) the due date.

   (The space taken up by this will not count against your total pages.)

6. For general assistance with writing (including my preferred method of paper preparation and format, as well as advice on grammar), see my website http://www.cse.buffalo.edu/~rapaport/howtowrite.html

   DUE AT THE BEGINNING OF LECTURE, 1 WEEK FROM TODAY

B.3.2   Suggestions and Guidelines for Peer-Group Editing

1. When you get into your small groups:

   - introduce yourselves quickly,
   - share copies of your papers with each other,
   - and write each other's names on your paper (so that we have a record of who peer-reviewed whom)

2. Choose one paper to discuss first.

   - Suggestion: Go in alphabetical order by family name.

3. The other people in the group might find it useful to imagine themselves as members of a committee set up by the Provost to make a recommendation. Their purpose is to try to help the author clarify his or her beliefs and arguments, so that they will be able to make a recommendation to the Provost on purely logical grounds (again: ignore politics!).

4. Start by asking the author to state (or read) his or her beliefs about whether computer science is a science, giving his or her *reasons* for those beliefs.

5. Be sure that the author has discussed:

   a) the validity of the argument

   b) the truth value of (or their (dis)agreement with) premise 1

   c) the truth value of (or their (dis)agreement with) premise 2

   d) the truth value of (or their (dis)agreement with) any missing premises.

   e) the truth value of (or their (dis)agreement with) the conclusion

   - And for each of the above, their *reasons*

6. Any time you have a question, ask it. Here are some suggestions:

   - Why did you say ____ rather than ____?

- What did you mean when you said _____?

- Can you give me an example of _____?

- Can you give me more details about _____?

- Do you think that _____ is always true?

- Why? (This is always a good question to ask.)

- How?

7. The author should not get defensive. The committee members are friendly. Critical, but friendly.

8. Keep a written record of the questions and replies. This will be useful to the author, for revision.

9. After spending *about* 10 minutes[3] on the first paper, move on to the next, going back to step 2 above, changing roles. Spend *no more than* 15 minutes[4] per paper (because you've only got about 45 minutes[5] at most). Perhaps one member of the group can be a timekeeper.

10. At home, over the next week, please *revise* your paper to take into consideration the comments made by your fellow students (that is, your "peers"):

    Perhaps defend your claims better, or clarify statements that were misunderstood, etc. For help, see your instructor.

    - At the top of the first page of your revision, please put all and only the following information:

        a) the title "Position Paper #1: 2nd Draft"

        b) your name

        c) the course you are enrolled in

        d) the due date

    - Please **staple** copies of your first draft, (with peer-editing comments, if any) to your second draft.

    - Your second draft should be substantially different from your first draft!

1–2 PAGE (250–500 WORD) REVISION, 1 COPY, TYPED, DUE ONE WEEK FROM TODAY. NO LATE PAPERS WILL BE ACCEPTED!

---

[3]Actually, some number n of minutes close to 10 but less than or equal to m/s, where m is the total number of minutes in the class, and s is the total number of students in the group.

[4]Actually, $n + 5$.

[5]Actually, m.

B.3.3 Grading

B.3.3.1 Philosophy of Grading

To make grading these essays easier on the instructor and easy for students (in case the instructor decides to have students grade each other's essays), I recommend using the "triage" method of grading. On this method, each item to be graded is given:

- full credit (for example, 3 points) if it is clearly done in a completely acceptable manner (even if it might not be entirely "correct")

- partial credit (for example, 2 points) if it is done, but is not clearly worth either full credit or minimal credit

- minimal credit (for example, 1 point) if it is done, but is clearly not done in an acceptable manner.

- no credit (that is, 0 points) if it is omitted.

The advantage to this method of grading is that the grader only has to decide if a response is worth full credit (that is, shows clear understanding of what is expected) or minimal credit (that is, shows clear *mis-* or lack of understanding of what is expected). Any response that is not *clearly* one or the other is given partial credit. And failure to respond, or omission of some requirement, is given no credit. This helps make grading slightly more objective (and certainly easier for novice graders). Details and the theory behind the method are given in [Rapaport, 2011] and online at:
http://www.cse.buffalo.edu/~rapaport/howigrade.html
accessed 19 November 2012.

B.3.4 Grading Rubric

1. Premise 1: Did you state clearly whether you agreed or disagreed with it?

   (Doesn't matter whether you agreed or didn't agree, only with whether you said so.)

   > 3 pts = clearly stated whether you agreed or not
   > 2 pts = not clearly stated but implied
   > 1 pts = stated, but incorrect terminology
   > 0 pts = did not clearly state whether you agreed

2. Did you give your reasons for your (dis)agreement?

   > 3 = reasons given, clearly stated, & pertinent
   > 2 = partial credit: I couldn't decide between 1 & 3
   > 1 = reasons given, but not clearly stated or not pertinent
   > 0 = no reasons

3. Premise 2: Did you state clearly whether you agreed or disagreed with it?

   (Doesn't matter whether you agreed or didn't agree, only with whether you said so.)

      0, 1, 2, or 3, as for Premise 1

4. Did you give your reasons for your (dis)agreement?

      0,1,2, or 3, as for Premise 1

5. Valid?

      no answer: 0

      yes; XOR no, but no MP: 1

      no, with wrong MP
      or yes, with right MP: 2

      no, with right MP: 3

   (MP [Missing Premise] = computers are not natural phenomena)

6. Evaluation of MP (agree? why?):

      no evaluation 0
      dis/agree, but no reason 1
      dis/agree, w/ unclear reason 2
      dis/agree, w/ clear/pertinent reason 3

7. Conclusion: Did you state clearly whether you agreed or disagreed with it?

      3 = clearly stated whether you agreed
      2 = not clearly stated, but implied
      0 = did not clearly state whether you agreed

8. Did you give your reasons for your (dis)agreement?

      0,1,2, or 3 points, as for Premise 1

9. Citation style:

      used sources w/o citing −3
      used sources w/ incomp & incorrect citation −2
      used sources w/ incomp XOR incorrect citation −1
      not applicable 0

10.   Attached draft 1 & list of peer editors to demonstrate that draft 2 $\neq$
        draft 1? 0
        Didn't -1

The total is 24 points, which, following my grading theory, maps into letter grades as follows:

    A   23–24
    A–   22
    B+   21
    B   19–20
    B–   18
    C+   17
    C   14–16
    C–   11–13
    D+   9–10
    D   5–8
    F   0–4
On my grading scheme,
'A' means "understood the material for all practical purposes",
(here, that's 24 pts = 8 parts $\times$ 3 pts full credit)

'B' has no direct interpretation, but comes about when averaging A and C grades

'C' means "average",
(here, that's max 16 pts = 8 $\times$ 2 pts partial credit)

'D' means "did not understand the material", (here, that's max 8 pts = 8 $\times$ 1 pt minimum credit)

'F' usually means "did not do the work" (that is, 0 pts), but can also come about when omitting some parts and doing D work on others.

B.4   Position Paper #2: What Is Computation?

B.4.1   Assignment

For this position paper, please evaluate the following argument:

1. [Knuth, 1973, 4–6] characterizes the informal, intuitive notion of "algorithm" as follows:

   a) "**Finiteness.** An algorithm must always terminate after a finite number of steps. . . "

   b) "**Definiteness.** Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case. . . "

   c) "**Input.** An algorithm has zero or more inputs. . . "

   d) "**Output.** An algorithm has one[6] or more outputs. . . "

---

[6]My former student Albert Goldfain made the following observation on Knuth's characterization of an algorithm:

Knuth's characterization of an algorithm includes:

   1d "**Output.** An algorithm has *one* or more outputs. . . " (my emphasis)

Contrast this with (1c) "*zero* or more inputs" (my emphasis). This is not a typo, and most likely not a mental slip by someone as precise as Knuth. There are several, modern, high-level languages that will allow you to return nothing from a function. For example, consider the *C* function:

```
int store-and-return(int x)

int i;
i=x;
return;
```

In some sense, there are no "outputs" from this function, yet it is still somehow an algorithm. (In the Pascal programming language, there was a distinction between procedures and functions; some other languages call procedures "subroutines". But the point is that these are not functions in the mathematical sense of ordered *pairs*.) In Lisp, which may have been more dominant when Knuth wrote, every function *had* to return at least one value.

We can, however, re-interpret the "output" of an algorithm so that it is not always the return-value of a function. So, in `store-and-return`, perhaps it is the memory location of the local variable `i` that gets the "output" of the algorithm. In a program like *C*'s `Hello World`,

```
main()
printf("Hello World
n");
return;
```

perhaps it is the screen that you are printing to that gets the "output" of the algorithm.

Notice also that if you think of the `store-and-return` function as a "black box" (that is, a device whose internal workings are hidden from you but whose external behavior is observable), there would be no way to tell that it was doing anything. Behaviorally, you would hand an input into the black box and observe nothing else! It may be that Knuth wants to exclude behaviorally unobservable algorithms.

   e) "**Effectiveness.** [A]ll of the operations to be performed in the algorithm must be sufficiently basic that they can in principle be done exactly and in a finite length of time by a [hu]man using pencil and paper..."

Note: We can also say that *A* **is an algorithm for computing a function** *f* means that *A* is an algorithm as characterized above and that, for any input *i*, *A*'s output for *i* = *f*'s output for *i*; that is, for any *i*, $A(i) = f(i)$.

2. Computer programming languages (like Java, Lisp, Fortran, etc.) are formal languages for expressing (or "implementing") algorithms.

3. Every computer programming language is equivalent in expressibility to a Turing-machine programming language.

   a) That is, every program in any programming language can be translated into the language for programming Turing machines, and vice versa.

   b) That is, any function that is computable by any programming language is computable by a Turing machine, and vice versa.

4. Some real computer programs violate parts of definition 1:

   a) For example, airline-reservation systems, ATMs, operating systems, etc., never terminate.

   b) For example, heuristic AI programs don't always compute exactly the function that they were written for, but only come very close. BE SURE TO PUT IN A SECTION ON HEURISTIC PROGRAMS, AND ALSO SITE KAHNEMAN'S BOOK

   c) For example, the "effectiveness" of "mundane" or everyday procedures (like recipes) depends on the environment in which they are executed.

        For example, using a different brand of some ingredient can ruin a recipe, or one chef's "pinch" of salt might be another's 1/8th teaspoon. And can you really execute a recipe "using pencil and paper"?

   d) For example, algorithms can apparently be written that can perform an infinite computation in a finite amount of time (by continually accelerating).

        For example, we can sum the terms of an infinite sequence in a finite amount of time if we take $\frac{1}{2^n}$ second to add the *n*th term.

        And so on.

Bottom line: the "one or more" from premise 1d should not trip you up from analyzing the rest of the argument.

5. Therefore, these programs (that is, the "real programs" referred to in premise 4) do not implement Turing machines (contrary to premise 3).

6. Therefore, they (that is, the "real programs" referred to in premise 4) are not computable. (But how can a real computer program not be computable?!)

To evaluate this argument, you must state whether the argument is **valid**, and you must state whether and why you **agree or disagree with each premise and conclusion**.

- If it is valid, and if you agree with each premise, then you believe that the argument is **sound**.

- You are logically obligated to believe the conclusions of sound arguments!

  So, if you ever come across an argument that you think **is** sound, but whose conclusion you **don't** believe, then either:

  - one (or more) of the premises is false,
  - or the argument is invalid (that is, there is some way for the premises to be true yet for the conclusion to be false),
  - or both.

To determine whether it is valid, you must suppose "for the sake of the argument" that all the premises *are* true, and then consider whether the conclusions logically follow from them.

(Or: Can you imagine some way the world might be so that the premises are true but the conclusion is false?)

Note that, in this argument, there are *two* conclusions: conclusions 5 and 6.

> So, do you agree that conclusion 5 follows logically from premises 1–4 and/or that conclusion 6 follows logically from 5? If not, are there missing premises that are needed to make the argument(s) valid? If there are, do you agree with them (why/why not)?

Next, you must evaluate each premise. Do you agree with it? Why or why not? Finally, do you agree with the conclusion(s)?

> You might agree with a conclusion because you think that the argument is sound;
>
> - if so, say so.

> Or you might think that there's something wrong with the argument but agree with the conclusion anyway;
>
> - if so, then try to present a better argument for the conclusion.

> Or you might not agree with the conclusion(s);
>
> - if not, state why, and try to give an argument for what you *do* believe.

a) Your position paper should be approximately **1–2 typed pages, double-spaced (that is, about 250–500 words), and single-sided.**

b) Please bring **5 copies** to **lecture** on the due date.

c) **At the top of the first page, please put the following information** *in the following format*:

<div align="center">

Position Paper #2, 1st draft   YOUR NAME
DATE DUE   CLASS

</div>

d) For general assistance with writing (including my preferred method of paper preparation and format, as well as advice on grammar), see my website, "How to Write":
http://www.cse.buffalo.edu/~rapaport/howtowrite.html

As before, this doesn't have to be a beautifully written essay with an abstract. You should just plunge in and evaluate the argument.

But you do need to give full citations to any sources that you cite.

DUE AT THE BEGINNING OF LECTURE, ONE WEEK FROM TODAY

B.4.2 Suggestions and Guidelines for Peer-Group Editing

1. When you get into your small groups, introduce yourselves quickly, and share copies of your papers with each other.

2. Choose one paper to discuss first. (Suggestion: Go in alphabetical order by family name.)

3. After spending *about* 10–15 minutes on the first paper, move on to the next, going back to step 2, above, changing roles.
   Spend *no more than* 15 minutes per paper (because you've only got about 45 minutes at most).
   Perhaps one member of the group can be a timekeeper.

4. For each paper, ask as many of the following questions as you have time for:

   a) Did the author state **whether** *and* **why** they did or did not agree with Knuth's definition in premise 1?

      - **Note:** Knuth's definition is a conjunction of 5 things: 1a & 1b & 1c & 1d & 1e.
        So, in **disagreeing** with premise 1, an author must
        a) explicitly disagree with (at least) one of 1a... 1e
        b) **and** say why they disagree with that part (or those parts).

      i) If the author **agreed** and gave reasons for agreeing, do **you** agree with those reasons? Why?

    ii) If the author **disagreed** and gave reasons for disagreeing, do **you** agree with those reasons? Why?

b) Did the author state **whether** *and* **why** they did or did not agree with the claim about the nature of programming languages in premise 2?

    (Plus questions 4(a)i and 4(a)ii, above.)

c) Did the author state **whether** *and* **why** they did or did not agree with the claim about the "Turing-equivalence" of programming languages in premise 3?

    (Plus questions 4(a)i and 4(a)ii, above.)

d) Did the author state **whether** *and* **why** they did or did not agree with the claim and/or the examples in premise 4?

    (Plus questions 4(a)i and 4(a)ii, above.)

e) Did the author state **whether** *and* **why** they believe that conclusion 5 does or does not validly follow from premises 1–4?

    Do **you** agree with their evaluation?

f) If the author believes that conclusion 5 follows **soundly** from premises 1–4, then they should state that they believe conclusion 5 for that reason.
Do they?

    i) On the other hand, if the author believes that conclusion 5 does **not** follow
(either because one or more of the premises is false or because the argument is invalid),
then did the author state **whether** *and* **why** they did or did not agree with the statement made in the conclusion?

        (Plus questions 4(a)i and 4(a)ii, above.)

    ii) Note that if the author believes that the argument is *unsound*,
that is **not** a sufficient reason for disbelieving the claim!

        (That's because even a **valid** argument can have **both** false premises **and** a true conclusion (or a false one),
and even an **invalid** argument can have a **true** conclusion (or a false one).
The only thing that can't happen is to have a valid argument both with true premises and with a false conclusion.)

g) If the author believes that conclusion 6 follows **soundly** from statement 5 considered as a premise along with some or all of the previous statements in the argument (and possibly along with one or more missing premises!), then they should state that they believe conclusion 6 for that reason. Do they?

    • On the other hand, if the author believes that conclusion 6 does **not** follow
(either because one or more of its premises is false or because the argument is invalid),
then did the author state **whether** *and* **why** they did or did not agree with the statement made in the conclusion?

(Plus questions 4(a)i and 4(a)ii, above.)

5. Keep a written record of the questions and replies. This will be useful to the author, for revision.

6. At home, over the next week, please *revise* your paper to take into consideration the comments made by your fellow students (that is, your "peers"):
Perhaps defend your claims better, or clarify statements that were misunderstood, etc.
For help, see your instructor.

1–2 PAGE (250–500 WORD) REVISION, 1 COPY, TYPED, DOUBLE-SPACED, IS DUE ONE WEEK FROM TODAY.
NO LATE PAPERS WILL BE ACCEPTED!

B.4.3 Grading Rubric

```
1.  Prem 1 (Knuth's characterization of "algorithm")

agree? why? 0 = no answer
1 = answer, no reason
2 = answer, unclear reason
3 = answer, clear reason


----------------------------------------------------------------------


2.  Prem 2 (Prog langs express/implement algorithms)

agree? why? 0,1,2,3, as above

----------------------------------------------------------------------


3.  Prem 3 (Prog langs are equiv to TM prog lang)

agree? why? 0,1,2,3, as above

----------------------------------------------------------------------


4.  Prem 4 (Some real comp progs violate Knuth's def)

agree? why? 0,1,2,3, as above

----------------------------------------------------------------------


5.  Conc 5 (So, such progs don't implement TMs)

1..4/.'.5 valid?
```

```
Why?

0 = no answer
1 = answer, no explanation
2 = answer, weak explanation
3 = answer, good explanation

agree w/ 5?
why? 0,1,2,3, as for 1
```

```
6.  Conc 6 (So, such progs are not computable)

        5/.'.6 valid?
Why?

        0 = no answer
                        1 = answer, no explanation
                        2 = answer, weak explanation
                        3 = answer, good explanation

        agree w/6?
why?    0,1,2,3, as for 1
```

```
7.  Citation style:

        used sources w/o citing                     -3
        used sources w/ incomp & incorrect citation  -2
        used sources w/ incomp XOR incorrect citation -1
        not applicable                                0
```

```
8.  Attached draft 1 & list of peer editors
to demo that draft 2 <> draft 1?                0
    Didn't                                          -1
```

===

```
The total is 24 points, which, following my grading theory, maps into
letter grades as follows:

letter  CSE484          both    CSE/PHI584
```

```
A                        23-24
A-                        22
B+                        21
B                        19-20
B-                        18
C+                        17
C     14-16                          9-16
C-    11-13
D+     9-10
D                         5-8
F                         0-4
```

B.5   Position Paper #3: Is the Brain a Computer?

B.5.1   Assignment

For this position paper, I would like you to evaluate the following "complex" argument. (It's "complex" because it consists of three "sub"arguments, two of which treat the conclusions of previous ones as premises.)

1. **Turing's Thesis:** A physical object can compute if and only if it can do what a (Universal) Turing Machine (TM) can do.

2. A computer is any physical device that can compute. (Consider this as a (proposed) definition of 'computer'.)

3. The human brain is a physical object that can do what a (Universal) TM can do.

4. Therefore, the human brain is a computer.

5. Microsoft Word is TM-computable. (That is, a Universal TM can execute Microsoft Word.)

6. Therefore, any computer can execute Microsoft Word.

7. Therefore, the human brain can execute Microsoft Word.

As usual, to evaluate this argument, you must determine whether (I) it is **valid** and whether (II) **all the premises are true**.

- If both of those conditions hold, then an argument is said to be **sound**.

- You are logically obligated to believe the conclusions of sound arguments!

  - So, if you ever come across an argument that you think **is** sound, but whose conclusion you **don't** believe

    (by the way, do you really believe line 7 of this argument?),

    then **either** one or more of the premises are false **or** it is invalid (that is, there is some way for the premises to be true yet for the conclusion to be false).

(I) To determine whether the argument is valid, you must suppose (or make believe) "for the sake of the argument" that all the premises **are** true, and then consider whether the conclusions logically follow from them. (Or: Can you imagine some way the world might be so that the premises **are** true but the conclusion is **false**?)

  - Note that there are *three* conclusions: lines 4, 6, and 7. So, do you agree that conclusion 4 follows logically from premises 1–3, and/or that conclusion 6 follows logically from premise 5 (maybe with the help of some of the earlier premises), and/or that conclusion 7 follows logically from lines 4 and 6 *considered as premises*?

    If not, are there missing premises that are needed to make the argument(s) valid? If there are, do you agree with them (why/why not)?

(II) It may be too difficult to determine whether each premise *is* true or false. More realistically, you should decide whether you *believe*, or agree with, each premise, **and** you must explain why you do or don't.

Finally, do you agree with the conclusion(s)? If you do, but you think that there's something wrong with the argument, try to present a better one. If you don't agree with the conclusion(s), state why, and try to give an argument for what you *do* believe.

a) Your position paper should be approximately

> **1–2 typed pages, double-spaced (that is, about 250–500 words), and single-sided.**

b) Please bring **5 copies** to lecture on the due date.

c) At the top of the first page, please put the following information **in the following format**:

> Position Paper #3, Draft 1 YOUR NAME
> DATE DUE   YOUR CLASS

For general assistance with writing (including my preferred method of paper preparation and format, as well as advice on grammar), see my website "How to Write"
http://www.cse.buffalo.edu/~rapaport/howtowrite.html
As before, no abstract is needed for this position paper, but you do need to give full citations to any sources that you cite.

DUE AT THE BEGINNING OF LECTURE, ONE WEEK FROM TODAY

B.5.2 Suggestions and Guidelines for Peer-Group Editing

d) When you get into your small groups, introduce yourselves quickly, and share copies of your papers with each other.

2. Choose one paper to discuss first. (Suggestion: Go in alphabetical order by family name.)

3. After spending *about* 10–15 minutes on the first paper, move on to the next, going back to step 2, above, changing roles. Spend *no more than* 15 minutes per paper (because you've only got about 45 minutes at most). Perhaps one member of the group can be a timekeeper.

4. For each paper, ask as many of the following questions as you have time for:

   a) Did the author state whether the argument from premises 1–3 to conclusion 4 was valid?

      i) If they thought it was invalid, did they suggest a missing premise that would make it valid (if that's possible)?

b) Did the author state whether the argument to conclusion 6 was valid?

    i) Did they correctly identify its other premises besides premise 5? (Very few real arguments can have only one premise.)

    ii) If they thought it was invalid, did they suggest a missing premise that would make it valid (if possible)?

c) Did the author state whether the argument to conclusion 7 was valid?

    i) Did they correctly identify its other premises besides premise 6? (Note that sentence 6 is both the conclusion of the previous argument and a premise of this one.)

    ii) If they thought it was invalid, did they suggest a missing premise that would make it valid (if possible)?

d) For each premise, ask whether the author stated whether **and** why they did or did not agree with it.

    i) If the author **agreed**, then it is preferable (but not necessary) that they give reasons for agreeing. If they did give such reasons, do **you** agree with those reasons? Why?

    ii) If the author **disagreed**, then it **is** necessary that they give reasons for disagreeing, so do **you** agree with those reasons? Why?

e) For each argument, if the author thought it was **unsound**, did they state whether they believed its conclusion anyway, on independent grounds (that is, for different reasons)?

    • And, if so, do you agree with those reasons?

5. Keep a written record of the questions and replies. This will be useful to the author, for revision.

6. At home, over the next week, please *revise* your paper to take into consideration the comments made by your fellow students (that is, your "peers"): Perhaps defend your claims better, or clarify statements that were misunderstood, etc. For help, see your instructor.

1–2 PAGE (250–500 WORD) REVISION, 1 COPY, TYPED, DOUBLE-SPACED, IS DUE IN LECTURE ONE WEEK FROM TODAY. NO LATE PAPERS WILL BE ACCEPTED!

B.5.3   Grading Rubric

```
Position Paper #3 Grading Rubric
Version: 14 Mar 10
========================================================================
I. VALIDITY OF ARGUMENTS:

 1.  Argument 1,2,3/.'. 4: valid? + reason
```

```
0 = no answer
1 = answer, no reason
    OR reason that confuses definitions
of 'valid'/'invalid'/'sound'
2 = answer, weak reason
3 = answer, good reason
```

```
 2.  Argument 1,2,5/.'. 6: valid? + reason
0,1,2,3 as above
```

```
 3.  Argument 4,6/.'. 7: valid? + reason 0,1,2,3 as above
```

```
II.  TRUTH VALUES OF STATEMENTS:

 4.  Prem 1: agree? + why? 0,1,2,3 as above

 5.  Prem 2: agree? + why? 0,1,2,3

 6.  Prem 3: agree? + why? 0,1,2,3

 7.  Conc 4: agree? + why? 0,1,2,3

 8.  Prem 5: agree? + why? 0,1,2,3

 9.  Conc 6: agree? + why? 0,1,2,3

10.  Conc 7: agree? + why? 0,1,2,3
```

==

```
The total is 30 points, which, following my grading theory, maps into
letter grades as follows:

letter CSE484 both CSE/PHI584

  A 29-30
  A-27-28
  B+  26
  B 24-25
  B-22-23
```

```
  C+  21
  C 17-20 11-20
  C-14-16
  D+ 11-13
  D 6-10
  F 0-5
```

As usual, on my grading scheme,

"A" means "understood the material for all practical purposes",
(here, that's 30 pts = 10 questions * 3 pts full credit)

"B" has no direct interpretation,
    but comes about when averaging grades of A and C

"C" means "average",
(here, that's 20 pts = 10 * 2 pts partial credit)

"D" means "did not understand the material,
(here, that's 10 pts = 10 * 1 pt minimum credit)

"F" usually means "did not do the work" (that is, 0 pts),
    but can also come about when averaging Ds and Fs.

Please see my grading website,

http://www.cse.buffalo.edu/˜rapaport/howigrade.html

for the theory behind all of this, which I'm happy to discuss via
UBLearns email.

B.6   Position Paper #4: What Is a Computer Program?

B.6.1   Assignment

B.6.1.1   The Argument

For this position paper, I would like you to evaluate the following argument:

1. A special-purpose computer (that is, a computer that does just one task) is essentially a hardwired computer program.

2. Such a hardwired computer program is a physical machine.

3. Physical machines can be patented.

4. Therefore, such a hardwired computer program can be patented.

5. The printed text of a computer program is a "literary work" (that is, a piece of writing) in the sense of the copyright law.

6. Literary works can be copyrighted.

7. Therefore, such a computer program can be copyrighted.

8. Nothing can be both patented and copyrighted.

    - **Note:** This premise is a matter of law. You must accept it as true. But you can argue that the law should be changed.

9. There is no computational or other relevant difference between the hardwired computer program and its textual counterpart (except for the different media in which they are implemented, one being hardwired and the other being written on, say, a piece of paper).

10. Therefore, computer programs can be both patented and copyrighted.

To help you evaluate this argument (which we look at in more detail in Ch. 13), here are some extracts from some relevant websites:

a) From the official US Patent Office definition of 'patent':

> a property right granted by the Government of the United States of America to an inventor "to exclude others from making, using, offering for sale, or selling the invention throughout the United States or importing the invention into the United States" for a limited time in exchange for public disclosure of the invention when the patent is granted.
> http://www.uspto.gov/main/glossary/index.html#patent
> accessed 22 November 2012.

b) The Patent Office definition of 'invention':

any art or process (*way of doing or making things*), machine, manufacture, design, or composition of matter, or any new and useful improvement thereof, or any variety of plant, which is or may be patentable under the patent laws of the United States.
http://www.uspto.gov/main/glossary/index.html#invention
accessed 22 November 2012.

c) The official US Copyright Office definition of 'copyright':

Copyright is a form of protection provided by the laws of the United States...to the authors of "original works of authorship," including literary, dramatic, musical, artistic, and certain other intellectual works. This protection is available to both published and unpublished works.
http://www.copyright.gov/circs/circ01.pdf
accessed 22 November 2012.

d) From the same website:

Copyrightable works include the following categories:

 (a) literary works

 (b) musical works, including any accompanying words

 (c) dramatic works, including any accompanying music

 (d) pantomimes and choreographic works

 (e) pictorial, graphic, and sculptural works

 (f) motion pictures and other audiovisual works

 (g) sound recordings

 (h) architectural works

These categories should be viewed broadly. For example, computer programs and most "compilations" may be registered as "literary works"; maps and architectural plans may be registered as "pictorial, graphic, and sculptural works."

WHAT IS NOT PROTECTED BY COPYRIGHT?

Several categories of material are generally not eligible for federal copyright protection. These include among others:

- works that have not been fixed in a tangible form of expression (for example, choreographic works that have not been notated or recorded, or improvisational speeches or performances that have not been written or recorded)

- titles, names, short phrases, and slogans; familiar symbols or designs; mere variations of typographic ornamentation, lettering, or coloring; mere listings of ingredients or contents

- ideas, procedures, methods, systems, processes, concepts, principles, discoveries, or devices, as distinguished from a description, explanation, or illustration

- works consisting entirely of information that is common property and containing no original authorship (for example: standard calendars, height and weight charts, tape measures and rulers, and lists or tables taken from public documents or other common sources)

  http://www.copyright.gov/circs/circ01.pdf
  accessed 22 November 2012.

To evaluate this argument, you must state whether the argument is **valid** and you must state whether and why you **agree or disagree with each premise**. Remember:

- **Only single statements (like premises and conclusions) can be true or false.** For our purposes, it's enough to say that a statement is true (or false) if you agree (or disagree) with it, because I'm not asking you to convince me that a statement really is true (or false); I'm only asking you to convince me that you have a good reason for agreeing (or disagreeing) with it.

- And **only arguments can be valid or invalid.** An argument is valid if it's impossible for all of its premises to be true while its conclusion is false (and it's invalid otherwise). For our purposes, to determine whether an argument is **valid**, you must suppose (or make believe) "for the sake of the argument" that all the premises *are* true (that is, that you agree with all of them), and then consider whether you would have to logically agree with the conclusion. To determine whether an argument is **invalid**, try to imagine some way the world might be so that the premises are true but the conclusion is false.

- Finally, **only arguments can be sound or unsound.** An argument is sound if it's valid and all of its premises are true (in which case, its conclusion will also have to be true). For our purposes, we'll say that an argument is sound if it's valid and you really do agree with all of its premises (in which case, you really have to agree with the conclusion).

- **You are logically obligated to believe the conclusions of sound arguments!** So, if you ever come across an argument that you think is sound, but whose conclusion you don't believe, then either one (or more) of the premises is false, or it is invalid (that is, there is some way for the premises to be true yet for the conclusion to be false), or both.

This means, of course, that you have to evaluate each premise and each (sub-)argument, and, as usual, I also want you to evaluate the conclusion independently of whether you think that it follows validly or doesn't follow validly from its premises.

a) For this position paper, I want to experiment with something a little bit different.

   Instead of writing a first draft of your paper, I simply want you to fill in the attached "thinksheet", which will be an outline of your argument analysis.

   You will write the paper after peer-editing the thinksheets.

b) Please bring **5 copies** of your filled-out thinksheet to lecture on the due date.

   DUE AT THE BEGINNING OF LECTURE, ONE WEEK FROM TODAY

B.6.1.2   Thinksheet

```
PhilCS 4/584, Spring 2010 NAME:
=============================================================================
Thinksheet for Position Paper #4:  What Is a Computer Program?
=============================================================================
Statement               |Agree?|   Why?
(abbreviated versions of|(T?F?)|   (use additional sheets if needed)
 prems & conc of arg't) |      |
-----------------------------------------------------------------------------
1. A special-purpose     |      |
   computer is           |      |
   essentially a hard-   |      |
   wired computer prog.  |      |
                         |      |
                         |      |
                         |      |
                         |      |
                         |      |
                         |      |
-----------------------------------------------------------------------------
2. Such a hardwired      |      |
   comp.prog. is a       |      |
   physical machine      |      |
                         |      |
                         |      |
                         |      |
                         |      |
                         |      |
                         |      |
                         |      |
-----------------------------------------------------------------------------
3. Physical machines     |      |
   can be patented       |      |
                         |      |
                         |      |
                         |      |
                         |      |
                         |      |
                         |      |
                         |      |
-----------------------------------------------------------------------------
4. Such a hardwired      |      |
   comp.prog. can be     |      |
   patented              |      |
```

```
                              |        |
                              |        |
                              |        |
                              |        |
                              |        |
                              |        |
                              |        |
------------------------------------------------------------------------
  1,2,3/.'.4 is valid         |        |
                              |        |
                              |        |
                              |        |
                              |        |
                              |        |
------------------------------------------------------------------------
5. The printed text of        |        |
   a comp.prog. is a          |        |
   ``lit.work'' in the        |        |
   sense of the (c) law       |        |
                              |        |
                              |        |
                              |        |
                              |        |
                              |        |
------------------------------------------------------------------------
6. Lit.works can be (c)       |        |
                              |        |
                              |        |
                              |        |
                              |        |
                              |        |
                              |        |
                              |        |
                              |        |
------------------------------------------------------------------------
7. The printed text of        |        |
   a comprog can be (c)       |        |
                              |        |
                              |        |
                              |        |
                              |        |
                              |        |
                              |        |
```

```
                             |      |
                             |      |
-------------------------------------------------------------------------
  5,6/.'.7 is valid          |      |
                             |      |
                             |      |
                             |      |
                             |      |
                             |      |
                             |      |
                             |      |
                             |      |
                             |      |
-------------------------------------------------------------------------
8. Nothing can be both       |      |
   patented & (c)            |      |
                             |      |
                             |      |
                             |      |
                             |      |
                             |      |
                             |      |
                             |      |
                             |      |
-------------------------------------------------------------------------
9. There's no comp'n'l       |      |
   or other diff. betw.      |      |
   the hardwired comp.       |      |
   prog. & its textual       |      |
   counterpart...            |      |
                             |      |
                             |      |
                             |      |
                             |      |
-------------------------------------------------------------------------
10. Comprogs can be both     |      |
    patented & (c)           |      |
                             |      |
                             |      |
                             |      |
                             |      |
                             |      |
                             |      |
```

```
                                 |        |
 ------------------------------------------------------------------------
  4,7,9/.'.10 is valid    |        |
                          |        |
                          |        |
                          |        |
                          |        |
                          |        |
                          |        |
                          |        |
                          |        |
                          |        |
 ------------------------------------------------------------------------
Additional comments:
```

B.6.2 Suggestions and Guidelines for Peer-Group Editing

A) (a) When you get into your small groups, introduce yourselves quickly, and share copies of your thinksheets with each other.

(b) Choose one thinksheet to discuss first. (Suggestion: Go in alphabetical order by family name.)

(c) After spending *about* 10–15 minutes on the first thinksheet, move on to the next, going back to step Ab, above, changing roles. Spend *no more than* 15 minutes per paper (because you've only got about 45 minutes at most). Perhaps one member of the group can be a timekeeper.

B) (a) **Make sure each "cell" of the thinksheet is filled in.**

  i. **The cells in the "Agree? (T?F?)" column should be filled in with 'agree' or 'disagree' (or 'T' or 'F').**

  ii. **The cells in the "Why?" column should contain a reason why the author agrees or disagrees with the statement, or why the author thinks that the argument is valid or invalid.**
  (For a reminder about how to explain why you think that an argument is, or isn't, valid,
  584/S10/EMAIL/20100323-PP3-Comments.txt
  see my earlier email (point #2).) FIX THIS.
  These don't have to be complete sentences, but they should be comprehensible.

(b) Keep a written record of the peer-editing suggestions. This will be useful to the author, for revision.

C) The "revision" this time should, of course, be a correctly formatted paper, like the ones you have been writing all semester.

(a) It should be fairly straightforward to turn the thinksheet outline into full sentences and paragraphs (with correct citations if needed).

(b) **I strongly urge you to have someone peer-edit your paper before you produce the final version! Tell that person to make sure that you have:**

  i. **Evaluated each statement (premise and conclusion) for ("absolute" or "independent") truth or falsity (see my**
  584/S10/EMAIL/20100323-PP3-Comments.txt
  email (point #1) about that terminology,
  FIX THIS.
  and given a reason for your evaluation.

  ii. **Evaluated each argument for validity or invalidity**
  584/S10/EMAIL/20100323-PP3-Comments.txt
  FIX THIS
  (that is, evaluated each conclusion for "relative truth", that is, truth relative to the premises), and given a reason for your evaluation.

  iii. **Correctly used the "true"/"false"/"valid"/"invalid" terminology.**

(c) Failure to correctly distinguish among **"true (or false) sentences, propositions, statements, premises, or conclusions"** and **"valid (or invalid) arguments"** will result in a lower grade! (After all, you need to demonstrate that you've learned something this semester!)

D) (a) Your position paper should be approximately **1–2 typed pages, double-spaced (that is, about 250–500 words), and single-sided.**

(b) At the top of the first page, please put the following information **in the following format**:

>   Position Paper #4   YOUR NAME
>   DATE DUE   YOUR CLASS

(c) Please attach the peer-edited thinksheets to your paper, as usual.

(d) For general assistance with writing (including my preferred method of paper preparation and format, as well as advice on grammar), see my website "How to Write".
http://www.cse.buffalo.edu/~rapaport/howtowrite.html
accessed 22 November 2012.

As before, no abstract is needed for this position paper, but you do need to give full citations to any sources that you cite.

1–2 PAGE (250–500 WORD) PAPER, 1 COPY, TYPED, SINGLE-SIDED, DOUBLE-SPACED, IS DUE ONE WEEK FROM TODAY. NO LATE PAPERS WILL BE ACCEPTED!

B.6.3   Grading Rubric

```
Position Paper #4 Grading Rubric
Version:  28 Mar 10
==========================================================================
In general:


3 = Statement of position ("agree"/"disagree", or "valid"/"invalid")
    with a clearly stated reason


2 = Statement of position, with an unclear reason


1 = Statement of position, with no reason given


0 = no response


I WILL DEDUCT 3 POINTS FROM THE TOTAL GRADE FOR THE PAPER
FOR INCORRECT USE OF THE TERMINOLOGY!
(If you are not sure of how to use the terminology, please ask me!)


AND MY OFFER TO GIVE YOU BACK ANY SUCH LOST POINTS
```

(OR LOST POINTS FOR INCORRECT USE OF CITATIONS)
STILL HOLDS ON ALL POSITION PAPERS.

a) Evaluation of premise 1:   Agree? Why? 0,1,2,3
b) Evaluation of premise 2:   Agree? Why? 0,1,2,3
c) Evaluation of premise 3:   Agree? Why? 0,1,2,3
d) Evaluation of statement 4: Agree? Why? 0,1,2,3
f) Evaluation of 1,2,3/.'.4:   Valid? why? 0,1,2,3

g) Evaluation of premise 5:   Agree? Why? 0,1,2,3
h) Evaluation of premise 6:   Agree? Why? 0,1,2,3
i) Evaluation of statement 7: Agree? Why? 0,1,2,3
j) Evaluation of 5,6/.'.7: Valid? Why? 0,1,2,3

k) Discussion of premise 8:   Agree? Why? 0,1,2,3

Since conclusion 10 conflicts with the law (8),
you have two options:

        Either accept the law, and reject 10...
        [In that case, you must reject at least one of 1-7, & 9.
        Which one, and why? (This is the "Alan Newell" solution.)]

        ...or else reject the law, and accept 10.
        [Of course, you can't do that in real life unless maybe you're a
        legislator (who can can write new laws)(see footnote (*), below)
        or a Supreme Court justice (who can declare laws unconstitutional)
        (see footnote (+), below.) In that case, you should propose a new
        law. (This is the "David Koepsell" solution.)

l) Evaluation of premise 9:   Agree? Why? 0,1,2,3
m) Evaluation of statement 10: Agree? Why? 0,1,2,3
n) Evaluation of 4,7,9/.'.10: Valid? Why? 0,1,2,3

===

The total is 39 points, which, following my grading theory, maps into
letter grades as follows:

letter CSE484 both CSE/PHI584

  A 37-39
  A-35-36
  B+ 33-34

```
B 31-32
B-29-30
C+ 27-28
C 21-26 14-26
C-18-21
D+ 14-17
D 7-13
F 0-6
```

On my grading scheme,

"A" means "understood the material for all practical purposes",
(here, that's 39 pts = 13 questions * 3 pts full credit)

"B" has no direct interpretation,
    but comes about when averaging grades of A with Cs.

"C" means "average",
(here, that's 26 pts = 13 * 2 pts partial credit)

"D" means "did not understand the material,
(here, that's 13 pts = 13 * 1 pt minimum credit)

"F" usually means "did not do the work" (that is, 0 pts),
    but can also come about when averaging Ds and Fs.

Please see my grading website,

http://www.cse.buffalo.edu/~rapaport/howigrade.html

for the theory behind all of this, which I'm happy to discuss via email.

Footnotes:
(*) Compare: Propose new axioms?
(+) Compare: Prove that a law is not a theorem of the US Constitution?

(There's a story that the famous logician Kurt Goedel found an
inconsistency in the US Constitution when he was studying for his
American citizenship. He was going to tell the judge about it, but
Albert Einstein, who accompanied him to the ceremony, quickly changed
the subject :-) See: Goldstein, Rebecca (2006), Incompleteness: The
Proof and Paradox of Kurt Godel (Norton).)

B.7   Position Paper #5: Can Computers Think?

B.7.1   Assignment

For this position paper, I would like you to evaluate the following hypothetical debate.

Pro: If something behaves in all relevant ways as if it were cognitive, then it *is* cognitive.

Con: What do you mean by "being cognitive"?

Pro: I mean that it:

- can perceive (see, hear, etc.);
- has beliefs, desires, and intentions;
- can remember;
- can use and understand natural language;
- can reason and make rational decisions; etc.

You know, the sort of thing that AI researchers are trying to achieve by computational means.

Con: Do you think they will succeed?

Pro: I'm optimistic: I think that a computer running a suitable AI program (or maybe a suite of programs) will eventually behave in all these ways.

Con: But that means that you think that such an AI-programmed computer *will be* cognitive?

Pro: Yes.

Con: But that's crazy! Computers and computer programs are purely syntactic!

Pro: Now it's my turn to ask for clarification: What do you mean by 'syntactic'?

Con: I mean that all a computer can do is to manipulate the symbols of a formal symbol system.[7]

Pro: So what's the problem?

Con: The problem is that cognition is semantic! That is, it involves the semantic interpretation of those symbols.

Pro: Well, I'm not so sure about that. But suppose you're right. What then?

Con: Well, syntax does not suffice for semantics. So, no computer executing a purely syntactic computer program can exhibit semantic cognition, even if it behaves in all relevant ways as if it were cognitive.

---

[7]http://www.cse.buffalo.edu/~rapaport/formalsystems—FIX THIS

- It may help if you rewrite Pro's and Con's arguments in terms of premises and conclusions, and then evaluate those arguments. That is, "extract" each argument from the debate and put them in the following forms:

1. Pro's premise 1	1. Con's premise 1
2. Pro's premise 2	2. Con's premise 2
3. (etc.)	(etc.)
4. Therefore, Pro's conclusion	4. Therefore, Con's conclusion

  Then analyze each argument.

- Keep in mind that premises and conclusions are declarative propositions (they can be deemed to be true or false) but that some lines uttered by Pro and Con are *not* declarative propositions (and thus can't be premises or conclusions). For example, Con's first statement is a question—it is not a premise or conclusion of anyone's argument—and Pro's second statment needs to be reformulated as something like "Something is cognitive means that it...".

1. For your peer-editing session next week, I will give you a choice: You may either:

   (a) create a "thinksheet" like the one for Position Paper #4 (§B.6.1.2)

       - with one column listing the premises, conclusions, and arguments;
       - one column of "cells" to indicate your agreement or disagreement with them;
       - and one column of "cells" to indicate your reasons for your agreement or disagreement

   (b) or write a 1–2 page, double-spaced (that is, about 250–500 word), single-sided, first draft.

   (Of course, you might want to do option 1a for your own use before doing option 1b! They are not mutually inconsistent.)

   If your document is more than 1 page long, please **staple** the pages together **and** make sure that your name is on all pages!

2. Please bring **5 copies** to class on the due date.

3. **At the top of the first page, please put ALL AND ONLY the following information** *in the following format***:**

Position Paper #5	**YOUR NAME**
DATE DUE	**YOUR CLASS**

4. Failure to correctly distinguish among **"true (or false) sentences, propositions, statements, premises, or conclusions"** and **"valid (or invalid) arguments"** will also result in a lower grade!

5. For general assistance with writing (including my preferred method of paper preparation and format, as well as advice on grammar), see my website "How to Write".
   http://www.cse.buffalo.edu/~rapaport/howtowrite.html

And don't forget to give full citations to any sources that you cite.

DUE AT THE BEGINNING OF LECTURE, ONE WEEK FROM TODAY

B.7.2   Suggestions and Guidelines for Peer-Group Editing

1. When you get into your small groups, introduce yourselves quickly, and share copies of your papers with each other.

2. Choose one paper to discuss first. (Suggestion: Go in alphabetical order by family name.)

3. After spending *about* 10–15 minutes on the first paper, move on to the next, going back to step 2 above, changing roles. Spend *no more than* 15 minutes per paper (because you've only got about 45 minutes at most). Perhaps one member of the group can be a timekeeper.

4. **Suggestion:** There are really 2 arguments in this dialogue: Pro's argument and Con's argument.

   So, the first task is to present each argument. Once you have identified the premises (including any hidden premises) and conclusion of each argument, you can then analyze it for validity of the argument and truth of the premises.

5. For each paper in your peer-editing group, ask as many of the following questions as you have time for:

   (a) Did the author present both Pro's and Con's arguments?

   (b) For each argument, did the author state whether and why s/he believes the argument to be valid?
   - It's possible to formulate both arguments so that they *are* valid!
   - If you do that, then ascertaining the truth value of the premises becomes your central task.

   (c) For each argument, did the author state whether and why s/he agrees with the premises?

   (d) For each argument, if the author believed either that the argument was invalid (even with missing premises added—that is, that there was *no* way to make the argument valid) or that one or more of the premises was false, then did the author state whether and why s/he agrees with the conclusion?
   - **Reminder:**

  i. If you think an argument is sound, then you are logically obligated to believe its conclusion (and you don't have to give any other justification for the conclusion).

  ii. If you don't believe the conclusion of an argument, then it is either invalid or else has at least one false premise; you must identify which, and explain why.

  iii. If you think an argument is unsound (either because it is invalid or has at least one false premise), then you might *still* believe the conclusion *for other reasons*; in that case, you must give those other reasons.

6. **Remember!**: Your revised paper must have the appropriate heading at the top of the first page, must use the terms 'true', 'false', 'valid', and 'invalid' appropriately, and must have your peer-edited first drafts attached!

7. Keep a written record of the questions and replies. This will be useful to the author, for revision.

8. At home, over the next week, please *revise* your paper to take into consideration the comments made by your fellow students (that is, your "peers"): Perhaps defend your claims better, or clarify statements that were misunderstood, etc. For help, see your instructor.

1–2 PAGE (250–500 WORD) REVISION, 1 COPY, TYPED, SINGLE-SIDED, DOUBLE-SPACED, IS DUE ONE WEEK FROM TODAY. NO LATE PAPERS WILL BE ACCEPTED!

B.7.3 Grading Rubric

```
Here's a draft of the grading rubric for Position Paper #5.

Because you need to spell out Pro's and Con's arguments in
premise-conclusion form, and because this may use up space,
it will not count against the word- and page-limits.

But if your paper is > 1 page, please STAPLE the pages together
(one staple, in upper left corner)
AND
please put your NAME on ALL pages.


========================================================================
Position Paper #5 Grading Rubric
Version: 20 Apr 2010
========================================================================


a)  Incorrect use of "true", "false", "valid", "invalid", "sound",
    "unsound", "argument", "premise", "conclusion", etc.:
```

```
-1 pt PER ERROR!

=========================================================================

b)  PRO'S ARGUMENT

b1)  List of premises & conclusion for Pro's argument:

3 = * clearly stated argument,
    * premises & conclusion clearly identified,
    * prems & conc clearly derived from dialogue

2 = neither clearly 3 nor 1,
    including:  not correctly identifying
some prem or conc.

1 = arg not clearly presented
    or not clearly derived from dialogue

0 = missing

-------------------------------------------------------------------------


b2)  Evaluation of validity of Pro's argument:

3 = valid XOR invalid
    + clear explanation why
      (including addition of any missing premises)
2 = valid XOR invalid, unclear explanation
1 = valid XOR invalid, no explanation
0 = no evaluation of validity

-------------------------------------------------------------------------

b3)  Evaluation of truth-value of Pro's premises:

Note:  Because each of you might have slightly different premises,
       I can't assign points to each one in any equally fair way,
       so I will grade you on your overall evaluation of the truth-
       values of the premises that you have explicitly identified.

6 = for EACH premise:
     truth-value clearly stated
     & good reasons given
```

```
4 = for SOME (but not all) premises:
      truth-value not stated
      OR no or weak reason given

2 = for MOST (UPDATED: or all) premises:
      truth-value not stated
      OR no or weak reasons given

0 = no evaluation of truth-values of premises

[Note: Because of my "quantum" scheme of grading,
       it is not possible to get 1,3, or 5 points!]


------------------------------------------------------------------------

b4)  Evaluation of truth-value of Pro's conclusion:

3 = if arg is sound,
then that is your reason for believing conc,
-- say so!
      else (if arg is not sound, then)
      say whether you believe conc
      & give clear reason why

2 = neither clearly 1 nor 3

1 = you think arg is not sound (which is fine),
      but you give no clear statement of truth-value of conc
      & no or weak reason given

0 = no evaluation of conclusion

========================================================================

c)  CON'S ARGUMENT (to be graded similarly, namely:)

c1)  List of premises & conclusion for Con's argument:

0,1,2,3 pts as above

c2)  Evaluation of validity of Con's argument:

0,1,2,3 pts as above

c3)  Evaluation of truth-value of Con's premises:
```

0,2,4,6 pts as above

c4) Evaluation of truth-value of Con's conclusion:

0,1,2,3 pts as above

==

The total is 30 points, which, following my grading theory, maps into
letter grades as follows:

letter CSE484 both CSE/PHI584

  A 29-30
  A-27-28
  B+ 26
  B 24-25
  B-22-23
  C+ 21
  C 17-20 11-20
  C-14-16
  D+ 11-13
  D 6-10
  F 0-5

On my "quantum-triage" grading scheme,

"A" means "understood the material for all practical purposes",
(here, that's 30 pts = 6 questions * 3 pts full credit
      + 2 questions * 6 pts full credit)

"B" has no direct interpretation,
    but comes about when averaging grades of A and C

"C" means "average",
(here, that's 20 pts = 6 * 2 pts partial credit
     + 2 * 4 pts partial credit)

"D" means "did not understand the material,
(here, that's 10 pts = 6 * 1 pt minimum credit
     + 2 * 2 pts minimum credit)

"F" usually means "did not do the work" (i.e., 0 pts),
    but can also come about when averaging Ds and Fs.

Please see my grading website,

`http://www.cse.buffalo.edu/~rapaport/howigrade.html`

for the theory behind all of this, which I'm happy to discuss.

B.8   Optional Position Paper: A Competition

Some of you have told us that you would like to come up with your own arguments instead of merely analyzing ones that we give you.

Here's your opportunity!

No later than two weeks from today, try your hand at creating an argument relevant to one of the topics of this course. It could be on a topic that we've already discussed, on a topic that we're going to discuss, or on some other topic in the philosophy of computer science (for ideas, take a look at the Further Readings at the ends of each chapter).

Your argument should have at least two premises and at least one conclusion. Try to make it a *valid* argument!

The "winner" (if there is one—we reserve the right to decide not to choose one) will have the honor of her or his argument being used as the argument to be analyzed for the next Position Paper. (To make it interesting and fair, for his or her position-paper assignment, the winner may be asked to *refute* the argument!)

You may submit the argument on paper (in lecture or by email). We also reserve the right to slightly modify the argument, if we feel that would make it more interesting.

Appendix C

Term Paper Topics

To the instructor: Discuss this at approximately 3 weeks into a 15-week term. Here is a list of some possible term-paper topics:

1. Further discussion of any topic covered in class, e.g.:

   (a) A critical examination of (someone else's) **published** answer to one of the questions listed on the syllabus.

   (b) **Your** answer to one of the questions listed on the syllabus, including a defense of your answer.

2. A critical examination of any of the required or recommended (or any other approved and relevant) readings.

3. A critical study of any monograph (i.e., single-topic book) or anthology (including special issues of journals) on the philosophy of computer science.

4. A critical, but general, survey article on the philosophy of computer science that would be appropriate for an encyclopedia of philosophy or an encyclopedia of computer science.

5. A presentation and well-argued defense of *your* "philosophy of computer science", that is, *your* answers to all (or most) of our questions, together with supporting reasons.

6. Other ideas of your own, approved by me in advance, including, but not limited to, such topics that we have mentioned but have not investigated as:

   - Is CS an empirical or a mathematical science?

   - Is CS an artificial or a natural science?

   - How does (the philosophy of CS) fit in with Alfred North Whitehead's "process philosophy"?

For general assistance with writing (including my required method of paper preparation and format, as well as advice on grammar), see my website "How to Write" http://www.cse.buffalo.edu/

For specific assistance on writing a philosophy paper, see:

- Mendelovici, Angela (2011), "A Sample Philosophy Paper", online at: http://tinyurl.com/SamplePhilPaper (accessed 13 February 2013)

- Wolff, Robert Paul (1975), "A Simple Foolproof Method for Writing Philosophy Papers", in his *About Philosophy* (Prentice-Hall); online at: http://www.amyscott.com/Philosophy%20Paper.pdf (accessed 13 February 2013)

- Or do an Internet search for "how to write a philosophy paper".

The paper should be a maximum of **10 double-spaced, single-sided pages** (that is, about 2500 words) (not counting the bibliography).

Deadlines:

1. **Two weeks from today: Proposal and reading list due.**

   - Your proposal and reading list must be approved by me before you begin your research and writing.

   - Because the term paper is optional, you do not need to commit yourself to it even if you turn in a proposal.

2. **Final paper will be due on the last day of the course.**

Appendix D

Final Exam

D.1 To the Instructor

At least one of the goals of philosophy education ought to be the fostering of the students' development of analytical and critical thinking skills. In order to explain the nature of the sample final exam to be presented in the next section, I want to call attention to a theory of cognitive development of college students, to discuss its implications for critical thinking, and to show how it can apply to the development of writing exercises such as this final exam.[1]

William G. Perry's scheme of cognitive development [Perry, 1970], [Perry, 1981] is a descriptive theory of stages that represent students' changing attitudes towards knowledge and values as they progress through their education. There are nine stages, which fall into four groups.[2]

I. Dualism Position 1. *Basic Duality:* Students at this stage believe that there are correct answers to all questions, that the instructor is the authority figure who has access to "golden tablets in the sky" containing all correct answers, and that their (the students') job is to *learn the correct answers* so that they can be repeated back to the instructor when asked. A Basic Duality student who offers a wrong answer to a question hears the authority figure say, "*You are wrong*".

Position 2. *Dualism:* Students move to this stage when faced with alternative opinions or disagreements among different authority figures (different instructors), such as when one literature teacher says that *Huckleberry Finn* is the best American novel, but another says that it is the worst. Dualistic students infer that one of those literature teachers' view of the golden tablets is obscured. Consequently, they see the purpose of education as *learning to find* the correct answers.

---

[1] This section is adapted from [Rapaport, 1984].

[2] My descriptions are culled from [Perry, 1981], [Cornfeld and Knefelkamp, 1979], and [Goldberger, 1979]. These are three essential readings for anyone concerned with implications and applications of Perry theory in the classroom.

Dualistic students prefer structured classes, which they see as providing the correct answers, and subjects such as math, which they see as having clear answers (all math teachers agree that the golden tablets say that $2 + 2 = 4$). Conflict between instructor and text, or between two instructors, is seen threateningly as conflicts among authority figures.

II. Multiplicity Position 3. *Early Multiplicity:* Here, the student has moved from the narrow Dualism of seeing all questions as having correct or else incorrect answers to a wider dualism of classifying questions into two kinds: those where instructors know the correct answers and those where they don't know the correct answers *yet*. Concerning the latter, the instructor's role is seen by Early Multiplistic students as providing *methods for finding* the correct answers, rather than as giving the correct answers directly.

Position 4. *Late Multiplicity:* As students move along, chronologically or cognitively, they begin to see the second kind of question as being the more common one. Because, it is felt, *most* questions are such that instructors don't have the correct answers for them, "everyone has a right to his own opinion; no one is wrong!" [Perry, 1981, 79]. The instructor's task, therefore, is seen as either teaching *how to think* or, worse, being irrelevant (after all, everyone has a right to their own opinion, including instructors— but it's *just their* opinion).

III. Contextual Relativism Position 5. Here, students have begun to see that instructors aren't always asking for correct answers but, rather, for *supported* answers. The second category of questions has become the only category (except, "of course", in mathematics and science!), but, although there can be many answers for each question, some are better (more adequate, more appropriate, etc.) than others. Answers are now seen as being better or worse *relative to* their supporting *context* (hence the name of this position).

IV. Commitment within Relativism Positions 6–9. These   stages characterize students as they begin to see the need for making their own decisions (making commitments, affirming values), as they balance their differing commitments, and as they realize the never-ending nature of this process.

Finally, there is evidence that a student as Position x will not understand—wil literally not be able to make any sense out of—instruction aimed at Position $x + 2$ (or beyond). Conversely, students at higher levels are bored by instruction aimed at lower levels.[3]

Here is a useful anecdote (adapted from Perry) for illustrating the scheme: Suppose that a teacher presents three theories for explaining the purpose of science (for example, is science merely descriptive, merely predictive, or explanatory?), or three different

---

[3]Perry's theory is far, far richer than I have portrayed it here, and the interested reader is urged to follow up the suggested readings. A useful survey of *criticisms*, together with a discussion of the relevance of the theory to mathematics education and to the history of matheamtics, is [Copes, 1980]. The relevance of the theory to philosophy is discussed in [Rapaport, 1982].

algorithms for solving some computational problem. The Dualistic student will wonder which is the correct one (and why the teacher bothered to talk about the incorrect ones); The Multiplistic student will think, "Only 3? Heck, I can come up with 6!"; the Contextual Relativist will wonder what advantages and disadvantages each theory has; and the Commitment-oriented student will be wondering about how to decide which is most appropriate or useful in a given situation.

Data from several studies indicate that most entering college freshmen are at Positions 2 or 3 [Perry, 1970], [Perry, 1981], [Copes, 1980]. *Courses designed to teach critical thinking skills to students through the first two years of college are thus dealing with Dualists or (early) Multiplists, and this can result in several problems that the instructor should be aware of in order to foster the students' development along the Perry scheme.*

First, Dualists want to be told the correct answers. But critical-thinking courses are largely involved with criticism and argument analysis. Accordingly, the entire activity may appear to them as incomprehensible at worst and pointless at best, or may simply result in the students learning the "sport" of "dumping" on "bad" or "incorrect" arguments. Hence, such courses, including the present one, must be more than mere criticism courses; they must make a serious attempt to teach ways of *constructing* arguments, *solving* problems, or *making decisions*. In this way, they can offer an appropriate "challenge" to Dualistic students, especially if couched in a context of adequate "support".[4]

Second, "The highly logical argument that, 'since everybody has a right to their own opinion, there is no basis for rational choice' is very typical of Multiplistic students" [Goldberger, 1979, 7]. But a goal of critical-thinking courses should be precisely to provide bases for rational choice: logical validity, inductive strength, etc. Accordingly, Multiplistic students either will not comprehend this goal or will view it as pointless. Again, such a course can be appropriately challenging to the students, but the instructor must be aware of how the students are likely to perceive it—to "hear" students' negative comments not as marks of pseudo-sophistication or worse, but as marks of viewing the world Multiplistically.

Finally, consider the concept of logical validity. Larry Copes (personal conversation) points out that it is a "relativistic" concept: A "valid" conclusion is one that is true *relative to* the truth of the premises. Dualistic students searching for absolutes and Multiplistic students feeling that "anything goes" may not accept, like, or understand validity. This may explain why so many students believ that arguments with true conclusions are valid or that valid arguments require true premises—even after having dutifully memorized the definition of 'validity'!

How can an instructor simultaneously *challenge* students in order to help them move to a higher-numbered stage, yet *not threaten* them, especially when a given class might have students at widely varying stages? One suggestion, based on work by Lee Knefelkamp, is to create assignments that can appeal to students at several levels.

The suggested final exam in the next section is one such assignment. Students are offered five questions and asked to answer any three of them. Question 1 is similar

---

[4]For details and specific advice, see [Sanford, 1967, Ch. 6, esp. pp. 51–52], [Cornfeld and Knefelkamp, 1979].

to, but not exactly the same as, the argument in Position Paper #1, so it is a *little bit* challenging to the Dualistic student, but is supportive in that they have already practiced giving a response to it. Question 2 is a brand-new argument for analysis, but one that students could have predicted that it would have appeared on a final exam, because it covers a topic that was explicitly discussed in the course yet was not the subject of a position paper; consequently, it is challenging, because it is new, but supportive, because it covers material that should be familiar—it should appeal to both Dualistic and Multiplistic students. Questions 3 and 4 appear to be open-ended questions that should appeal to Multiplistic students, though they can be understood as questions that might appeal to Dualistic students, too; after all, they are topics that were covered in the course, and the students have been given tools for evaluating answers to such questions—the challenge here is to construct arguments for answers to the questions. Here, the student's choice of which question (3 or 4) to answer (if any) is a function of their personal interests or familiarity with the issues. Question 5 is the most challenging, because the student must come up with a question and then answer it; it should appeal to Multiplistic as well as Contextual Relativistic students.

If left to their own choices, students will choose the least challenging question commensurate with their current stage. Thus, students need not be threatened by a question that they perceive as being to difficult or even unintelligible. But each question is just a bit more challenging than the previous one, and, because the students must answer three questions, they are offered a choice that includes at least one fully supportive question and at least one more-challenging question. (If such an exam is offered as a mid-term exam, then the final exam could begin with a least-challenging question that is *more* challenging than the least-challenging one on the mid-term.) In this way, students are encouraged to strive for a bit more, thus, hopefully, beginning the move to the next stage of cognitive development.

It is imperative for those of us who teach such courses to learn how to challenge our students appropriately in order to foster their intellectual "growth"; we must "hear" how our students inevitably make their *own* meaning out of what we say to them; and we must be ready to support them in the ego-threatening process of development.

D.2   A Final Exam

Do any 3 of the following. Write about 250–500 words for each answer.

This is a closed-book, closed-notes, closed-neighbor, open-mind exam.

No books, notebooks, food, beverages, or electronic devices of any kind are permitted in the exam room.

1. Evaluate the following argument (note that it is similar to, but *not* exactly the same as, the argument in Position Paper #1):

    > Natural science is the systematic observation, description, experimental investigation, and theoretical explanation of natural phenomena. Computer science is the study of computers and computing. Therefore, computer science is not a natural science.

2. Evaluate the following argument:

> Suppose that computers running certain computer programs can make rational decisions (at least in the sense of outputting values of functions that serve as a basis for decision making). That is, suppose that they can determine the validity of arguments and ascertain the probable truth-values of the premises of the arguments, and that they can consider the relative advantages and disadvantages of different courses of action, in order to determine the best possible choices. (For example, there are computers and computer programs that can diagnose certain diseases and (presumably) recommend appropriate medical treatments; there are computers and computer programs that can prove and verify proofs of mathematical theorems; and there are computers and computer programs that can play winning chess.) Suppose for the sake of argument that some of these computers and computer programs can make decisions (and recommendations) on certain important matters concerning human welfare. Suppose further that they can regularly make *better* recommendations than human experts on these matters. Therefore, these computers *should* make decisions on these important matters concerning human welfare.

3. What is computer science?

4. Can computers think?

5. Choose either 5a or 5b:

   (a) In your opinion, what is the most fundamental or important question in the philosophy of computer science?

   (b) What is a question that interests you in the philosophy of computer science that we did *not* discuss this semester?

Pose the question, explain why you think it is important or interesting, and present your answer to it.

D.3   NOTES FOR NEXT DRAFT

- On the Perry Scheme:
  See http://www.askphilosophers.org/question/5563

Bibliography

[Aaronson, 2008] Aaronson, S. (2008). The limits of quantum computers. *Scientific American*, pages 62–69. Accessed 30 June 2014 from:
http://www.cs.virginia.edu/~robins/The_Limits_of_Quantum_Computers.pdf.

[Aaronson, 2011] Aaronson, S. (2011). Quantum computing promises new insights, not just supermachines. *New York Times*, page D5. Accessed 16 July 2014 from:
http://www.nytimes.com/2011/12/06/science/scott-aaronson-quantum-computing-promises-new-insights.html.

[Aaronson, 2013] 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. http://www.scottaaronson.com/papers/philos.pdf (accessed 16 December 2013).

[Aaronson, 2014] Aaronson, S. (2014). Quantum randomness. *American Scientist*, 102(4):266–271. Accessed 8 September 2014 from:
http://www.americanscientist.org/issues/pub/quantum-randomness.

[Abrahams, 1987] Abrahams, P. (1987). What is computer science? *Communications of the ACM*, 30(6):472–472.

[Abramson, 2014] Abramson, D. (2014). The philosophical legacy of alan turing". http://www.thecritique.com/articles/the-philosophical-legacy-of-alan-turing/. (accessed 19 January 2015).

[Acocella, 2009] Acocella, J. (2009). Betrayal: Should we hate judas iscariot? *The New Yorker*, pages 68–73.

[Adami, 2012] Adami, C. (2012). Boldly going beyond mathematics. *Science*, 338:1421–1422. review of [Holland, 2012].

[Adleman, 1998] Adleman, L. M. (1998). Computing with dna. *Scientific American*, pages 54–61. http://www.usc.edu/dept/molecular-science/papers/fp-sciam98.pdf (accessed 9 December 2013).

[Aizawa, 2010] Aizawa, K. (2010). Computation in cognitive science: It is not all about Turing-equivalent computation. *Studies in History and Philosophy of Science*, 41(3):227–236. Reply to [Piccinini, 2004a].

[Allen, 2001] Allen, L. G. (2001). Teaching mathematical induction: An alternative approach. *Mathematics Teacher*, 94(6):500–504.

[Allen, 1989] Allen, S. (1989). *Meeting of Minds*. Prometheus Books.

[Anderson and Belnap, 1975] Anderson, A. R. and Belnap, Jr., N. D., editors (1975). *Entailment: The Logic of Relevance and Necessity*, volume I. Princeton University Press, Princeton, NJ.

[Anderson et al., 1975] Anderson, A. R., Belnap Jr., N. D., and Dunn, J. M., editors (1975). *Entailment: The Logic of Relevance and Necessity*, volume II. Princeton University Press, Princeton, NJ.

[Angier, 2010] Angier, N. (2010). Abtract thoughts? the body takes them literally. *New York Times/Science Times*, page D2. Accessed 31 January 2014 from: http://www.nytimes.com/2010/02/02/science/02angier.html.

[Anonymous, 1853] Anonymous (1853). Review of j.h. wythes, *the microscopist; or a complete manual on the use of the microscope*. *Quarterly Journal of Microscopical Science*, 1(1):51–53.

[Anthes, 2006] Anthes, G. (1 May 2006). Computer science looks for a remake. In *Computerworld*, accessed 2 July 2013 from http://www.computerworld.com/s/article/110959/Computer_Science_Looks_for_a_Remake.

[Antoy and Hanus, 2010] Antoy, S. and Hanus, M. (2010). Functional logic programming. *Communications of the ACM*, 53(4):74–85.

[Apostel, 1961] Apostel, L. (1961). Towards the formal study of models in the non-formal sciences. In Freudenthal, H., editor, *The Concept and the Role of the Model in Mathematics and Natural and Social Sciences: Proceedings of the Colloquium Sponsored by the Division of Philosophy of Sciences of the International Union of History and Philosophy of Sciences Organized at Utrecht, January 1960*, pages 1–37. D. Reidel, Dordrecht, Holland.

[Appiah, 2007] Appiah, K. A. (2007). The new new philosophy. *New York Times Magazine*, pages 34–36.

[Appiah, 2008] Appiah, K. A. (2008). Experimental philosophy. *Proceedings and Addresses of the American Philosophical Association*, 82(2):7–22. https://member.apaonline.org/V82_2_experimentalphilosophy.aspx.

[Ardis et al., 1989] Ardis, M., Basili, V., Gerhart, S., Good, D., Gries, D., Kemmerer, R., Leveson, N., Musser, D., Neumann, P., and von Henke, F. (1989). Editorial process verification. *Communications of the ACM*, 32(3):287–290. "ACM Forum" letter to the editor, with replies by James H. Fetzer and Peter J. Denning.

[Aspray, 1990] Aspray, W., editor (1990). *Computing before Computers*. Iowa State University Press, Ames, IA. online at [http://ed-thelen.org/comp-hist/CBC.html].

[Austin, 1983] Austin, A. K. (1983). An elementary approach to *np*-completeness. *American Mathematical Monthly*, 90:398–399.

[Baars, 1997] Baars, B. J. (1997). Contrastive phenomenology: A thoroughly empirical approach to consciousness. In Block, N., Flanagan, O., and Güzeldere, G., editors, *The Nature of Consciousness: Philosophical Debates*, pages 187–201. MIT Press, Cambridge, MA.

[Bader, 2013] Bader, R. M. (2013). Towards a hyperintensional theory of intrinsicality. *Journal of Philosophy*, 110(10):525–563. Preprint accessed 13 February 2014 from http://users.ox.ac.uk/~sfop0426/Intrinsicality%20(R.%20Bader).pdf.

[Bajcsy et al., 1992] Bajcsy, R. K., Borodin, A. B., Liskov, B. H., and Ullman, J. D. (1992). Computer science statewide review draft preface. Technical report, Computer Science Rating Committee. Confidential Report and Recommendations to the Commissioner of Education of the State of New York (unpublished), http://www.cse.buffalo.edu/~rapaport/Papers/Papers.by.Others/bajcsyetal92.pdf, accessed 30 September 2011.

[Ballard, 1997] Ballard, D. H. (1997). *An Introduction to Natural Computation*. MIT Press, Cambridge, MA.

[Barr, 1985] Barr, A. (1985). Systems that know that they don't understand. http://www.stanford.edu/group/scip/avsgt/cognitiva85.pdf (accessed 21 July 2013).

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

[Baum, 1900] Baum, L. F. (1900). *The Wizard of Oz*. PUB, CITY.

[Beebe, 2011] Beebe, J. R. (2011). Experimental epistemology research group. http://eerg.buffalo.edu/. Accessed 9 September 2011.

[Benenson, 2013] Benenson, Y. (2013). Recombinatorial logic. *Science*, 340(6132):554–555. Accessed 31 October 2014 from: http://www.rle.mit.edu/sbg/wp-content/uploads/2013/05/Science-2013-Benenson-554-5.pdf.

[Berners-Lee et al., 2006] Berners-Lee, T., Hall, W., Hendler, J., Shadbolt, N., and Weitzner, D. J. (2006). Creating a science of the web. *Science*, 313:769–771. 10.1126/science.1126902.

[Blessing, 2013] Blessing, K. (2013). I re-read, therefore i understand. *Philosophy Now*, 94:17. Accessed 20 February 2013 from http://philosophynow.org/issues/94/I_Re-Read_Therefore_I_Understand.

[Boden, 1977] Boden, M. A. (1977). *Artificial Intelligence and Natural Man*. Basic Books, New York.

[Boden, 2006] Boden, M. A. (2006). *Mind as Machine: A History of Cognitive Science*. Oxford University Press, Oxford.

[Boey, 2014] Boey, G. (2014). On my interview with peter unger, and the value of philosophy. *3 Quarks Daily*. Accessed 2 July from: http://www.3quarksdaily.com/3quarksdaily/2014/06/on-my-interview-with-peter-unger-and-the-value-of-philosophy.html.

[Böhm and Jacopini, 1966] Böhm, C. and Jacopini, G. (1966). Flow diagrams, turing machines and languages with only two formation rules. *Communications of the ACM*, 9(5):366–371. Accessed 19 November 2013 from: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.119.9119&represent=rep1&type=pdf.

[Boole, 2009] Boole, G. (2009). *An Investigation of the Laws of Thought: On Which Are Founded the Mathematical Theories of Logic and Probabilities*. Cambridge University Press, Cambridge, UK.

[Boolos and Jeffrey, 1974] Boolos, G. S. and Jeffrey, R. C. (1974). *Computability and Logic*. Cambridge University Press, Cambridge, UK.

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

[Borbely, 2005] Borbely, R. (2005). Letter to the editor. *Scientific American*, pages 12, 15.

[Borges, 1981] Borges, J. L. (1981). Partia enchantments of the quixote. In Monegal, E. R. and Reid, A., editors, *Borges, A Reader; A Selection from the Writings of Jorge Luis Borges*, pages 232–235. E.P. Dutton, New York.

[Bostrom, 2003] Bostrom, N. (2003). Are you living in a computer simulation? *Philosophical Quarterly*, 53(211):243–255. Available online, along with many related items, at http://www.simulation-argument.com/.

[Bowie, 1973] Bowie, G. L. (1973). An argument against church's thesis. *Journal of Philosophy*, 70(3):66–76.

[Brenner, 2012] Brenner, S. (2012). The revolution in the life sciences. *Science*, 338:1427–1428.

[Bringsjord, 2006] Bringsjord, S. (11 August 2006). unpublished ms. . http://www.cse.buffalo.edu/∼rapaport/Papers/Papers.by.Others/bringsjord06-constraints_on_philofcompsci.pdf.

[Bringsjord and Arkoudas, 2004] Bringsjord, S. and Arkoudas, K. (2004). The modal argumenbt for hypercomputing minds. *Theoretical Computer Science*, 317:167–190. doi:10.1016/j.tcs.2003.12.010; accessed 5 February 2014 from http://kryten.mm.rpi.edu/modal.hypercomputing.pdf.

[Brooks, 1996] Brooks, Jr., F. P. (1996). The computer scientist as toolsmith ii. *Communications of the ACM*, 39(3):61–68. Revised and extended verson of Brooks Jr., Frederick P. (1977), "The Computer Scientist as Toolsmith—Studies in Interactive Computer Graphics", in B. Gilchrist (ed.), *Information Processing 77,*

Proceedings of IFIP Congress 77 (Toronto) (Amsterdam: North-Holland): 625–634; accessed 6 June 2013 from http://www.cs.unc.edu/techreports/88-041.pdf.

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

[Brooks, 1991] Brooks, R. A. (1991). Intelligence without representation. *Artificial Intelligence*, 47:139–159.

[Brown, 1992] Brown, R. (4 August 1992). contribution to newsgroup discussion of cyc. Article 13305 of comp.ai. http://www.cse.buffalo.edu/~rapaport/definitions.of.ai.html.

[Buchanan, 2006] Buchanan, B. G. (2006). What do we know about knowledge? *AI Magazine*, pages 35–46.

[Bundy, 1983] Bundy, A. (1983). The nature of ai: A reply to schank. *AI Magazine*, 4(4):29–31.

[Bunge, 1974] Bunge, M. (1974). Toward a philosophy of technology. In Michalos, A. C., editor, *Philosophical Problems of Science and Technology*, pages 28–47. Allyn & Bacon, Boston.

[Burks, 1970] Burks, A. W., editor (1970). *Essays on Cellular Automata*. University of Illinois Press, Urbana, IL.

[Buxton, 2012] Buxton, W. A. (2012). Innovative interaction: From concept to the wild. *Communications of the ACM*, 55(9):90.

[Campos, 2011] Campos, D. G. (2011). On the distinction between peirce's abduction and lipton's inference to the best explanation. *Synthese*, 180:419–442.

[Cane, 2014] Cane, S. (2014). Interview: David chalmers and andy clark. *New Philosopher*, 2:mind. Accessed 13 March 2014 from: http://www.newphilosopher.com/articles/interview-david-chalmers-and-andy-clark/.

[Cannon, 2013] Cannon, P. (2013). Kant at the bar: Transcendental idealism in daily life. *Philosophy Now*, Issue 95:15–17. Accessed 15 November 2013 from http://philosophynow.org/issues/95/Kant_at_the_Bar_Transcendental_Idealism_in_Daily_Life.

[Cardoso, 1997] Cardoso, S. H. (1997). Specialized functions of the cerebral cortex. http://www.cerebromente.org.br/n01/arquitet/cortex_i.htm.

[Carhart, 1956] Carhart, R. R. (1956). The systems approach to reliability. In *Proceedings, 2nd National Symposium on Quality Control and Reliabillity in Electronics, January 9–10*, pages 149–155. IRE Professional Group on Reliability and Quality Control, American Society for Quality Control, Electronics Technical Committee, Institute of Radio Engineers, Washingtoe, DC.

[Carleton, 1984] Carleton, L. R. (1984). Programs, language understanding, and searle. *Synthese*, 59:219–230.

[Carlson et al., 1996]
Carlson, B., Burgess, A., and Miller, C. (1996). Timeline of computing history. http://www.computer.org/cms/Computer.org/Publications/timeline.pdf.

[Carnap, 1956] Carnap, R. (1956). *Meaning and Necessity: A Study in Semantics and Modal Logic, Second Edition*. University of Chicago Press, Chicago.

[Carpenter and Doran, 1977] Carpenter, B. and Doran, R. (1977). The other turing machine. *The Computer Journal*, 20(3):269–279. Accessed 6 November 2013 from http://comjnl.oxfordjournals.org/content/20/3/269.full.pdf+html.

[Carroll, 1871] Carroll, L. (1871). Through the looking-glass. online at: http://www.gutenberg.org/files/12/12-h/12-h.htm. (accessed 20 July 2013).

[Carroll, 1893] Carroll, L. (1893). *Sylvie and Bruno Concluded*. Macmillan, London.

[Carroll, 1895] Carroll, L. (1895). What the tortoise said to achilles. *Mind*, 4(14):278–280. online at http://www.ditext.com/carroll/tortoise.html, accessed 7 January 2013.

[Carroll, 1848] Carroll, L. (ca. 1848). The two clocks. In *The Rectory Umbrella*. (unknown). Accessed 4 February 2014 from: http://etc.usf.edu/lit2go/112/poems-puzzles-and-stories-of-lewis-carroll/4953/the-two-clocks/.

[Cathcart and Klein, 2007] Cathcart, T. and Klein, D. (2007). *Plato and a Platypus Walk into a Bar: Understanding Philosophy through Jokes*. Abrams Image, New York.

[Cerf, 2012] Cerf, V. G. (2012). Computer science revisited. *Communications of the ACM*, 55(12):7.

[Cerf, 2015] Cerf, V. G. (2015). There is nothing new under the sun. *Communications of the ACM*, 58(2):7.

[Ceruzzi, 1989] Ceruzzi, P. (1989). Electronics technology and computer science, 1940–1975: A coevolution. *Annals of the History of Computing*, 10(4):257–275.

[Chaitin, 1987] Chaitin, G. J. (1987). Computing the busy beaver function. In Cover, T. and Gopinath, B., editors, *Open Problems in Communication and Computation*, pages 108–112. Springer. http://www.cs.auckland.ac.nz/∼chaitin/bellcom.pdf (accessed 16 December 2013).

[Chalmers, 1995] Chalmers, D. J. (1995). On implementing a computation. *Minds and Machines*, 4:391–402.

[Chalmers, 1996] Chalmers, D. J. (1996). Does a rock implement every finite-state automaton? *Synthese*, 108:309–333. 1993 preprint accessed 23 May 2012 from http://consc.net/papers/rock.html; Chalmers corrects "an error in my arguments" in this paper in [Chalmers, 2012, 236–238].

[Chalmers, 2011] Chalmers, D. J. (2011). A computational foundation for the study of cognition. *Journal of Cognitive Science (South Korea)*, 12(4):323–357. Originally written in 1993; slightly revised version published 2011; accessed 3 November 2014 from:
http://j-cs.org/gnuboard/bbs/board.php?bo_table=__vol012i4&wr_id=2.

[Chalmers, 2012] Chalmers, D. J. (2012). The varieties of computation: A reply. *Journal of Cognitive Science (South Korea)*, 13(3):211–248. http://j-cs.org/gnuboard/bbs/board.php?bo_table=__vol013i3&wr_id=1
(accessed 9 November 2014).

[Chase, 1980] Chase, G. C. (1980). History of mechanical computing machinery. *Annals of the History of Computing*, 2(3):198–226.

[Chater and Oaksford, 2013] Chater, N. and Oaksford, M. (2013). Programs as causal models: Speculations on mental programs and mental representation. *Cognitive Science*, 37(6):1171–1191.

[Chetty, 2013] Chetty, R. (2013). Yes, economics is a science. *New York Times*, page A21. online at http://www.nytimes.com/2013/10/21/opinion/yes-economics-is-a-science.html (accessed 4 November 2013).

[Chisum, 1986] Chisum, D. S. (1985-1986). The patentability of algorithms. *University of Pittsburgh Law Review*, 47:959–1022.

[Chopra, 2014] Chopra, S. (2014). Computer programs are people, too. http://www.thenation.com/article/180047/computer-programs-are-people-too. Accessed 13 June 2014.

[Church, 1936a] Church, A. (1936a). A note on the entscheidungsproblem. *Journal of Symbolic Logic*, 1(1):40–41. See also "Correction to *A Note on the Entscheidungsproblem*", *Journal of Symbolic Logic* 1((3) (September): 101–102.

[Church, 1936b] Church, A. (1936b). An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363. Accessed 4 February 2014 from: http://phil415.pbworks.com/f/Church.pdf.

[Church, 1937] Church, A. (1937). Review of [Turing, 1936]. *Journal of Symbolic Logic'*, 2(1):42–43. For commentary on this review, see [Hodges, 2013].

[Church, 1956] Church, A. (1956). *Introduction to Mathematical Logic*. Princeton University Press, Princeton, NJ.

[Churchland and Sejnowski, 1992] Churchland, P. S. and Sejnowski, T. J. (1992). *The Computational Brain*. MIT Press, Cambridge, MA.

[Clarke, 1953] Clarke, A. C. (1953). *Childhood's End*. PUB, CITY.

[Cleland, 1993] Cleland, C. E. (1993). Is the Church-Turing thesis true? *Minds and Machines*, 3(3):283–312.

[Cleland, 1995] Cleland, C. E. (1995). Effective procedures and computable functions. *Minds and Machines*, 5(1):9–23.

[Cleland, 2001] Cleland, C. E. (2001). Recipes, algorithms, and programs. *Minds and Machines*, 11(2):219–237.

[Cleland, 2002] Cleland, C. E. (2002). On effective procedures. *Minds and Machines*, 12(2):159–179.

[Cleland, 2004] Cleland, C. E. (2004). The concept of computability. *Theoretical Computer Science*, 317:209–225.

[Cockshott and Michaelson, 2007] Cockshott, P. and Michaelson, G. (2007). Are there new models of computation? reply to wegner and eberbach. *The Computer Journal*, 50(2):232–247. Preprint accessed 30 June 2014 from: http://www.dcs.gla.ac.uk/∼wpc/reports/wegner25aug.pdf.

[Coffa, 1991] Coffa, J. (1991). *The Semantic Tradition from Kant to Carnap: To the Vienna Station*. Cambridge University Press, Cambridge, UK.

[Cohen), 1919] Cohen), P. M. R. (1919). A slacker's apology. *The New Republic*, pages 19–20.

[Colburn and Shute, 2011] Colburn, T. and Shute, G. (2011). Decoupling as a fundamental value of computer science. *Minds and Machines*, 21(2):241–259. Accessed 23 July 2014 from: http://www.d.umn.edu/∼tcolburn/papers/Decoupling.pdf.

[Colburn, 1999] Colburn, T. R. (1999). Software, abstraction, and ontology. *The Monist*, 82(1):3–19.

[Colburn, 2000] Colburn, T. R. (2000). *Philosophy and Computer Science*. M.E. Sharpe, Armonk, NY.

[Colburn et al., 1993] Colburn, T. R., Fetzer, J. H., and Rankin, T. L., editors (1993). *Program Verification: Fundamental Issues in Computer Science*. Kluwer Academic Publishers, Dordrecht, The Netherlands.

[Cooper, 2012] Cooper, S. B. (2012). Turing's titanic machine? *Communications of the ACM*, 55(3):74–83. http://cacm.acm.org/magazines/2012/3/146259-turings-titanic-machine/fulltext (accessed 26 February 2014).

[Copeland, 1996] Copeland, B. J. (1996). What is computation? *Synthese*, 108:335–359. Preprint (accessed 18 March 2014) at http://www.alanturing.net/turing_archive/pages/pub/what/what.pdf.

[Copeland, 1997] Copeland, B. J. (1997). The broad conception of computation. *American Behavioral Scientist*, 40(6):690–716.

[Copeland, 1998] Copeland, B. J. (1998). Even turing machines can compute uncomputable functions. In Calude, C., Casti, J., and Dinneen, M. J., editors, *Unconventional Models of Computtion*, pages 150–164. Springer-Verlag. Preprint accessed 4 June 2014 from:
http://www.alanturing.net/turing_archive/pages/pub/even/even.pdf.

[Copeland, 2000] Copeland, B. J. (2000). The church-turing thesis. http://www.alanturing.net/turing_archive/pages/Reference%20Articles/The%20Turing-Church%20Thesis.html. Accessed 24 September 2014.

[Copeland, 2002] Copeland, B. J. (2002). Hypercomputation. *Minds and Machines*, 12(4):461–502.

[Copeland and Proudfoot, 2010] Copeland, B. J. and Proudfoot, D. (2010). Deviant encodings and turing's analysis of computability. *Studies in History and Philosophy of Science*, 41(3):247–252.

[Copeland and Shagrir, 2011] Copeland, B. J. and Shagrir, O. (2011). Do accelerating Turing machines compute the uncomputable? *Minds and Machines*, 21(2):221–239.

[Copeland and Sylvan, 1999] Copeland, B. J. and Sylvan, R. (1999). Beyond the universal turing machine. *Australasian Journal of Philosophy*, 77(1):46–67. Preprint accessed 27 May 2014 from:
http://www.alanturing.net/turing_archive/pages/pub/beyond/beyond.pdf.

[Copeland and Sylvan, 2000] Copeland, B. J. and Sylvan, R. (2000). Computability is logic-relative. In Hyde, D. and Priest, G., editors, *Sociative Logics and Their Applications: Essays by the Late Richard Sylvan*, pages 189–199. Ashgate.

[Copeland, 1999] Copeland, B. J. e. (1999). The turing-wilkinson lecture series on the automatic computing engine. In Furukawa, K., Michi, D., and Muggleton, S., editors, *Machine Intelligence 15: Intelligent Agents*, pages 381–444. Oxford University Press, Oxford.

[Copes, 1980] Copes, L. (1980). College teaching, mathematical knowledge, and the perry development scheme. (St. Paul, MN: Institute for Studies in Educational Mathematics).

[Cornfeld and Knefelkamp, 1979] Cornfeld, J. and Knefelkamp, L. (1979). Combining student stage and type in the design of learning environments: An integration of perry stages and holland typologies. Paper presented at the American College Personnel Association, Los Angeles, March.

[Covert, 2014] Covert, M. W. (2014). Simulating a living cell. *Scientific American*, 310(1):44–51.

[Crossley and Henry, 1990] Crossley, J. N. and Henry, A. S. (1990). Thus spake al-k̲hwārizmī: A translation of the text of cambridge university library ms. ii.vi.5. *Historia Mathematica*, 17:103–131.

[Crowcroft, 2005] Crowcroft, J. (2005). On the nature of computing. *Communications of the ACM*, 48(2):19–20.

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

[David, 2009] David, M. (2009). The correspondence theory of truth. In Zalta, E. N., editor, *Stanford Encyclopedia of Philosophy (Fall 2009 Edition)*. http://plato.stanford.edu/archives/fall2009/entries/truth-correspondence/.

[Davidson, 2006] Davidson, J. (2006). Measure for measure: Exploring the mysteries of conducting. *The New Yorker*, pages 60–69.

[Davies, 1999] Davies, D. W. (1999). Repairs to turing's universal computing machine. In Furukawa, K., Michie, D., and Muggleton, S., editors, *Machine Intelligence 15: Intelligent Agents*, pages 477–488. Oxford University Press, Oxford. http://tinyurl.com/lqero7l (accessed 21 January 2014).

[Davis, 1995a] Davis, M. (1995a). An historical preface to engineering ethics. *Science and Engineering Ethics*, 1(1):33–48.

[Davis, 1995b] Davis, M. (1995b). Mathematical logic and the origin of modern computers. In Herken, R., editor, *The Universal Turing Machine: A Half-Century Survey, Second Edition*, pages 135–158. Springer-Verlag, Vienna.

[Davis, 1996] Davis, M. (1996). Defining 'engineer:' how to do it and why it matters. *Journal of Engineering Education*, 85(2):97–101.

[Davis, 1998] Davis, M. (1998). *Thinking Like an Engineer: Studies in the Ethics of a Profession*. Oxford University Press, New York.

[Davis, 2000] Davis, M. (2000). Overheard in the park. *American Scientist*, 88:366–367. Accessed 19 March 2014 from http://www.americanscientist.org/bookshelf/pub/overheard-in-the-park.

[Davis, 2004] Davis, M. (2004). The myth of hypercomputation. In Teuscher, C., editor, *Alan Turing: The Life and Legacy of a Great Thinker*, pages 195–202. Springer, Berlin.

[Davis, 2006a] Davis, M. (2006a). The church-turing thesis: Consensus and opposition. In Beckmann, A., Berger, U., Löwe, B., and Tucker, J., editors, *Logical Approaches to Computational Barriers: Second Conference on Computability in Europe, CiE 2006, Swansea, UK, June 30–July 5*, pages 125–132. Springer-Verlag Lecture Notes in Computer Science 3988, Berlin.

[Davis, 2006b] Davis, M. (2006b). What is Turing reducibility? *Notices of the AMS*, 53(10):1218–1219. http://www.ams.org/notices/200610/whatis-davis.pdf (accessed 30 May 2014).

[Davis, 2006c] Davis, M. (2006c). Why there is no such discipline as hypercomputation. *Applied Mathematics and Computation*, 178:4–7.

[Davis, 2011] Davis, M. (2011). Will software engineering ever be engineering? *Communications of the ACM*, 54(11):32–34.

[Davis, 2012] Davis, M. (2012). *The Universal Computer: The Road from Leibniz to Turing; Turing Centenary Edition*. CRC Press/Taylor & Francis Group, Boca Raton, FL. also published as *Engines of Logic: Mathematicians and the Origin of the Computer* (New York: W.W. Norton, 2000).

[Davis and Weyuker, 1983] Davis, M. D. and Weyuker, E. J. (1983). *Computability, Complexity and Langauges*. Academic Press, New York.

[Davis et al., 1996] 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. Accessed 23 September 2014 from: people.csail.mit.edu/davis/cacm96.ps. A summary version of [Samuelson et al., 1994].

[Davis, 1977] Davis, R. M. (1977). Evolution of computers and computing. *Science*, 195:1096–1102.

[Daylight, 2013] Daylight, E. G. (2013). Towards a historical notion of "Turing— the father of computer science". http://www.dijkstrascry.com/sites/default/files/papers/Daylightpaper91.pdf. (accessed 7 April 2015).

[De Millo et al., 1979] De Millo, R. A., Lipton, R. J., and Perlis, A. J. (1979). Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271–280.

[Dedekind, 1890] Dedekind, R. (1890). Letter to keferstein. In van Heijenoort, J., editor, *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*, pages 98–103. Harvard University Press, Cambridge, MA.

[DeJohn and Dietrich, 2003] DeJohn, J. and Dietrich, E. (2003). Editorial: Subvert the dominant paradigm! a review of *computationalism: new directions*, edited by Matthias Scheutz. *Journal of Experimental and Theoretical Artificial Intelligence*, 15(4):375–382.

[Dennett, 2009] Dennett, D. (2009). Darwin's 'strange inversion of reasoning'. *Proceedings of the National Academy of Science*, 106, suppl. 1:10061–10065. http://www.pnas.org/cgi/doi/10.1073/pnas.0904433106. See also [Dennett, 2013].

[Dennett, 2013] Dennett, D. (2013). Turing's 'strange inversion of reasoning'. In Cooper, S. B. and van Leeuwen, J., editors, *Alan Turing: His Work and Impact*, pages 569–573. Elsevier, Amsterdam. See also [Dennett, 2009].

[Dennett, 1971] Dennett, D. C. (1971). Intentional systems. *Journal of Philosophy*, 68:87–106. Reprinted in Daniel C. Dennett, *Brainstorms* (Montgomery, VT: Bradford Books): 3–22.

[Dennett, 1982] Dennett, D. C. (1982). Notes on prosthetic imagination. *Boston Review*, 7(3):3–7.

[Dennett, 2012] Dennett, D. C. (2012). Sakes and dints: And other definitions that philosophers really need not seek. *Times Literary Supplement*, pages 12–14. http://ase.tufts.edu/cogstud/papers/TLS2012.pdf (accessed 21 November 2013).

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

[Denning, 1999] Denning, P. J. (1999). Computer science: The discipline. In Ralston, A., Reilly, E. D., and Hemmendinger, D., editors, *Encyclopedia of Computer Science, Fourth Edition*. Grove's Dictionaries, 2000, New York. preprint (dated 1999) available online at:
http://cs.gmu.edu/cne/pjd/PUBS/ENC/cs99.pdf (accessed 11 June 2013); page references are to the preprint.

[Denning, 2003] Denning, P. J. (2003). Great principles of computing. *Communications of the ACM*, 46(11):15–20.

[Denning, 2005] Denning, P. J. (2005). Is computer science science? *Communications of the ACM*, 48(4):27–31.

[Denning, 2013] Denning, P. J. (2013). The science in computer science. *Communications of the ACM*, 56(5):35–38.

[Denning et al., 1989] 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 and Freeman, 2009] Denning, P. J. and Freeman, P. A. (2009). Computing's paradigm. *Communications of the ACM*, 52(12):28–30.

[Dershowitz and Gurevich, 2008] Dershowitz, N. and Gurevich, Y. (2008). A natural axiomatization of computability and proof of church's thesis. *Bulletin of Symbolic Logic*, 14(3):299–350. Accessed 30 June 2014 from:
http://research.microsoft.com/pubs/70459/tr-2007-85.pdf.

[Deutsch, 1985] Deutsch, D. (1985). Quantum theory, the church-turing principle and the universal computer. *Proceedings of the Royal Society of London A*, 400:97–117. Preprint accessed 8 April 2014 from:
http://www.cs.berkeley.edu/∼christos/classics/Deutsch_quantum_theory.pdf;page references are to the preprint.

[Deutsch, 2009] Deutsch, M. (2009). Experimental philosophy and the theory of reference. *Mind & Language*, 24(4):445–466.

[Dewdney, 1989] Dewdney, A. (1989). *The Turing Omnibus: 61 Excursions in Computer Science*. Computer Science Press, Rockville, MD. esp. Chs. 1 ("Algorithms: Cooking Up Programs"), 28 ("Turing Machines: The Simplest Computers"), and 48 ("Universal Turing Machines: Computers as Programs").

[Dewey, 1993] Dewey, J. (1993). *How We Think: A Restatement of Reflective Thinking to the Educative Process, revised ed.* Heath, Boston.

[Dietrich, 2001] Dietrich, E. (2001). Homo sapiens 2.0: Why we should build the better robots of our nature. *Journal of Experimental and Theoretical Artificial Intelligence*, 13(4):323–328.

[Dietrich, 2007] Dietrich, E. (2007). After the humans are gone. *Journal of Experimental and Theoretical Artificial Intelligence*, 19(1):55–67.

[Dijkstra, 1968] Dijkstra, E. W. (1968). Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148. Preprint accessed 27 July 2014 from: https://www.cs.utexas.edu/∼EWD/transcriptions/EWD03xx/EWD361.html.

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

[Dijkstra, 1975] 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. Accessed 19 April 2013 from http://www.cs.utexas.edu/∼EWD/ewd05xx/EWD512.PDF.

[Dijkstra, 1976] 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. online at http://www.cs.utexas.edu/∼EWD/transcriptions/EWD06xx/EWD611.html (accessed 19 April 2013).

[Dijkstra, 1986] Dijkstra, E. W. (1986). Mathematicians and computing scientists: The cultural gap. *Mathematical Intelligencer*, 8(1):48–52.

[Dipert, 1993] Dipert, R. R. (1993). *Artifacts, Art Works, and Agency*. Temple University Press, Philadelphia.

[Doxiadis et al., 2009] Doxiadis, A., Papadimitriou, C. H., Papadatos, A., and Di Donna, A. (2009). *Logicomix: An Epic Search for Truth*. Bloomsbury USA, New York.

[Dresner, 2003] Dresner, E. (2003). Effective memory and Turing's model of mind. *Journal of Experimental & Theoretical Artificial Intelligence*, 15(1):113–123.

[Dresner, 2012] Dresner, E. (2012). Turing, Matthews and Millikan: Effective memory, dispositionalism and pushmepullyou states. *International Journal of Philosophical Studies*, 20(4):461–472.

[Dublon and Paradiso, 2014] Dublon, G. and Paradiso, J. A. (2014). Extra sensory perception. *Scientific American*, 311(1):36–41.

[Dumas, 1844] Dumas, A. p. (1844). *The Count of Monte Cristo*. Penguin Books, 2003, London.

[Dunning, 2007] Dunning, B. (16 May 2007). The importance of teaching critical thinking. http://skeptoid.com/episodes/4045 (accessed 13 February 2013).

[Dyson, 2004] Dyson, F. (2004). The world on a string. *New York Review of Books*, pages 16–19.

[Dyson, 2011a] Dyson, F. (2011a). The case for far-out possibilities. *New York Review of Books*, pages 26–27.

[Dyson, 2011b] Dyson, F. (2011b). How we know. *New York Review of Books*, pages 8, 10, 12.

[Dyson, 2012a] Dyson, G. (2012a). Turing centenary: The dawn of computing. *Nature*, 482(7386):459–460. doi:10.1038/482459a.

[Dyson, 2012b] Dyson, G. (2012b). *Turing's Cathedral*. PUB, CITY.

[Easton, 2006] Easton, T. A. (2006). Beyond the algorithmization of the sciences. *Communications of the ACM*, 49(5):31–33.

[Eco, 1982] Eco, U. (1982). On the impossibility of drawing a map of the empire on a scale of 1 to 1. In *Hot to Travel with a Salmon and Other Essays*, pages 95–106. Harcourt Brace, New York.

[Edelman, 2008] Edelman, S. (2008). *Computing the Mind*. Oxford University Press, New York.

[Eden, 2007] Eden, A. H. (2007). Three paradigms of computer science. *Minds and Machines*, 17(2):135–167. online at:
http://www.ic.unicamp.br/~wainer/cursos/2s2006/epistemico/filosofia-cs.pdf
(accessed 8 July 2013).

[Eden and Turner, 2007] Eden, A. H. and Turner, R. (2007). Problems in the ontology of computer programs. *Applied Ontology*, 2(1):13–36. Preprint accessed 23 October 2014 from:
http://www.eden-study.org/articles/2007/problems-ontology-programs_ao.pdf.

[Editorial, 2006] Editorial (2006). Bush restrained. *New York Times*, page WK13.
http://www.nytimes.com/2006/09/17/opinion/17sun1.html (accessed 11 December 2013).

[Edmonds and Warburton, 2010] Edmonds, D. and Warburton, N. (2010). *Philosophy Bites*. Oxford University Press, Oxford.

[Egan, 2010] Egan, F. (2010). Computational models: A modest role for content. *Studies in History and Philosophy of Science*, 41(3):253–259.

[Egan, 2012] Egan, F. (2012). Metaphysics and computational cognitive science: Let's not let the tail wag the dog. *Journal of Cognitive Science (South Korea)*, 13(1):39–49. http://j-cs.org/gnuboard/bbs/board.php?bo_table=__vol013i1&wr_id=2
(accessed 9 November 2014).

[Egan, 2014] Egan, F. (2014). How to think about mental content. *Philosophical Studies*, 170(1):115–135. Preprint at https://www.academia.edu/4160744/How_to_think_about_Mental_Content (accessed 23 February 2015); video at https://vimeo.com/60800468 (accessed 23 February 2015).

[Einstein, 1940] Einstein, A. (1940). Considerations concerning the fundaments of theoretical physics. *Science*, 91(2369):487–492.

[Ekdahl, 1999] Ekdahl, B. (1999). Interactive computing does not supersede church's thesis. *The Association of Management and the International Association of Management, 17th Annual International Conference, San Diego, CA, August 6–8, Proceedings Computer Science*, 17(2):261–265.

[Elser, 2012] Elser, V. (2012). In a class by itself. *American Scientist*, 100:418–420. http://www.americanscientist.org/bookshelf/pub/in-a-class-by-itself, http://www.americanscientist.org/authors/detail/veit-elser (both accessed 16 December 2013).

[Ensmenger, 2003] Ensmenger, N. (2003). Bits of history: Review of a.r. burks's *who invented the computer? the legal battle that changed computing history*. *American Scientist*, 91:467–468. http://www.americanscientist.org/bookshelf/pub/bits-of-history
(accessed 30 October 2013).

[Everett, 2012] Everett, M. (2012). Answer to 'what's the most important question, and why?'. *Philosophy Now*, 92:38–41.

[Eysenck, 1990] Eysenck, M. W. (1990). Artificial intelligence. In Eysenck, M., editor, *The Blackwell Dictionary of Cognitive Psychology*, page 22. Basil Blackwell, Oxford.

[Feferman, 1992] Feferman, S. (1992). Turing's 'oracle': From absolute to relative computability—and back. In Echeverria, J., Ibarra, A., and Mormann, T., editors, *The Space of Mathematics: Philosophical, Epistemological, and Historical Explorations*, pages 314–348. Walter de Gruyter, Berlin.

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

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

[Feyerabend, 1975] Feyerabend, P. (1975). *Against Method: Outline of an Anarchistic Theory of Knowledge*. Verso, London.

[Fisher, 1989] Fisher, L. M. (1989). Xerox sues apple computer over macintosh copyright. *New York Times'*. Accessed 8 February 2014 from: http://www.nytimes.com/1989/12/15/business/company-news-xerox-sues-apple-computer-over-macintosh-copyright.html.

[Fitch et al., 2005] Fitch, W. T., Hauser, M. D., and Chomsky, N. (2005). The evolution of the language faculty: Clarifications and implications. *Cognition*, 97:179–210. Accessed 14 July 2014 from:
http://dash.harvard.edu/bitstream/handle/1/3117935/Hauser_EvolutionLanguageFaculty.pdf.

[Flanagan, 2012] Flanagan, O. (2012). Buddhism without the hocus-pocus. *The Chronicle [of Higher Education] Review*, 58(19):B4–B5.

[Fletcher, 1972] Fletcher, J. (1972). Indicators of humanhood: A tentative profile of man. *Hastings Center Report*, 2(5):1–4.

[Fletcher, 2007] Fletcher, L. R. (2007). Slow reading: The affirmation of authorial intent. http://www.freelance-academy.org/slowread.htm. Accessed 18 February 2014.

[Florman, 1994] Florman, S. C. (1994). *The Existential Pleasures of Engineering, Second Edition*. St. Martin's Press, New York.

[Fodor, 1968] Fodor, J. A. (1968). *Psychological Explanation: An Introduction to the Philosophy of Psychology*. Random House, New York.

[Fodor, 1974] Fodor, J. A. (1974). Special sciences (or: The disunity of science as a working hypothesis). *Synthese*, 28(2):97–115.

[Fodor, 1978] Fodor, J. A. (1978). Tom Swift and his procedural grandmother. *Cognition*, 6:229–247. Accessed 11 December 2013 from:
http://www.nyu.edu/gsas/dept/philo/courses/mindsandmachines/Papers/tomswift.pdf.

[Fodor, 1980] Fodor, J. A. (1980). Methodological solipsism considered as a research strategy in cognitive psychology. *Behavioral and Brain Sciences*, 3:63–109.

[Fodor, 1981] Fodor, J. A. (1981). The mind-body problem. *Scientific American*, 244(1):114–123.

[Foley, 2002] Foley, J. (2002). Computing > computer science. *Computing Research News*, 14(4):6. revised version online at:
http://archive.cra.org/reports/computing/index.html
(accessed 18 June 2013); response by Robert L. Glass online at:
http://archive.cra.org/reports/computing/glass.html
(accessed 18 June 2013).

[Forster, 1909] Forster, E. (1909). The machine stops. In ED, editor, *TITLE*. PUB, ADDR. http://archive.ncsa.illinois.edu/prajlich/forster.html.

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

[Forsythe, 1967b] Forsythe, G. E. (1967b). What to do till the computer scientist comes. *American Mathematical Monthly*, 75(5):454–462.

[Forsythe, 1968] Forsythe, G. E. (1968). Computer science and education. *Proceedings IFIP 68 Cong*, pages 92–106.

[Fortnow, 2009] Fortnow, L. (2009). The status of the P versus NP problem. *Communications of the ACM*, 52(9):78–86. Accessed 7 April 2014 from: http://cacm.acm.org/magazines/2009/9/38904-the-status-of-the-p-versus-np-problem/fulltext.

[Fortnow, 2010] Fortnow, L. (2010). What is computation? *Ubiquity*, pages 1–5. Article 5 of DOI:10.1145/1895419.1921573. Accessed 28 June 2017 from: http://people.cs.uchicago.edu/∼fortnow/papers/compute.pdf.

[Fox, 2013] Fox, M. (2013). Janos starker, master of the cello, dies at 88. *New York Times*, page B16. http://www.nytimes.com/2013/04/30/arts/music/janos-starker-master-cellist-dies-at-88.html (accessed 12 May 2013).

[Franzén, 2005] Franzén, T. (2005). *Gödel's Theorem: An Incomplete Guide to Its Use and Abuse*. A K Peters, Wellesley, MA.

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

[Freeman, 1994] Freeman, P. A. (1994). Effective computer science. *ACM Computing Surveys*, 27(1):27–29.

[Freeth, 2006] Freeth, T., e. a. (2006). Decoding the ancient greek astronomical calculator known as the antikythera mechanism. *Nature*, 444:587–591.

[Freeth, 2009] Freeth, T. (2009). Decoding an ancient computer. *Scientific American*, 301(6):76–83.

[Frenkel, 2013] Frenkel, E. (2013). *Love and Math: The Heart of Hidden Reality*. Basic Books, New York.

[Frenkel, 1993] Frenkel, K. (1993). An interview with Robin Milner. *Communications of the ACM*, 36(1):90–97. Accessed 26 May 2014 from: http://delivery.acm.org/10.1145/160000/151241/a1991-frenkel.pdf.

[Friedman and Kahn, 1997] Friedman, B. and Kahn, Peter H., J. (1997). People are responsible, computers are not. In Ermann, M. D., Williams, M. B., and Shauf, M. S., editors, *Computers, Ethics, and Society, Second Edition*, pages 303–314. Oxford University Press, New York. Excerpt from their "Human Agency and Responsible Computing: Implications for Computer System Design", *Journal of Systems and Software* (1992): 7–14.

[Frigg and Hartmann, 2012] Frigg, R. and Hartmann, S. (2012). Models in science. In Zalta, E. N., editor, *Stanford Encyclopedia of Philosophy (Fall 2012 Edition)*. online. http://plato.stanford.edu/archives/fall2012/entries/models-science/.

[Gal-Ezer and Harel, 1998] Gal-Ezer, J. and Harel, D. (1998). What (xelse) shoud cs educators know? *Communications of the ACM*, 41(9):77–84.

[Galbi, 1971] Galbi, E. W. (1971). Software and patents: A status report. *Communications of the ACM*, 14(4):274–280. Accessed 3 September 2014 from: http://simson.net/ref/2007/cs3610/ref/p274-galbi.pdf.

[Gandy, 1980] Gandy, R. (1980). Church's thesis and principles for mechanisms. In Barwise, J., Keisler, H., and Kunen, K., editors, *The Kleene Symposium*, pages 123–148. North-Holland.

[Gandy, 1988] Gandy, R. (1988). The confluence of ideas in 1936. In Herken, R., editor, *The Universal Turing Machine: A Half-Century Survey, Second Edition*, pages 51–102. Springer-Verlag, Vienna.

[Gazzaniga, 2010] Gazzaniga, M. S. (2010). Neuroscience and the correct level of explanation for understanding mind: An extraterrestrial roams through some neuroscience laboratories and concludes earthlings are not grasping how best to understand the mind-brain interface. *Trends in Cognitive Sciences*, 14(7):291–292.

[Gettier, 1963] Gettier, E. L. (1963). Is justified true belief knowledge? *Analysis*, 23:121–123. Reprinted in A.P. Griffiths (ed.), *Knowledge and Belief* (Oxford: Oxford University Press, 1967); http://www.ditext.com/gettier/gettier.html (accessed 19 September 2011).

[Gigerenzer and Goldstein, 1996] Gigerenzer, G. and Goldstein, D. G. (1996). Mind as computer: Birth of a metaphor. *Creativity Research Journal*, 9(2–3):131–144. http://web.ebscohost.com/ehost/pdfviewer/pdfviewer?sid=9fd0749e-892b-4f80-8663-7a539038aa82%40sessionmgr198&vid=1&hid=114 (accessed 31 January 2014).

[Gladwell, 2011] Gladwell, M. (2011). Creation myth. *New Yorker*, pages 44, 46, 48–50, 52A–53A. slightly different version accessed 8 February 2014 from: http://www.newyorker.com/reporting/2011/05/16/110516fa_fact_gladwell.

[Gleick, 2008] Gleick, J. (2008). 'if shakespeare had been able to google...'. *New York Review of Books*, 55(20):77–79; highlighted paragraph on p. 78, cols. 2–3.

[Gleiser, 2014] Gleiser, M. (2014). Does consciousness compute? *The Chronicle Review*, 61(2):B13–B15.

[Gödel, 930s] Gödel, K. (ca 1930s). Undecidable diophantine propositions. In Feferman, S. et al., editors, *Collected Works, Vol. III*, pages 164–175. Oxford University Press, Oxford.

[Gold, 1967] Gold, E. M. (1967). Language identification in the limit. *Information and Control*, 10:447–474.

[Goldberger, 1979] Goldberger, N. R. (1979). Developmental assumptions underlying models of general education. Paper presented at the Conference on General Education, William Patterson College.

[Goldin et al., 2006] Goldin, D., Smolka, S. A., and Wegner, P., editors (2006). *Interactive Computation: The New Paradigm*. Springer-Verlag, Berlin. Some chapters available online at:
http://www.springer.com/computer/theoretical+computer+science/book/978-3-540-34666-1.

[Goldin and Wegner, 2005] Goldin, D. and Wegner, P. (2005). The church-turing thesis: Breaking the myth. In Cooper, S. and Löwe, B., editors, *CiE: New Computational Paradigms*, pages 152–168. Springer LNCS 3526. Accessed 26 February 2014 from: http://www.cse.uconn.edu/∼dqg/papers/cie05.pdf.

[Goldstein, 2014] Goldstein, R. N. (2014). *Plato at the Googleplex: Why Philosophy Won't Go Away*. Pantheon Books, New York.

[Goodman, 1987] Goodman, N. D. (1987). Intensions, church's thesis, and the formalization of mathematics. *Notre Dame Journal of Formal Logic*, 28(4):473–489. Accessed 10 April 2014 from:
http://projecteuclid.org/download/pdf_1/euclid.ndjfl/1093637644.

[Google, 2012] Google, I. (2012). Google's 4/12/12 copyright liability trial brief, case no. 3:10-cv 03561-wha, document 897. http://www.wired.com/wp-content/uploads/blogs/wiredenterprise/wp-content/uploads/2012/04/20120412-Googles-Copyright-Stance.pdf. Accessed 26 September 2014.

[Gopnik, 1996] Gopnik, A. (1996). The scientist as child. *Philosophy of Science*, 63(4):485–514. http://courses.media.mit.edu/2002fall/mas962/MAS962/gopnik.pdf (accessed 13 May 2015).

[Gopnik, 2009a] Gopnik, A. (2009a). *The Philosophical Baby: What Children's Minds Tell Us about Truth, Love, and the Meaning of Life*. Farrar, Straus and Giroux, New York.

[Gopnik, 2009b] Gopnik, A. (2009b). What's the recipe? *New Yorker*, pages 106–112. Accessed 12 May 2014 from:
http://www.newyorker.com/arts/critics/atlarge/2009/11/23/091123crat_atlarge_gopnik.

[Gopnik, 2013] Gopnik, A. (2013). Moon man: What galileo saw. *New Yorker*, pages 103–109. online at:
http://www.newyorker.com/arts/critics/atlarge/2013/02/11/130211crat_atlarge_gopnik (accessed 24 July 2013).

[Gries, 1981] Gries, D. (1981). *The Science of Programming, 3rd printing*. Springer-Verlag, 1985, New York.

[Grobart, 2011] Grobart, S. (2011). Spoiled by the all-in-one gadget. *New York Times*, page WK3. http://www.nytimes.com/2011/03/27/weekinreview/27grobart.html (accessed 25 February 2014).

[Grover, 1999] Grover, L. K. (1999). Quantum computing. *The Sciences*, pages 24–30. Accessed 9 December 2013 from: http://cryptome.org/qc-grover.htm.

[Guernsey, 2009] Guernsey, L. (2009). Computers track the elusive metaphor. *Chronicle of Higher Education*, page A11.

[Gurevich, 1999] Gurevich, Y. (1999). The sequential ASM thesis. *Bulletin of the European Association for Theoretical Computer Science*, 67:93–124. http://research.microsoft.com/en-us/um/people/gurevich/Opera/136.pdf (accessed 24 December 2013).

[Gurevich, 2000] Gurevich, Y. (2000). Sequential abstract-state machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111. http://research.microsoft.com/en-us/um/people/gurevich/opera/141.pdf (accessed 17 December 2013).

[Gurevich, 2011] Gurevich, Y. (2011). What is an algorithm? In *SOFSEM 2012: Theory and Practice of Computer Science*, pages 31–42. Springer Lecture Notes in Computer Science 7147. http://research.microsoft.com/pubs/155608/209-3.pdf (accessed 17 December 2013).

[Gurevich, 2012] Gurevich, Y. (2012). Foundational analyses of computation. Technical Report MSR-TR-2012-14, Microsoft Research, Redmond, WA. http://research.microsoft.com/pubs/158617/210.pdf (accessed 21 January 2014).

[Gurevich, 2013] Gurevich, Y. ((accessed 24 December 2013)). transcript of aaai talk. http://msrvideo.vo.msecnd.net/rmcvideos/120362/120362.doc. see also: https://www.facebook.com/events/47127123902/ (accessed 24 December 2013).

[Gutting, 2012] Gutting, G. (2012). Philosophy—what's the use? *New York Times*. online at http://opinionator.blogs.nytimes.com/2012/01/25/philosophy-whats-the-use/ (accessed 29 May 2013).

[Guzdial and Kay, 2010] Guzdial, M. and Kay, A. (2010). The core of computer science:
Alan kay's 'triple whammy'. http://computinged.wordpress.com/2010/05/24/the-core-of-computer-science-alan-kays-triple-whammy/. Part of Guzdial's *Computing Education Blog*; accessed 28 October 2014.

[Haberman, 2009] Haberman, C. (2009). The story of a landing. *New York Times Book Review*. http://www.nytimes.com/2009/11/29/books/review/Haberman-t.html (accessed 14 November 2012).

[Habib, 1983] Habib, S. (1983). Emulation. In Ralston, A. and Edwin D.R̃eilly, J., editors, *Encyclopedia of Computer Science and Engineering, 2nd edition*, pages 602–603. Van Nostrand Reinhold, New York.

[Haigh, 2013] Haigh, T. (2013). 'stored program concept' considered harmful: History and historiography. In Bonizzoni, P., Brattka, V., and Löwe, B., editors, *CiE 2013*, pages 241–251. Springer-Verlag Lecture Notes in Computer Science 7921, Berlin.

[Halmos, 1973] Halmos, P. R. (1973). The legend of John von Neumann. *American Mathematical Monthly*, pages 382–394. http://poncelet.math.nthu.edu.tw/disk5/js/biography/v-n.pdf, http://stepanov.lk.net/mnemo/legende.html (accessed 7 April 2014).

[Halpern et al., 2001] Halpern, J. Y., Harper, R., Immerman, N., Kolaitis, P. G., Vardi, M. Y., and Vianu, V. (2001). On the unusual effectiveness of logic in computer science. *Bulletin of Symbolic Logic*, 7(2):213–236. online at: http://www.lsi.upc.edu/~roberto/EffectivenessOfLogic.pdf (accessed 17 June 2013).

[Hamming, 1968] Hamming, R. (1968). One man's view of computer science. *Journal of the Association for Computing Machinery*, 16(1):3–12.

[Hamming, 1980] Hamming, R. (1980). The unreasonable effectiveness of mathematics. *American Mathematical Monthly*, 87(2).

[Hamming, 1998] Hamming, R. (1998). Mathematics on a distant planet. *American Mathematical Monthly*, 105(7):640–650.

[Hammond, 2003] Hammond, T. A. (6 May 2003). What is computer science? myths vs. truths. http://web.archive.org/web/20030506091438/http://www.columbia.edu/~tah10/cs1001/whatcs.html. (accessed 21 June 2013).

[Harman, 1965] Harman, G. (1965). The inference to the best explanation. *The Philosophical Review*, 74(1):88–95.

[Hartmanis, 1993] Hartmanis, J. (1993). Some observations about the nature of computer science. In Shyamasundar, R., editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 761 of *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin / Heidelberg. 10.1007/3-540-57529-4_39.

[Hartmanis, 1995a] Hartmanis, J. (1995a). 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, 1995b] Hartmanis, J. (1995b). Response to the essays 'on computational complexity and the nature of computer science'. *ACM Computing Surveys*, 27(1):59–61.

[Hartmanis and Lin, 1992] 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.

[Hartmanis and Stearns, 1965] Hartmanis, J. and Stearns, R. (1965). On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306.

[Hartmanis and Stearns, 1967] Hartmanis, J. and Stearns, R. (1967). Sets of numbers defined by finite automata. *American Mathematical Monthly*, pages 539–542.

[Hartree, 1949] Hartree, D. (1949). *Calculating Instruments and Machines*. University of Illinois Press, Urbana, IL.

[Haugeland, 1981] Haugeland, J. (1981). Analog and analog. *Philosophical Topics*, 12(1):213–225.

[Hauser et al., 2002] Hauser, M. D., Chomsky, N., and Fitch, W. T. (2002). The faculty of language: What is it, who has it, and how did it evolve. *Science*, 298:1569–1579. Accessed 14 July 2014 from:
http://www.chomsky.info/articles/20021122.pdf.

[Hayes, 1990] Hayes, B. (1990). No forwarding address. *The Sciences*, pages 8–11. Accessed 18 March 2014 from http://bit-player.org/wp-content/extras/bph-publications/Sciences-1990-09-Hayes- forwarding.pdf.

[Hayes, 1995] Hayes, B. (1995). The square root of not. *American Scientist*, 83:304–308. http://www.americanscientist.org/issues/num2/the-square-root-of-not/ (accessed 6 December 2013).

[Hayes, 2007a] Hayes, B. (2007a). Calculating the weather. *American Scientist*, 95(3). online at:
http://www.americanscientist.org/bookshelf/pub/calculating-the-weather
(accessed 3 July 2013).

[Hayes, 2007b] Hayes,
B. (2007b). Trains of thought. *American Scientist*, 95(2). Accessed 31 January 2014 from: http://www.americanscientist.org/issues/pub/trains-of-thought.

[Hayes, 2014a] Hayes, B. (2014a). Pencil, paper, and pi. *American Scientist*, 102(1):342–345. Accessed 22 October 2014 from:
http://www.americanscientist.org/libraries/documents/20148513205110590-2014-09CompSciHayes.pdf.

[Hayes, 2014b] Hayes, B. (2014b). Programming your quantum computer. *American Scientist*, 102(1):22–25. Accessed 19 January 2014 from:
https://www.americanscientist.org/issues/pub/programming-your-quantum-computer/.

[Hayes, 2015] Hayes, B. (2015). The 100-billion-body problem. *American Scientist*, 103(2):90–93. http://www.americanscientist.org/issues/pub/the-100-billion-body-problem (accessed 13 May 2015).

[Hayes, 1997] Hayes, P. J. (1997). What is a computer? *Monist*, 80(3):389–404.

[Hearst and Hirsh, 2000] Hearst, M. and Hirsh, H. (2000). Ai's greatest trends and controversies. *IEEE Intelligent Systems*, 15(1):8–17. Accessed 2 July 2014 from: http://www.cs.cornell.edu/courses/cs472/2002fa/handouts/challenges-ai.pdf.

[Hedger, 1998] Hedger, L. (1998). Analog computation: Everything old is new again. *Indiana University Research & Creative Activity*, 21(2). Accessed 31 January 2014 from:
http://www.indiana.edu/~rcapub/v21n2/p24.html.

[Heng, 2014] Heng, K. (2014). The nature of scientific proof in the age of simulations. *American Scientist*, 102(3):174–177. Accessed 6 May 2014 from:
http://www.americanscientist.org/issues/pub/2014/3/the-nature-of-scientific-proof-in-the-age-of-simulations.

[Herman, 1983] Herman, G. (1983). Algorithms, theory of. In Ralston, A. S. and Riley, E. D., editors, *Encyclopedia of Computer Science, 3rd edition*, pages 37–39. Van Nostrand Reinhold, New York.

[Higginbothan, 2014] Higginbothan, A. (2014). The disillusionist. *New York Times Magazine*, pages 48–53, 60–61. Online as "The Unbelievable Skepticism of the Amazing Randi", accessed 15 December 2014 from:
http://www.nytimes.com/2014/11/09/magazine/the-unbelievable-skepticism-of-the-amazing-randi.html.

[Hilbert, 1900] Hilbert, D. (1900). Mathematical problems: Lecture delivered before the international congress of mathematicians at paris in 1900. *Bulletin of the American Mathematical Society*, 8(10):437–479. trans. by Mary Winston Newson; first published in *Göttinger Nachrichten* (1900): 253–297.

[Hilbert and Ackermann, 1928] Hilbert, D. and Ackermann, W. (1928). *Principles of Mathematical Logic*. Chelsea, New York (1950). Robert E. Luce (ed.); Lewis M. Hammon, George G. Leckie, & F. Steinhardt (trans.); based on the 1938 second edition of *Grundzüge der Theoretischen Logik*.

[Hill, 2008] Hill, R. (25 September 2008). Empire, regime, and perspective change in our creative activities. Conference on ReVisioning the (W)hole II: Curious Intersections. http://web.archive.org/web/20080925064006/http://www.newhumanities.org/events/revisioning.html (accessed 16 December 2013).

[Hills, 2007] Hills, D. (November 2007). professional website
. http://www.stanford.edu/~dhills/cv.html (accessed 12 February 2013).

[Hilpinen, 2011] Hilpinen, R. (2011). Artifact. In Zalta, E. N., editor, *Stanford Encyclopedia of Philosophy*. online. http://plato.stanford.edu/entries/artifact/ (accessed 15 February 2013).

[Hintikka and Mutanen, 1997] Hintikka, J. and Mutanen, A. (1997). An alternative concept of computability. In Hintikka, J., editor, *Language, Truth, and Logic in Mathematics*, pages 174–188. Springer, Dordrecht, The Netherlands.

[Hirst, 1991] Hirst, G. (1991). Existence assumptions in knowledge representation. *Artificial Intelligence*, 49:199–242.

[Hoare, 1969] Hoare, C. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583.

[Hoare, 1986] Hoare, C. (1986). Mathematics of programming. *Byte*, page 115ff. Reprinted in Timothy R. Colburn, James H. Fetzer, & Terry L. Rankin (eds.), *Program Verification: Fundamental Issues in Computer Science* (Dordrecht, Holland: Kluwer Academic Publishers, 1993): 135–154.

[Hodges, 2013] Hodges, A. (2013). Church's review of computable numbers. In Cooper, S. B. and van Leeuwen, J., editors, *Alan Turing: His Work and Impact*, pages 117–118. Elsevier, Amsterdam.

[Hoffmann, 2010] Hoffmann, L. (2010). Amir Pnueli: Ahead of his time. *Communications of the ACM*, 53(1):22–23. http://cacm.acm.org/magazines/2010/1/55750-amir-pnueli-ahead-of-his-time/fulltext (accessed 4 March 2015).

[Hofstadter, 1979] Hofstadter, D. (1979). *"Gödel, Escher, Bach: An Eternal Golden Braid"*. Basic Books, New York.

[Hofstadter, 2007] Hofstadter, D. (2007). *I Am a Strange Loop*. Basic Books, New York.

[Hofstadter and Sander, 2013] Hofstadter, D. and Sander, E. (2013). *Surfaces and Essences: Analogy as the Fuel and Fire of Thinking*. Basic Books, New York.

[Hollan et al., 2000] Hollan, J., Hutchins, E., and Kirsh, D. (2000). Distributed cognition: Toward a new foundation for human-computer interaction research. *ACM Transactions on Computer-Human Interaction*, 7(2):174–196.

[Holland, 2012] Holland, J. H. (2012). *Signals and Boundaries: Building Blocks for Complex Adaptive Systems*. MIT Press, Cambridge, MA.

[Holst, 2000] Holst, P. A. (2000). Analog computer. In Ralston, A., Reilly, E. D., and Hemmendinger, D., editors, *Encyclopedia of Computer Science, Fourth Edition*, pages 53–59. Grove's Dictionaries, New York.

[Holt, 2009] Holt, J. (2009). Death: Bad? *New York Times Book Review*, page BR27. http://www.nytimes.com/2009/02/15/books/review/Holt-t.html.

[Hopcroft and Ullman, 1969] Hopcroft, J. E. and Ullman, J. D. (1969). *Formal Languages and Their Relation to Automata*. Addison-Wesley, Reading, MA.

[Hopper, 1981] Hopper, G. M. (1981). The first bug. *Annals of the History of Computing*, 3(3):285–286. online at: http://ieeexplore.ieee.org/stamp/stamp.jsp?reload=true&tp=&arnumber=4640691 (accessed 6 November 2013).

[Horwich, 2013] Horwich, P. (2013). Was wittgenstein right? *New York Times*. online at http://opinionator.blogs.nytimes.com/2013/03/03/was-wittgenstein-right/ (accessed 29 May 2013).

[Huber, 1966] Huber, H. G. (1966). Algorithm and formula. *Communications of the ACM*, 9(9):653–654.

[Huemer, 2010] Huemer, W. (2010). Franz brentano. In Zalta, E. N., editor, *Stanford Encyclopedia of Philosophy*. online. http://plato.stanford.edu/entries/brentano/.

[Hugo, 1862] Hugo, V. (1862). *Les Misérables*. Signet Classics, 1987, New York.

[Humphreys, 1990] Humphreys, P. (1990). Computer simulations. *PSA: Proceedings of the [1990] Biennial Meeting of the Philosophy of Science Association*, 2:497–506.

[Humphreys, 2002] Humphreys, P. (2002). Computational models. *Philosophy of Science*, 69:S1–S11. http://ist.uap.asia/∼jpira/compthink/17232634.pdf (accessed 12 January 2015).

[Hutchins, 995a] Hutchins, E. (1995a). *Cognition in the Wild*. MIT Press, Cambridge, MA.

[Hutchins, 995b] Hutchins, E. (1995b). How a cockpit remembers its speeds. *Cognitive Science*, 19:265–288.

[Huxley, 1932] Huxley, A. (1932). *Brave New World*. online. http://www.huxley.net/bnw/.

[Irmak, 2012] Irmak, N. (2012). Software is an abstract artifact. *Grazer Philosophische Studien*, 86:55–72. Accessed 4 August 2014 from: http://philpapers.org/archive/IRMSIA.pdf.

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

[Jackendoff and Pinker, 2005] Jackendoff, R. and Pinker, S. (2005). The nature of the language faculty and its implications for evolution of language. *Cognition*, 97:211–225. Accessed 14 July 2014 from: http://pinker.wjh.harvard.edu/articles/papers/2005_09_Jackendoff_Pinker.pdf.

[Jackson, 1960] Jackson, A. S. (1960). *Analog Computation*. McGraw-Hill, New York.

[Jacob, 1999] Jacob, C. (1999). What is computer science? Chapter 3; http://pages.cpsc.ucalgary.ca/∼jacob/Courses/Fall99/CPSC231/03-ComputerScience.pdf, accessed 20 September 2011.

[Jacob, 2010] Jacob, P. (2010). Intentionality. In Zalta, E. N., editor, *Stanford Encyclopedia of Philosophy*. online. http://plato.stanford.edu/entries/intentionality/.

[James, 1892] James, W. (1892). Psychology: Briefer course. In Myers, G. E., editor, *William James: Writings 1878–1899*, pages 1–443. Library of America, New York. Original edition accessed 16 September 2014 from: https://archive.org/details/psychologybriefe00willuoft.

[James, 1897] James, W. (1897). The will to believe. In Myers, G. E., editor, *William James: Writings 1878–1899*, pages 457–479. Library of America, New York. Project Gutenberg version accessed 16 September 2014 from: http://www.gutenberg.org/files/26659/26659-h/26659-h.htm.

[Johnson, 2001] Johnson, G. (2001). All science is computer science. *New York Times*. online at: http://www.nytimes.com/2001/03/25/weekinreview/the-world-in-silica-fertilization-all-science-is-computer-science.html (accessed 11 June 2013).

[Johnson, 2002] Johnson, G. (2002). To err is human. *New York times*. online at: http://www.nytimes.com/2002/07/14/weekinreview/deus-ex-machina-to-err-is-human.html (accessed 11 June 2013).

[Johnson-Laird, 1981] Johnson-Laird, P. N. (1981). Mental models in cognitive science. In Norman, D. A., editor, *Perspectives on Cognitive Science*, chapter 7, pages 147–191. Ablex, Norwood, NJ.

[Johnson-Laird, 1988] Johnson-Laird, P. N. (1988). *The Computer and the Mind: An Introduction to Cognitive Science*. Harvard University Press, Cambridge, MA. Ch. 3 ("Computability and Mental Processes"), pp. 37–53.

[Kahneman, 2011] Kahneman, D. (2011). *Thinking, Fast and Slow*. Farrar, Strauss and Giroux, New York.

[Kaiser, 2012] Kaiser, D. (2012). Paradoxial roots of 'social construction'. *Science*, 335:658–659.

[Kalmár, 1959] Kalmár, L. (1959). An argument against the plausibility of church's thesis. In Heiting, A., editor, *Constructivity in Mathematics*, pages 72–80. North-Holland, Amsterdam.

[Kanat-Alexander, 2008] Kanat-Alexander, M. (10 October 2008). What is a computer? http://www.codesimplicity.com/post/what-is-a-computer/. Accessed 29 March 2014 from Code Simplicity blog.

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

[Kay, 2010] Kay, M. (2010). Introduction. In Flickinger, D. and Oepen, S., editors, *Collected Papers of Martin Kay: A Half Century of Computational Linguistics*, pages 1–18. CSLI Studies in Computational Linguistics, Stanford, CA.

[Kearns, 1997] Kearns, J. (1997). Thinking machines: Some fundamental confusions. *Minds and Machines*, 7(2):269–287.

[Keith, 2014] Keith, T. (2014). The letter that kicked off a radio career. *NPR.org*. Accessed 14 July 2014 from: http://www.npr.org/2014/07/05/328512614/the-letter-that-kicked-off-a-radio-career.

[Kemeny, 1959] Kemeny, J. G. (1959). *A Philosopher Looks at Science*. D. van Nostrand, Princeton, NJ.

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

[Kleene, 1936a] Kleene, S. C. (1935-1936a). General recursive functions of natural numbers. *Mathematische Annalen*, 112:727–742.

[Kleene, 1936b] Kleene, S. C. (1936b). λ-definability and recursiveness. *Duke Mathematical Journal*, 2:340–353.

[Kleene, 1952] Kleene, S. C. (1952). *Introduction to Metamathematics*. D. Van Nostrand, Princeton, NJ.

[Kleene, 1967] Kleene, S. C. (1967). *Mathematical Logic*. Wiley, New York.

[Kleene, 1987] Kleene, S. C. (1987). Reflections on church's thesis. *Notre Dame Journal of Formal Logic*, 28(4):490–498. Accessed 9 April 2014 from: http://projecteuclid.org/download/pdf_1/euclid.ndjfl/1093637645.

[Kleene, 1995] Kleene, S. C. (1995). Turing's analysis of computability, and major applications of it. In Herken, R., editor, *The Universal Turing Machine: A Half-Century Survey, Second Edition*, pages 15–49. Springer-Verlag, Vienna.

[Kling et al., 1993] Kling, R., Wegner, P., Rice, J. R., and Weiss, E. A. (1993). Broadening computer science. *Communications of the ACM*, 36(2):15–19.

[Knobe, 2009] Knobe, J. (2008/2009). Can a robot, an insect, or god be aware? *Scientific American Mind*, 19(6):68–71.

[Knuth, 1966] Knuth, D. E. (1966). Algorithm and program: Information and data. *Communications of the ACM*, 9(9):654.

[Knuth, 972a] Knuth, D. E. (1972a). Ancient babylonian algorithms. *Communications of the ACM*, 15(7):671–677.

[Knuth, 972b] Knuth, D. E. (1972b). George forsythe and the development of computer science. *Communications of the ACM*, 15(8):721–727.

[Knuth, 1973] Knuth, D. E. (1973). *The Art of Computer Programming, Second Edition*. Addison-Wesley, Reading, MA.

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

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

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

[Knuth, 2001] Knuth, D. E. (2001). *Things a Computer Scientist Rarely Talks About*. CSLI Publications, Stanford, CA. CSLI Lecture Notes Number 136.

[Knuth, 2014] Knuth, D. E. (2014). Twenty questions for donald knuth. Informit.com. Accessed 3 June from:
http://www.informit.com/articles/article.aspx?p=2213858&WT.mc_id=Author_Knuth_20Questions.

[Koen, 1988] 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; http://dx.doi.org/10.1080/03043798808939429.

[Koepsell, 1998] Koepsell, D. R. (1998). Review of smith 1996. *Philosophical Psychology*, 11(3):389–391.

[Koepsell, 2000] Koepsell, D. R. (2000). *The Ontology of Cyberspace: Philosophy, Law, and the Future of Intellectual Property*. Open Court, Chicago.

[Kolata, 2004] Kolata, G. (2004). New studies question value of opening arteries. *New York Times*, pages A1, A21. http://www.nytimes.com/2004/03/21/us/new-heart-studies-question-the-value-of-opening-arteries.html (accessed 14 November 2012).

[Kornblith, 2013] Kornblith, H. (2013). Naturalism vs. the first-person perspective. *Proceedings & Addresses of the American Philosophical Association*, 87:122–141.

[Korsmeyer, 2012] Korsmeyer, C. (2012). Touch and the experience of the genuine. *British Journal of Aesthetics*, 52(4):365–377.

[Kramer, 2007] Kramer, J. (2007).   Is abstracton the key to computing? *Communications of the ACM*, 50(4):36–42.

[Krantz, 1984] Krantz, S. G. (1984). Letter to the editor. *American Mathematical Monthly*, 91(9):598–600.

[Krebs and Thomas, 1981] Krebs, A. and Thomas, Jr., R. M. (1981). Historic moth. *New York Times*. online at:
http://www.nytimes.com/1981/08/07/nyregion/notes-on-people-historic-moth.html (accessed 6 November 2013).

[Kripke, 2013] Kripke, S. A. (2013). The church-turing 'thesis' as a special corollary of gödel's completeness theorem. In Copeland, B. J., Posy, C. J., and Shagrir, O., editors, *Computability: Turing, Gödel, Church, and Beyond*, pages 77–104. MIT Press, Cambridge, MA.

[Kugel, 2002] Kugel, P. (2002). Computing machines can't be intelligent (...and turing said so). *Minds and Machines*, 12(4):563–59?

[Kuhn, 1957] Kuhn, T. S. (1957). *The Copernican Revolution: Planetary Astronomy in the Development of Western Thought*. Harvard University Press, Cambridge, MA.

[Kuhn, 1962] Kuhn, T. S. (1962). *The Structure of Scientific Revolutions*. University of Chicago Press, Chicago.

[LaChat, 1986] LaChat, M. R. (1986). Artificial intelligence and ethics: An exercise in the moral imagination. *AI Magazine*, 7(2):70–79.

[Ladyman, 2009] Ladyman, J. (2009). What does it mean to say that a physical system implements a computation? *Theoretical Computer Science*, 410(4–5):376–383. http://ac.els-cdn.com/S0304397508007238/1-s2.0-S0304397508007238-main.pdf?_tid=04248730-5f6c-11e4-8b44-00000aab0f26&acdnat=1414588006_ff4ead0d50f1774e6d0779cf705e19e6 (accessed 28 October 2014).

[Lakoff, 1987] Lakoff, G. (1987). *Women, Fire, and Dangerous Things: What Categories Reveal about the Mind*. University of Chicago Press, Chicago.

[Lakoff and Mark, 1980] Lakoff, G. and Mark, J. (1980). *Meaphors We Live By*. University of Chicago Press, Chicago.

[Lakoff and Mark, 980a] Lakoff, G. and Mark, J. (1980a). Conceptual metaphor in everyday language. *Journal of Philosophy*, 77(8):453–486.

[Lamport, 2011] Lamport, L. (2011). Euclid writes an algorithm: A fairytale. *International Journal of Software and Informatics*, 5(1-2, Part 1):7–20. http://research.microsoft.com/en-us/um/people/lamport/pubs/euclid.pdf (accessed 23 April 2015), page references to PDF version.

[Lamport, 2012] Lamport, L. (2012). How to write a 21st century proof. *Journal of Fixed Point Theory and Applications*, 11(1):43–63. Accessed 6 October 2014 from: http://research.microsoft.com/en-us/um/people/lamport/pubs/proof.pdf.

[Lamport, 2015] Lamport, L. (2015). Who builds a house without drawing blueprints? *Communications of the ACM*, 58(4):38–41. http://cacm.acm.org/magazines/2015/4/184705-who-builds-a-house-without-drawing-blueprints/fulltext (accessed 18 April 2015).

[Langewiesche, 2009] Langewiesche, W. (2009). *Fly by Wire: The Geese, the Glide, the Miracle on the Hudson*. Farrar, Strauss & Giroux, New York?

[Lanier, 2005] Lanier, J. (2005). Early computing's long, strange trip. *American Scientist*, 93(4):364–365. http://www.americanscientist.org/bookshelf/pub/early-computings-long-strange-trip (accessed 12 November 2013).

[Lazowska, 2014] Lazowska, E. (2014). Letter to the editor. *New York Times*, page A26. Accessed 26 May 2014 from: http://www.nytimes.com/2014/05/16/opinion/should-young-children-learn-coding.html.

[Leibniz, 1685] Leibniz, G. W. (1683–1685). Introduction to a secret encyclopedia. In Dascal, M., editor, *G.W. Leibniz: The Art of Controversies*, pages 219–224. Springer (2008), Dordrecht, The Netherlands. http://tinyurl.com/Leibniz1683 (accessed 6 December 2013).

[Leibniz, 1714] Leibniz, G. W. (1714). The principles of philosophy known as monadology. http://www.earlymoderntexts.com/pdfs/leibniz1714b.pdf. Trans. by Jonathan Bennett (July 2007). Accessed 29 March 2014 from *Some Texts from Early Modern Philosopny*.

[Leiter, 2005] Leiter, B. (12 October 2005). Why is there a nobel prize in economics? Leiter Reports: A Philosophy Blog. http://leiterreports.typepad.com/blog/2005/10/why_is_there_a_.html (accessed 4 November 2013).

[Leiter, 2009] Leiter, B. (20 September 2009). Alex rosenberg on cochrane and economics. Leiter Reports: A Philosophy Blog. http://leiterreports.typepad.com/blog/2009/09/alex-rosenberg-on-cochrane-and-economics.html (accessed 4 November 2013).

[Leiter, 2004] Leiter, B. (8 October 2004). Is economics a 'science'? Leiter Reports: A Philosophy Blog. http://leiterreports.typepad.com/blog/2004/10/is_economics_a_.html (accessed 4 November 2013).

[Leler, 1985] Leler, W. (3 December 1985). newsgroup message. net.ai. contribution to discussion of "definition of AI".

[Lem, 1971] Lem, S. (1971). Non serviam. In *A Perfect Vacuum*. Harcourt Brace Jovanovich, New York. 1979.

[Lemonick, 2015] Lemonick, M. (2015). The pluto wars revisited. http://www.newyorker.com/tech/elements/nasa-dawn-ceres-pluto-dwarf-planets. *The New Yorker* online.

[Lessing, 1778] Lessing, G. E. (1778). Anti-goetze: Eine duplik. In Göpfert, H., editor, *Werke*, pages Vol. 8, pp. 32–33. (published online). http://harpers.org/blog/2007/11/lessings-search-for-truth/ (accessed 5 February 2013).

[Lewis, 1953] Lewis, W. (1953). Electronic computers and telephone switching. *Proceedings of the Institute of Radio Engineers*, 41(10):1242–1244.

[Lewontin, 2014] Lewontin, R. (2014). The new synthetic biology: Who gains? *New York Review of Books*, 61(8):22–23. Accessed 19 May 2014 from: http://www.nybooks.com/articles/archives/2014/may/08/new-synthetic-biology-who-gains/.

[Licklider and Taylor, 1968] Licklider, J. and Taylor, R. W. (1968). The computer as a communication device. *Science and Technology*. online at http://memex.org/licklider.pdf (accessed 22 May 2013).

[Lightman, 2012] Lightman, A. (2012). *Mr g*. Pantheon, New York.

[Lindell, 2001] Lindell, S. (24 January 2001). Computer science as a liberal art: The convergence of technology and reason. talk presented at Haverford College. online at: http://www.haverford.edu/cmsc/slindell/Presentations/Computer%20Science%20as%20a%20Liberal%20Art.pdf (accessed 1 July 2013).

[Lipton, 2004] Lipton, P. (2004). *Inference to the Best Explanation*. Routledge, 2nd edition edition.

[Lloyd and Ng, 2004] Lloyd, S. and Ng, Y. J. (2004). Black hole computers. *Scientific American*, 291(5):52–61.

[Lohr, 2006] Lohr, S. (2006). Group of university researchers to make web science a field of study. *New York Times*, page C6. Accessed 3 July 2013 from: http://www.nytimes.com/2006/11/02/technology/02compute.html.

[Lohr, 2008] Lohr, S. (2008). Does computing add up in the classroom? *New York Times Bits (blog)*. Accessed 8 July 2013 from: http://bits.blogs.nytimes.com/2008/04/01/does-computing-add-up-in-the-classroom/.

[Lohr, 2013] Lohr, S. (2013). Algorithms get a human hand in steering web. *New York Times*. Accessed 18 July 2014 from: http://www.nytimes.com/2013/03/11/technology/computer-algorithms-rely-increasingly-on-human-helpers.html.

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

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

[Loui, 1996] Loui, M. C. (1996). Computational complexity theory. *ACM Computing Surveys*, 28(1):47–49. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.15.9179&represent=rep1&type=pdf (accessed 6 December 2013).

[Loui, 1998] Loui, R. (1998). Review of smith 1996. *Artificial Intelligence*, 106:353–358.

[Lycan, 1990] Lycan, W. G. (1990). The continuity of levels of nature. In Lycan, W. G., editor, *Mind and Cognition: A Reader*, pages 77–96. Basil Blackwell, Cambridge, MA. Accessed 6 October 2014 from: http://mugwump.pitzer.edu/∼bkeeley/CLASS/FoNS/lycan.pdf.

[Lynch, 2013]
Lynch, M. P. (2013). Of flies and philosophers: Wittgenstein and philosophy. *New York Times*. online at http://opinionator.blogs.nytimes.com/2013/03/05/of-flies-and-philosophers-wittgenstein-and-philosophy/ (accessed 29 May 2013).

[Macari, 2012] Macari, M. (2012). Oracle thinks you can copyright a programming language, google disagrees. *The Verge*. Accessed 26 September 2014 from: http://www.theverge.com/2012/4/13/2944440/google-oracle-lawsuit-programming-language-copyright.

[MacFarlane, 2013] MacFarlane, A. (2013). Information, knowledge and intelligence. *Philosophy Now*, 98:18–20. http://philosophynow.org/issues/98/Information_Knowledge_and_Intelligence (accessed 24 December 2013).

[Machery, 2012] Machery, E. (2012). Why i stopped worrying about the definition of life. . . and why you should as well. *Synthese*, 185:145–164. Preprint accessed 2 May 2014 from: http://www.pitt.edu/∼machery/papers/Definition

[MacKenzie, 1992] MacKenzie, D. (1992). Computers, formal proofs, and the law courts. *Notices of the American Mathematical Society*, 39(9):1066–1069. Also see introduction by Keith Devlin, same issue, pp. 1065–1066.

[Madigan, 2014] Madigan, T. (2014). A mind is a wonderful thing to meet. *Philosophy Now*, 100:46–47. Accessed 26 February 2014 from: http://philosophynow.org/issues/100/A_Mind_is_a_Wonderful_Thing_to_Meet.

[Mahoney and Haigh (ed.), 2011] Mahoney, M. S. and Haigh (ed.), T. (2011). *Histories of Computing*. Harvard University Press, Cambridge, MA.

[Maida and Shapiro, 1982] Maida, A. S. and Shapiro, S. C. (1982). Intensional concepts in propositional semantic networks. *Cognitive Science*, 6:291–330. reprinted in Ronald J. Brachman & Hector J. Levesque (eds.), *Readings in Knowledge Representation* (Los Altos, CA: Morgan Kaufmann, 1985): 169–189.

[Malcolm, 2001] Malcolm, N. (2001). *Ludwig Wittgenstein: A Memoir, with a Biographical Sketch by G.H. von Wright, Second Edition, with Wittgenstein's Letters to Malcolm*. Clarendon Press, Oxford. Originally published in 1984.

[Malpas, 2000] Malpas, R. (2000). The universe of engineering: A uk perspective. Technical report, Royal Academy of Engineering, London. http://www.engc.org.uk/ecukdocuments/internet/document%20library/The%20Universe%20of%20Engineering%20Report%20(The%20Malpas%20Report).pdf, accessed 5 October 2011.

[Mander, 2007] Mander, K. (February 2007). Demise of computer science exaggerated. BCS: The Chartered Institute for IT; Features, Press and Policy. http://www.bcs.org/content/ConWebDoc/10138.

[Manovich, 2013] Manovich, L. (2013). The algorithms of our lives. *The Chronicle [of Higher Education] Review*, 60(16):B10–B13. Accessed 7 April 2014 from: https://chronicle.com/article/The-Algorithms-of-Our-Lives-/143557/.

[Manzano, 1997] Manzano, M. (1997). Alonzo church: His life, his work and some of his miracles. *History and Philosophy of Logic*, 18:211–232.

[Markov, 1954] Markov, A. (1954). Theory of algorithms. *Tr. Mat. Inst. Steklov*, 42:1–14. trans. by Edwin Hewitt, in *American Mathematical Society Translations*, Series 2, Vol. 15 (1960).

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

[Martin, 2015] Martin, C. (2015). She seconds that emotion. *The Chronicle [of Higher Education] Review*, 61(33):B16. http://chronicle.com/article/Choosing-Love/229569/ (accessed 14 June 2015).

[Marx, 1845] Marx, K. (1845). Theses on Feuerbach. https://www.marxists.org/archive/marx/works/1845/theses/theses.htm (accessed 14 March 2015).

[McAllister, 2012] McAllister, N. (2012). Oracle vs. google: Who owns the java apis? *InfoWorld*. Accessed 26 September 2014 from: http://www.infoworld.com/article/2617268/java/oracle-vs--google--who-owns-the-java-apis-.html.

[McBride, 2007] McBride, N. (22 January 2007). The death of computing. BCS: The Chartered Institute for IT; Features, Press and Policy. http://www.bcs.org/content/ConWebDoc/9662 (accessed 3 July 2013).

[McCarthy, 1988] McCarthy, J. (1 November 1988). contribution to newsgroup discussion of "ai as cs and the scientific epistemology of the commone sense world". Article 1818 of comp.ai.digest. http://www.cse.buffalo.edu/∼rapaport/mccarthy.txt.

[McCarthy, 2007] McCarthy, J. (12 November 2007). What is ai? http://www-formal.stanford.edu/jmc/whatisai.html.

[McCarthy, 1959] McCarthy, J. (1959). Programs with common sense. In Blake, D. and Uttley, A., editors, *Proceedings of the ["Teddington"] Symposium on Mechanization of Thought Processes*. HM Stationary Office, London. Original version accessed 7 July 2014 from: http://aitopics.org/sites/default/files/classic/TeddingtonConference/Teddington-1.3-McCarthy.pdf
Version accessed 7 July 2014 from McCarthy's archive at: http://www-formal.stanford.edu/jmc/mcc59/mcc59.html.

[McCarthy, 1963] 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. http://www-formal.stanford.edu/jmc/basis.html (accessed 13 May 2015), page references to online PDF version.

[McCarthy, 1974] McCarthy, J. (1974). Review of 'artificial intelligence: A general survey'. *Artificial Intelligence*, 5:317–322.

[McCarthy et al., 1955] McCarthy, J., Minsky, M., Rochester, N., and Shannon, C. (31 August 1955). A proposal for the dartmouth summer research project on artificial intelligence. http://www-formal.stanford.edu/jmc/history/dartmouth.html.

[McDermott, 2001] McDermott, D. (2001). *Mind and Mechanism*. MIT Press, Cambridge, MA.

[McDermott, 2014] McDermott, D. (2014). On the claim that a table-lookup program could pass the turing test. *Minds and Machines*, 24(2):143–188. Accessed 29 April 2014 from:
http://cs-www.cs.yale.edu/homes/dvm/papers/humongous.pdf
with supplementary material at:
http://link.springer.com/content/esm/art:10.1007/s11023-013-9333-3/file/MediaObjects/11023_2013_9333_MOESM1_ESM.pdf.

[McGinn, 2015]
McGinn, C. (2015). The science of philosophy. *Metaphilosophy*, 46(1):84–103.
http://onlinelibrary.wiley.com/store/10.1111/meta.12116/asset/meta12116.pdf?v=1&t=i5dvrsm5&s=210194c1272c25e8bac66889cc9db416aac0340d
(accessed 26 January 2015); video online at
https://www.youtube.com/watch?v=TEkTbq5EE_M
(accessed 26 January 2015); preprint online at
https://docs.google.com/file/d/0BzokFqaWjk4JZ0NPMHFiUnNnZWs/edit
(accessed 26 January 2015).

[McGinn, 2012b] McGinn, C. (4 March 2012b). Philosophy by another name. The Stone/The New York Times Opinionator. http://opinionator.blogs.nytimes.com/2012/03/04/philosophy-by-another-name/ (accessed 12 February 2013).

[McGinn, 2012a] McGinn, C. (9 March 2012a). Name calling: Philosophy as ontical science. The Stone/The New York Times Opinionator. http://opinionator.blogs.nytimes.com/2012/03/09/name-calling-philosophy-as-ontical-science/ (accessed 12 February 2013).

[McMillan, 2013] McMillan, R. (7 July 2013). The end of digital tyranny: Why the future of computing is analog. Wired. http://www.wired.com/wiredenterprise/2013/07/analogfuture/ (accessed 29 July 2013).

[McSherry, 2014] McSherry, C. (2014). Dangerous decision in oracle v. google: Federal circuit reverses sensible court ruling

on apis. https://www.eff.org/deeplinks/2014/05/dangerous-ruling-oracle-v-google-federal-circuit-reverses-sensible-lower-court. Accessed 26 September 2014 from Electronic Frontier Foundation.

[Melville, 2002] Melville, H. (1851/2002). *Moby-Dick (Norton Critical Edition, second edition)*. W.W. Norton, New York. Hershel Parker & Harrison Hayford (eds.).

[Menabrea, 1843] Menabrea, L. F. (1843). English translation of *notions sur la machine analytique de m. charles babbage. [Richard Taylor's] Scientific Memoirs*, 3:666ff. translation by Augusta Ada Lovelace.

[Mendelson, 1990] Mendelson, E. (1990). Second thoughts about church's thesis and mathematical proofs. *Journal of Philosophy*, 87(5):225–233.

[Mertens, 2004] Mertens, S. (2004). The revolution will be digitized. *American Scientist*, 92:195–196.

[Mervis and Rosch, 1981] Mervis, C. B. and Rosch, E. (1981). Categorization of natural objects. *Annual Review of Psychology*, 32:89–115.

[Micali, 2015]
Micali, S. (2015). What it means to receive the turing award. *Communications of the ACM*, 58(1):52–53. http://cacm.acm.org/magazines/2015/1/181611-what-it-means-to-receive-the-turing-award/fulltext
(accessed 26 January 2015).

[Michie, 1971] Michie, D. (1971). Formation and execution of plans by machine. In Findler, N. and Meltzer, B., editors, *Artificial Intelligence and Heuristic Programming*, pages 101–124. American Elsevier, New York.

[Mili et al., 1986] Mili, A., Desharnais, J., and Gagné, J. R. (1986). Formal models of stepwise refinement of programs. *ACM Computing Surveys*, 18(3):231–276.

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

[Milner, 1993] Milner, R. (1993). Elements of interaction: Turing award lecture. *Communications of the ACM*, 36(1):78–89. Accessed 26 May 2014 from: http://delivery.acm.org/10.1145/160000/151240/a1991-milner.pdf.

[Minsky, 1967] Minsky, M. (1967). *Computation: Finite and Infinite Machines*. Prentice-Hall, Englewood Cliffs, NJ.

[Minsky, 1968] Minsky, M. (1968). Preface. In Minsky, M., editor, *Semantic Information Processing*, page v. MIT Press, Cambridge, MA.

[Minsky, 1979] 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.

[Mish, 1983] Mish, F. C., editor (1983). *Webster's Ninth New Collegiate Dictionary*. Merriam-Webster, Springfield, MA.

[Mitcham, 1994] Mitcham, C. (1994). *Thinking through Techology: The Path between Engineering and Philosophy*. University of Chicago Press, Chicago.

[Mizoguchi and Kitamura, 2009] Mizoguchi, R. and Kitamura, Y. (2009). A functional ontology of artifacts. *The Monist*, 92(3):387–402.

[Monroe and Wineland, 2008] Monroe, C. R. and Wineland, D. J. (2008). Quantum computing with ions. *Scientific American*, pages 64–71. Accessed 29 March 2014 from http://www.cs.virginia.edu/∼robins/Quantum_Computing_with_Ions.pdf.

[Montague, 1960] Montague, R. (1960). Towards a general theory of computability. *Synthese*, 12(4):429–438.

[Montague, 1970] Montague, R. (1970). English as a formal language. In Thomason, R. H., editor, *Formal Philosophy: Selected Papers of Richard Montague*, pages 192–221. Yale University Press, 1974, New Haven, CT. Accessed 10 February 2014 from: http://strangebeautiful.com/uwo/metaphys/montague-formal-philosophy.pdf.

[Mooers, 1975] Mooers, C. N. (1975). Computer software and copyright. *Computing Surveys*, 7(1):45–72.

[Moor, 1978] Moor, J. H. (1978). Three myths of computer science. *British Journal for the Philosophy of Science*, 29(3):213–222.

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

[Moravec, 1998] Moravec, H. (1998). When will computer hardware match the human brain? *Journal of Evolution and Technology*, 1(1). Accessed 24 March 2014 from http://www.jetpress.org/volume1/moravec.htm and http://www.jetpress.org/volume1/moravec.pdf.

[Morris, 1938] Morris, C. (1938). *Foundations of the Theory of Signs*. University of Chicago Press, Chicago.

[Morris and Reilly, 2000] Morris, G. J. and Reilly, E. D. (2000). Digital computer. In Ralston, A., Reilly, E. D., and Hemmendinger, D., editors, *Encyclopedia of Computer Science, Fourth Edition*, pages 539–545. Grove's Dictionaries, New York.

[Mycielski, 1983] Mycielski, J. (1983). The meaning of the conjecture $p \neq np$ for mathematical logic. *American Mathematical Monthly*, 90:129–130.

[Nagel et al., 2001] Nagel, E., Newman, J. R., and Hofstadter, D. R. (2001). *Gödel's Proof, Revised Edition*. New York University Press, New York.

[Nagel, 1987] Nagel, T. (1987). *What Does It All Mean? A Very Short Introduction to Philosophy*. Oxford University Press, New York.

[Nahmias et al., 2006] Nahmias, E., Morris, S. G., Nadelhoffer, T., and Turner, J. (2006). Is incompatibilism intuitive? *Philosophy and Phenomenological Research*, 73(1):28–53.

[Natarajan, 2014] Natarajan, P. (2014). What scientists really do. *New York Review of Books*, 61(16):64–66. Accessed 5 November 2014 from: http://www.nybooks.com/articles/archives/2014/oct/23/what-scientists-really-do/.

[Naur, 2007] Naur, P. (2007). Computing versus human thinking. *Communications of the ACM*, 50(1):85–94.

[Nayebi, 2014] Nayebi, A. (2014). Practical intractability: A critique of the hypercomputation movement. *Minds and Machines*, 24(3):275–305.

[Neuberger, 2012] Neuberger, J. (2012). Benefits of a teacher and coach collaboration: A case study. *Journal of Mathematical Behavior*, 31(2):290–311.

[Newcombe et al., 2015] Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., and Deardeuff, M. (2015). How Amazon web services uses formal methods. *Communications of the ACM*, 58(4):66–73. http://delivery.acm.org/10.1145/2700000/2699417/p66-newcombe.pdf (accessed 18 April 2015) and http://m.cacm.acm.org/magazines/2015/4/184701-how-amazon-web-services-uses-formal-methods/fulltext (accessed 15 June 2015).

[Newell, 1980] Newell, A. (1980). Physical symbol systems. *Cognitive Science*, 4:135–183. Accessed 15 March 2014 from http://tinyurl.com/Newell1980.

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

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

[Newell et al., 1958] Newell, A., Shaw, J., and Simon, H. A. (1958). Elements of a theory of human problem solving. *Psychological Review*, 65(3):151–166.

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

[Nichols, 2011] Nichols, S. (2011). Experimental philosophy and the problem of free will. *Science*, 331(6023):1401–1403.

[Nilsson, 1971] Nilsson, N. J. (1971). *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill, New York.

[Nilsson, 1983] Nilsson, N. J. (1983). Artificial intelligence prepares for 2001. *AI Magazine*, 4(4):7–14.

[O'Connor and Robertson, 2005] O'Connor, J. and Robertson, E. (October 2005). The function concept. The MacTutor History of Mathematics Archive. http://www-history.mcs.st-andrews.ac.uk/HistTopics/Functions.html.

[of Computer Science, 2003] of Computer Science, G. W. U. D. (2003). Computer science careers. http://www.seas.gwu.edu/∼simhaweb/misc/cscareers.html. (accessed 21 June 2013).

[of DATA-LINK, 1958] of DATA-LINK, E. (1958). What's in a name? *Communications of the ACM*, 1(4):6.

[O'Neill, 2013] O'Neill, O. (2013). Interpreting the world, changing the world. *Philosophy Now*, Issue 95:8–9. online at:
http://philosophynow.org/issues/95/Interpreting_The_World_Changing_The_World (accessed 13 November 2013).

[Oracle America, 2012] Oracle America, I. (2012). Oracle's april 12, 2012 brief regarding copyright issues, case no. 3:10-cv 10-03561-wha, document 900. http://assets.sbnation.com/assets/1057275/Oracle_s_Brief.pdf. Accessed 26 September 2014.

[Orr, 2013] Orr, H. A. (2013). Awaiting a new darwin. *New York Review of Books*, 60(2):26–28.

[Pandya, 2013] Pandya, H. (2013). Shakuntala devi, 'human computer' who bested the machines, dies at 83. *New York Times*. Accessed 26 February 2014 from: http://www.nytimes.com/2013/04/24/world/asia/shakuntala-devi-human-computer-dies-in-india-at-83.html.

[Parlante, 2005] Parlante, N. (2005). What is computer science? *Inroads—The SIGSCE Bulletin*, 37(2):24–25.

[Parnas, 1985] Parnas, D. L. (1985). Software aspects of strategic defense systems. *American Scientist*, 73(5):432–440. Reprinted in *Communications of the ACM* 28(12) (December 1985): 1326–1335.

[Parnas, 1998] Parnas, D. L. (1998). Software engineering programmes are not computer science programs. *Annals of Software Engineering*, 6:19–37. pre-print online at:
http://www.cas.mcmaster.ca/serg/papers/crl361.pdf (accessed 10 June 2013; page references are to the online pre-print); a paper with the same title (but American spelling as 'program') appeared in *IEEE Software* 16(6) (1990): 19–30.

[Parsons, 2015] Parsons, K. M. (2015). What is the public value of philosophy? *Huffington Post*. http://www.huffingtonpost.com/keith-m-parsons/what-is-the-public-value-of-philosophy_b_7018022.html (accessed 13 May 2015).

[Pawley, 2009] Pawley, A. L. (2009). Universalized narratives: Patterns in how faculty members define 'engineering'. *Journal of Engineering Education*, 98(4):309–319.

[Peacocke, 1995] Peacocke, C. (1995). Content, computation and externalism. *Philosophical Issues*, 6:227–264.

[Peacocke, 1999] Peacocke, C. (1999). Computation as involving content: A response to Egan. *Mind & Language*, 14(2):195–202.

[Peano, 1889] Peano, G. (1889). The principles of arithmetic, presented by a new method. In van Heijenoort, J., editor, *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*, pages 83–97. Harvard University Press, Cambridge, MA.

[Pearl, 2000] Pearl, J. (2000). *Causality: Models, Reasoning, and Inference*. Cambridge University Press, Cambridge, UK.

[Pelletier, 1999] Pelletier, F. J. (1999). A history of natural deduction and elementary logic textbooks. *History and Philosophy of Logic*, 20:1–31. Accessed 17 February 2014 from: http://www.sfu.ca/~jeffpell/papers/NDHistory.pdf.

[Pennisi, 2013] Pennisi, E. (2013). Nervous system may have evolved twice. *Science*, 339(6118):391.

[Perlis, 1962] 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.

[Perruchet and Vinter, 2002] Perruchet, P. and Vinter, A. (2002). The self-organizing consciousness. *Behavioral and Brain Sciences*, 25(3):297–388.

[Perry, 1981] Perry, William G., J. (1981). Cognitive and ethical growth: The making of meaning. In Chickering, A. and Associates, editors, *The Modern American College*, pages 76–116. Jossey-Bass, San Francisco.

[Perry, 1970] Perry, Jr., W. G. (1970). *Forms of Intellectual and Ethical Development in the College Years: A Scheme*. Holt, Rinehart and Winston, New York.

[Peschl and Scheutz, 2001a] Peschl, M. and Scheutz, M. (2001a). Some thoughts on computation and simulation in cognitive science. In *Proceedings of the 6th Congress of the Austrian Philosophical Society*, pages 534–540. publisher unknown. http://hrilab.tufts.edu/publications/scheutzpeschl00linz.pdf (accessed 5 January 2015; citations to the online version).

[Peschl and Scheutz, 2001b] Peschl, M. F. and Scheutz, M. (2001b). Explicating the epistemological role of simulation in the development of theories of cognition. In *Proceedings of the 7th Colloquium on Cognitive Science (ICCS-01)*, pages 274–280. publisher unknown. http://www.academia.edu/719102/Explicating_the_epistemological_role_of_simulation_in_the_development_of_theories_of_cognition (accessed 5 January 2015).

[Petersen, 2007] Petersen, S. (2007). The ethics of robot servitude. *Journal of Experimental and Theoretical Artificial Intelligence*, 19(1):43–54.

[Petroski, 2003] Petroski, H. (2003). Early education. *American Scientist*, 91:206–209. online at:
http://childrensengineering.org/resources/Petroski.pdf (accessed 21 October 2013).

[Petroski, 2005] Petroski, H. (2005). Technology and the humanities. *American Scientist*, 93:304–307.

[Petroski, 2006] Petroski, H. (2006). A great profession. *American Scientist*, 94:304–307. online at:
http://www.wsechicago.org/2006-07Petroski.pdf (accessed 21 October 2013).

[Petzold, 2008] Petzold, C. (2008). *The Annotated Turing: A Guided Tour through Alan Turing's Historic Paper on Computability and the Turing Machine*. Wiley, Indianapolis.

[Picard, 1997] Picard, R. (1997). *Affective Computing*. MIT Press, Cambridge, MA.

[Piccinini, 2003] Piccinini, G. (2003). Alan turing and the mathematical objection. *Minds and Machines*, 13:23–48. http://www.umsl.edu/∼piccininig/Alan_Turing_and_Mathematical_Objection.pdf (accessed 13 February 2015).

[Piccinini, 2004a] Piccinini, G. (2004a). The first computational theory of mind and brain: A close look at McCulloch and Pitts's 'Logical calculus of ideas immanent in nervous activity'. *Synthese*, 141:175–215. For a reply, see [Aizawa, 2010].

[Piccinini, 2004b] Piccinini, G. (2004b). Functionalism, computationalism, and mental contents. *Canadian Journal of Philosophy*, 34(3):375–410. Accessed 22 April 2014 from:
http://www.umsl.edu/∼piccininig/Functionalism_Computationalism_and_Mental_Contents.pdf.

[Piccinini, 2005] Piccinini, G. (2005). Symbols, strings, and spikes: The empirical refutation of computationalism. Accessed 24 March 2011 from
[http://www.umsl.edu/∼piccininig/Symbols

[Piccinini, 2006a] Piccinini, G. (2006a). Computation without representation. *Philosophical Studies*, 137(2):204–241. Accessed 29 April 2014 from:
http://www.umsl.edu/∼piccininig/Computation_without_Representation.pdf.

[Piccinini, 2006b] Piccinini, G. (2006b). Computational explanation in neuroscience. *Synthese*, 153(3):343–353. Accessed 31 January 2014 from:
http://www.umsl.edu/∼piccininig/Computational%20Explanation%20in%20Neuroscience.pdf.

[Piccinini, 2007a] Piccinini, G. (2007a). Computational explanation and mechanistic explanation of mind. In Marraffa, M., Caro, M. D., and Ferretti, F., editors, *Cartographies of the Mind: Philosophy and Psychology in Intersection*, pages 23–36. Springer, Dordrecht, The Netherlands.

[Piccinini, 2007b] Piccinini, G. (2007b). Computational modelling vs. computational explanation: Is everything a turing machine, and does it matter to the philosophy of mind? *Australasian Journal of Philosophy*, 85(1):93–115. Accessed 31 January 2014 from: http://www.umsl.edu/∼piccininig/Is_Everything_a_TM.pdf.

[Piccinini, 2007c] Piccinini, G. (2007c). Computing mechanisms. *Philosophy of Science*, 74(4):501–526.

[Piccinini, 2008] Piccinini, G. (2008). Computers. *Pacific Philosophical Quarterly*, 89:32–73.

[Piccinini, 2009] Piccinini, G. (2009). Computationalism in the philosophy of mind. *Philosophy Compass*, 4(3):515–532. 10.1111/j.1747-9991.2009.00215.x.

[Piccinini, 2010a] Piccinini, G. (2010a). The mind as neural software? understanding functionalism, computationalism, and computational functionalism. *Philosophy and Phenomenological Research*, 81(2):269–311.

[Piccinini, 2010b] Piccinini, G. (2010b). The resilience of computionalism. *Philosophy of Science*, 77(5):852–861.

[Piccinini, 2011] Piccinini, G. (2011). The physical Church-Turing thesis: Modest or bold? *British Journal for the Philosophy of Science*, 62:733–769.

[Piccinini, 2012] Piccinini, G. (2012). Computationalism. In Margolis, E., Samuels, R., and Stich, S. P., editors, *The Oxford Handbook of Philosophy of Cognitive Science*. Oxford University Press. 10.1093/oxfordhb/9780195309799.001.0001.

[Piccinini and Craver, 2011] Piccinini, G. and Craver, C. (2011). Integrating psychology and neuroscience: Functional analyses as mechanism sketches. *Synthese*, 183(3):283–311. preprint online at:
http://philosophy.artsci.wustl.edu/files/philosophy/imce/integrating_psychology_and_neuroscience_functional_analyses_as_mechanism_sketches_0.pdf
(accessed 29 July 2013).

[Piccinini and Scarantino, 2011] Piccinini, G. and Scarantino, A. (2011). Information processing, computation, and cognition. *Journal of Biological Physics*, 37:1–38.

[Pincock, 2011] Pincock, C. (2011). Philosophy of mathematics. In Saatsi, J. and French, S., editors, *Companion to the Philosophy of Science*, pages 314–333. Continuum. Pre-print accessed 31 March 2014 from:
http://pincock-yilmazer.com/chris/pincock%20philosophy%20of%20mathematics%20final%20draft.pdf.

[Pinker and Jackendoff, 2005] Pinker, S. and Jackendoff, R. (2005). The faculty of language: What's special about it? *Cognition*, 95:201–236. Accessed 14 July 2014 from:
http://pinker.wjh.harvard.edu/articles/papers/2005_03_Pinker_Jackendoff.pdf.

[Plato, 1961a] Plato (1961a). Phaedrus. In Hamilton, E. and Cairns, H., editors, *The Collected Dialogues of Plato, including the Letters*. Princeton University Press, Princeton, NJ.

[Plato, 1961b] Plato (1961b). Republic. In Hamilton, E. and Cairns, H., editors, *The Collected Dialogues of Plato, including the Letters*, pages 575–844. Princeton University Press, Princeton, NJ.

[Polger, 2011] Polger, T. W. (2011). Are sensations still brain processes? *Philosophical Psychology*, 24(1):1–21.

[Pollan, 2013] Pollan, M. (2013). The intelligent plant. *The New Yorker*, pages 92–105. Accessed 31 January 2014 from: http://www.newyorker.com/reporting/2013/12/23/131223fa_fact_pollan.

[PolR, 2009] PolR (11 November 2009). An explanation of computation theory for lawyers. http://www.groklaw.net/article.php?story=20091111151305785. also at: http://www.groklaw.net/staticpages/index.php?page=200911110152507492 and at: http://www.groklaw.net/pdf2/ComputationalTheoryforLawyers.pdf (accessed 16 December 2013).

[Popova, 2012] Popova, M. (2012). What is philosophy? an omnibus of definitions from prominent philosophers. http://www.brainpickings.org/index.php/2012/04/09/what-is-philosophy/. (accessed 6 February 2013).

[Popper, 1978] Popper, K. (1978). Three worlds. http://tannerlectures.utah.edu/_documents/a-to-z/p/popper80.pdf. Accessed 20 February 2015.

[Post, 1941] Post, E. L. (1941). Absolutely unsolvable problems and relatively undecidable propositions: Account of an anticipation. In Davis, M., editor, *Solvability, Provability, Definability: The Collected Works of Emil L. Post*, pages 375–441. Birkhaäuser, Boston.

[Post, 1943] Post, E. L. (1943). Formal reductions of the general combinatorial decision problem. *American Journal of Mathematics*, 65:197–215.

[Pour-El, 1974] Pour-El, M. B. (1974). Abstract computability and its relation to the general purpose analog computer (some connections between logic, differential equations and analog computers). *Transactions of the American Mathematical Society*, 199:1–28.

[Prasse and Rittgen, 1998] Prasse, M. and Rittgen, P. (1998). Why church's thesis still holds. some notes on peter wegner's tracts on interaction and computability. *The Computer Journal*, 41(6):357–362. Accessed 3 June 2014 from: http://research.cs.queensu.ca/∼akl/cisc879/papers/SELECTED_PAPERS_FROM_VARIOUS_SOURCES/Prasse.pdf.

[Preston, 2006] Preston, B. (2006). The centralized control model of production. from *The Stuff of Life: Towards a Philosophy of Material Culture*, Ch. 1.

[Preston, 2012] Preston, J. (2012). Paul feyerabend. In Zalta, E. N., editor, *Stanford Encyclopedia of Philosophy*. online. http://plato.stanford.edu/archives/win2012/entries/feyerabend/ (accessed 23 July 2013).

[Proudfoot and Copeland, 2012] Proudfoot, D. and Copeland, B. J. (2012). Artificial intelligence. In Margolis, E., Samuels, R., and Stich, S. P., editors, *The Oxford Handbook of Philosophy of Cognitive Science*. Oxford University Press.

[Putnam, 1960] Putnam, H. (1960). Minds and machines. In Hook, S., editor, *Dimensions of Mind: A Symposium*, pages 148–179. New York University Press, New York.

[Putnam, 1965] Putnam, H. (1965). Trial and error predicates and the solution to a problem of mostowski. *Journal of Symbolic Logic*, 30(1):49–57. Accessed 5 February 2014 from: http://www.ninagierasimczuk.com/flt2013/wp-content/uploads/2013/01/Putnam_1965.pdf.

[Putnam, 1975] Putnam, H. (1975). The meaning of 'meaning'. In Gunderson, K., editor, *Minnesota Studies in the Philosophy of Science, Vol. 7: Language, Mind, and Knowledge*, pages 131–193. University of Minnesota Press, Minneapolis. reprinted in Hilary Putnam, *Mind, Language and Reality* (Cambridge, UK: Cambridge University Press): 215–271; accessed 13 February 2014 from: http://mcps.umn.edu/assets/pdf/7.3_Putnam.pdf.

[Putnam, 1994] Putnam, H. (1994). Sense, nonsense, and the senses: An inquiry into the powers of the human mind. *Journal of Philosophy*, 91(9):445–517.

[Putnam, 2001] Putnam, H. (2001). Skepticism, stroud and the contextuality of knowledge. *Philosophical Explorations: An International Journal for the Philosophy of Mind and Action*, 4(1):2–16.

[Pylyshyn, 1984] Pylyshyn, Z. W. (1984). *Computation and Cognition: Towards a Foundation for Cognitive Science*. MIT Press, Cambridge, MA. Ch. 3 ("The Relevance of Computation"), pp. 48–86, esp. the section "The Role of Computer Implementation" (pp. 74–78).

[Pylyshyn, 1992] Pylyshyn, Z. W. (1992). Computers and the symbolization of knowledge. In Morelli, R., Brown, W. M., Anselmi, D., Haberlandt, K., and Lloyd, D., editors, *Minds, Brains & Computers: Perspectives in Cognitive Science and Artificial Intelligence*, pages 82–94. Ablex, Norwood, NJ. Preprint (1987) accessed 9 April 2014 from:
http://ruccs.rutgers.edu/ftp/pub/papers/suffolk.pdf
(page references are to the preprint).

[Qian and Winfree, 2011] Qian, L. and Winfree, E. (2011). Scaling up digital circuit computation with dna strand displacement cascades. *Science*, 332:1196–1201. See review in [Reif, 2011].

[Quillian, 1994] Quillian, M. R. (1994). A content-independent explanation of science's effectivenss. *Philosohy of Science*, 61(3):429–448.

[Quine, 1948] Quine, W. v. O. (1948). On what there is. *Review of Metaphysics*, 2(5):21–38. reprinted in W.V.O. Quine, *From a Logical Point of View: 9 Logico-Philosophical Essays, Second Edition, revised* (Cambridge, MA: Harvard University Press, 1980): 1–19.

[Quine, 1951] Quine, W. v. O. (1951). Two dogmas of empiricism. *Philosophical Review*, 60:20–43.

[Quine, 1988] Quine, W. v. O. (1988). Letter from quine to chris hookway. In Hilary Putnam's *Sardonic Comment* blog. Accessed 14 July 2014 from: http://putnamphil.blogspot.com/2014/07/a-letter-1988-from-quine-to-chris.html.

[Radó, 1962] Radó, T. (1962). On non-computable functions. *The Bell System Technical Journal*, pages 877–884. http://alcatel-lucent.com/bstj/vol41-1962/articles/bstj41-3-877.pdf (accessed 16 December 2013).

[Ralston, 1999] Ralston, A. (1999). Let's abolish pencil-and-paper arithmetic. *Journal of Computers in Mathematics and Science Teaching*, 18(2):173–194. Preprint accessed 17 April 2014 from: http://tonyralston.com/papers/abolpub.htm.

[Ramsey, 1929] Ramsey, F. P. (1929). Theories. In Mellor, D., editor, *Foundations: Essays in Philosophy, Logic, Mathematics and Economics; revised edition*, pages 101–125. Humanities Press, 1978, Atlantic Highlands, NJ.

[Randell, 1994] Randell, B. (1994). The origins of computer programming. *IEEE Annals of the History of Computing*, 16(4):6–14. online at: https://www.researchgate.net/publication/3330487_The_origins_of_computer_programming/ (accessed 11 November 2013).

[Rapaport, 1978] Rapaport, W. J. (1978). Meinongian theories and a russellian paradox. *Noûs*, 12(2):153–180. Errata, *Noûs* 13 (1979): 125.

[Rapaport, 1982] Rapaport, W. J. (1982). Unsolvable problems and philosophical progress. *American Philosophical Quarterly*, 19:289–298.

[Rapaport, 1984] Rapaport, W. J. (1984). Critical thinking and cognitive development. *American Philosophical Association Newsletter on Pre-College Instruction in Philosophy*, 1:4–5. Reprinted in *Proceedings and Addresses of the American Philosophical Association* 57(5) (May 1984): 610–615; http://www.cse.buffalo.edu/~rapaport/Papers/rapaport84-perryAPA.pdf (accessed 13 January 2015).

[Rapaport, 1986] Rapaport, W. J. (1986). Logical foundations for belief representation. *Cognitive Science*, 10:371–422.

[Rapaport, 1988] 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. Reprinted with numerous errors in Eric Dietrich (ed.) (1994), *Thinking Machines and Virtual Persons: Essays on the Intentionality of Machines* (San Diego: Academic Press): 225–273.

[Rapaport, 1995] Rapaport, W. J. (1995). Understanding understanding: Syntactic semantics and computational cognition. In Tomberlin, J. E., editor, *AI, Connectionism, and Philosophical Psychology*, pages 49–88. Ridgeview, Atascadero, CA. *Philosophical Perspectives*, Vol. 9; reprinted in Toribio, Josefa, & Clark, Andy (eds.) (1998), *Language and Meaning in Cognitive Science: Cognitive Issues and Semantic Theory*, *Artificial Intelligence and Cognitive Science: Conceptual Issues*, Vol. 4 (New York: Garland): 73–88.

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

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

[Rapaport, 2000] Rapaport, W. J. (2000). Cognitive science. In Ralston, A., Reilly, E. D., and Hemmendinger, D., editors, *Encyclopedia of Computer Science, 4th edition*, pages 227–233. Grove's Dictionaries, New York.

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

[Rapaport, 2005a] Rapaport, W. J. (2005a). Castañeda, hector-neri. In Shook, J. R., editor, *The Dictionary of Modern American Philosophers, 1860–1960*, pages 452–412. Thoemmes Press, Bristol, UK.

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

[Rapaport, 2007] Rapaport, W. J. (2007). Searle on brains as computers. *American Philosophical Association Newsletter on Philosophy and Computers*, 6(2):4–9. Accessed 31 January 2014 from: http://c.ymcdn.com/sites/www.apaonline.org/resource/collection/EADE8D52-8D02-4136-9A2A-729368501E43/v06n2Computers.pdf.

[Rapaport, 2009] Rapaport, W. J. (2009). Inten_s_ionality vs. inten_t_ionality. http://www.cse.buffalo.edu/∼rapaport/intensional.html, accessed 30 September 2011.

[Rapaport, 2011] Rapaport, W. J. (2011). A triage theory of grading: The good, the bad, and the middling. *Teaching Philosophy*, 34(4):347–372.

[Rapaport, 2012a] Rapaport, W. J. (2012a). Inten*s*ionality vs. inten*t*ionality. http://www.cse.buffalo.edu/∼rapaport/intensional.html.

[Rapaport, 2012b] Rapaport, W. J. (2012b). Semiotic systems, computers, and the mind: How cognition could be computing. *International Journal of Signs and Semiotic Systems*, 2(1):32–71.

[Rapaport, 2013] Rapaport, W. J. (2013). How to write. Accessed 4 February 2014 from: http://www.cse.buffalo.edu/∼rapaport/howtowrite.html.

[Rapaport, 2014] Rapaport, W. J. (2014). How to study: A brief guide. http://www.cse.buffalo.edu/∼rapaport/howtostudy.html. Accessed 19 January 2014.

[Rapaport, 2015] Rapaport, W. J. (2015). On the relation of computing to the world. Current draft in progress at
http://www.cse.buffalo.edu/∼rapaport/Papers/covey.pdf.

[Rapaport and Kibby, 2010] Rapaport, W. J. and Kibby, M. W. (2010). Contextual vocabulary acquisition: From algorithm to curriculum. http://www.cse.buffalo.edu/∼rapaport/CVA/reading4CgSJnl.pdf (accessed 13 April 2013).

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

[Reese, 2014a] Reese, H. (18 February 2014a). The joy of teaching computer science in the age of facebook. *The Atlantic*. Accessed 13 March 2014 from: http://www.theatlantic.com/education/archive/2014/02/the-joy-of-teaching-computer-science-in-the-age-of-facebook/283879/.

[Reese, 2014b] Reese, H. (27 February 2014b). Why study philosophy? 'to challenge your own point of view'. *The Atlantic*. Accessed 12 March 2014 from: http://www.theatlantic.com/education/archive/2014/02/why-study-philosophy-to-challenge-your-own-point-of-view/283954/.

[Reif, 2011] Reif, J. H. (2011). Scaling up dna computation. *Science*, 332:1156–1157. Review of [Qian and Winfree, 2011].

[Rescorla, 2007] Rescorla, M. (2007). Church's thesis and the conceptual analysis of computability. *Notre Dame Journal of Formal Logic*, 48(2):253–280. Preprint (accessed 29 April 2014) at
http://www.philosophy.ucsb.edu/people/profiles/faculty/cvs/papers/church2.pdf.

[Rescorla, 2012a] Rescorla, M. (2012a). Are computational transitions sensitive to semantics? *Australian Journal of Philosophy*, 90(4):703–721. Preprint at http://www.philosophy.ucsb.edu/docs/faculty/papers/formal.pdf
(accessed 30 October 2014).

[Rescorla, 2012b] Rescorla,
M. (2012b). How to integrate representation into computational modeling, and why we should. *Journal of Cognitive Science (South Korea)*, 13(1):1–38. http://j-cs.org/gnuboard/bbs/board.php?bo_table=__vol013i1&wr_id=1
(accessed 9 November 2014).

[Rescorla, 2013] Rescorla, M. (2013). Against structuralist theories of computational implementation. *British Journal for the Philosophy of Science*, 64(4):681–707. Preprint (accessed 31 October 2014) at
http://philosophy.ucsb.edu/docs/faculty/papers/against.pdf.

[Rescorla, 2014a] Rescorla, M. (2014a). The causal relevance of content to computation. *Philosophy and Phenomenological Research*, 88(1):173–208. Preprint at
http://www.philosophy.ucsb.edu/people/profiles/faculty/cvs/papers/causalfinal.pdf
(accessed 7 May 2014).

[Rescorla, 2014b] Rescorla, M. (2014b). A theory of computational implementation. *Synthese*, 191:1277–1307. Preprint accessed 3 November 2014 from:
http://philosophy.ucsb.edu/docs/faculty/papers/implementationfinal.pdf.

[Richards, 1988] Richards, W. (1988). *Natural Computation*. MIT Press, Cambridge, MA.

[Roberts, 1996] Roberts, E. (1996). Computer science user's guide. http://web.archive.org/web/19970609072402/http://www-cse.stanford.edu/resources/users_guide/1.html. (accessed 21 June 2013).

[Roberts, 2006] Roberts, E. S. (2006). *The Art and Science of Java*. Stanford University,
Stanford, CA. http://people.reed.edu/∼jerry/121/materials/artsciencejava.pdf, accessed 20 September 2011.

[Robertson, 2003] Robertson, D. S. (2003). *Phase Change: The Computer Revolution in Science and Mathematics*. Oxford University Press.

[Robertson, 1979] Robertson, J. I. (1979). How to do arithmetic. *American Mathematical Monthly*, 86:431–439.

[Robinson, 1994] 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. Accessed 19 February 2014 from: http://staffweb.worc.ac.uk/DrC/Courses%202013-14/COMP3202/Reading%20Materials/ robinson94.pdf.

[Rohlf, 2010] Rohlf, M. (2010). Immanuel kant. In Zalta, E. N., editor, *Stanford Encyclopedia of Philosophy (Fall 2010 Edition)*. online. http://plato.stanford.edu/archives/fall2010/entries/kant/
(accessed 15 November 2013).

[Rosch, 1978] Rosch, E. (1978). Principles of categorization. In Rosch, E. and Lloyd, B. B., editors, *Cognition and Categorization*, pages 27–48. Lawrence Erlbaum Associates, Hillsdale, NJ.

[Rosch and Mervis, 1975] Rosch, E. and Mervis, C. B. (1975). Family resemblances: Studies in the internal structure of categories. *Cognitive Psychology*, 7:573–605.

[Rosenberg, 1994] Rosenberg, A. (1994). If economics isn't science, what is it? In Hausman, D. M., editor, *The Philosophy of Economics: An Anthology, Second Edition*, pages 376–394. Cambridge University Press, New York. online at http://philoscience.unibe.ch/documents/kursarchiv/SS07/Rosenberg1994.pdf (accessed 4 November 2013).

[Rosenblueth and Wiener, 1945] Rosenblueth, A. and Wiener, N. (1945). The role of models in science. *Philosophy of Science*, 12:316–321.

[Ross, 1974] Ross, D. (1974). Church's thesis: What its difficulties are and are not. *Journal of Philosophy*, 71(15):515–525.

[Rosser, 1939] Rosser, J. B. (1939). An informal exposition of proofs of gödel's theorems and church's theorem. *Journal of Symbolic Logic*, 4(2):53–60.

[Rosser, 1978] Rosser, J. B. (1978). *Logic for Mathematicians: Second Edition*. Dover Publications, Mineola, NY. First edition (1953) online at: https://archive.org/details/logicformathemat00ross (accessed 5 March 2015).

[Roth, 1983] Roth, P. (1983). Simulation. In Ralston, A. and Edwin D.Ẽeilly, J., editors, *Encyclopedia of Computer Science and Engineering, 2nd edition*, pages 1327–1341. Van Nostrand Reinhold, New York.

[Royce, 1900] Royce, J. (1900). *The World and the Individual*. Macmillan, London.

[Rubinoff, 1953] Rubinoff, M. (1953). Analogue vs. digital computers—a comparison. *Proceedings of the IRE*, 41(10):1254–1262.

[Rupp, 2003] Rupp, N. A. (9 August 2003). Computer science is philosophy. https://weblogs.java.net/blog/n_alex/archive/2003/08/computer_scienc.html. (accessed 21 June 2013).

[Russell, 1917] Russell, B. (1917). *Mysticism and Logic and Other Essays*. George Allen & Unwin, London. http://archive.org/details/mysticism00russuoft.

[Russell, 1936] Russell, B. (1936). The limits of empiricism. *Proceedings of the Aristotelian Society, New Series*, 36:131–150.

[Russell, 1995] Russell, S. (1995). Rationality and intelligence. *Artificial Intelligence*, 94:57–77. online at: http://www.cs.berkeley.edu/∼russell/papers/aij-cnt.pdf (accessed 18 September 2013).

[Sacks, 1999] Sacks, O. (1999). *Migraine: Revised and Expanded*. Random House/Vintage Books, New York.

[Salter, 2010] Salter, J. (2010). The art of the ditch. *New York Review of Books*, 57(1). http://www.nybooks.com/articles/archives/2010/jan/14/the-art-of-the-ditch/ (accessed 14 November 2012).

[Samuel, 1953] Samuel, A. L. (1953). Computing bit by bit, or digital computers made easy. *Proceedings of the IRE*, 41(10):1223–1230.

[Samuelson, 1989] Samuelson, P. (1989). Why the look and feel of software user interfaces should not be protected by copyright law. *Communications of the ACM*, 32(5):563–572. Accessed 8 September 2014 from: http://www.foo.be/andria/docs/p563-samuelson.pdf.

[Samuelson et al., 1994] 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. Accessed 19 September 2014 from: http://scholarship.law.duke.edu/cgi/viewcontent.cgi?article=1783&context=faculty_scholarship. From a special issue on the legal protection of computer programs; other articles elaborate on, or reply to, Samuelson et al. §1 is a good overview; §2 (esp. §2.2) is also good, as are §5 and the Conclusion section. A summary version appears as [Davis et al., 1996].

[Samuleson, 1991] Samuleson, P. (1991). Digital media and the law. *Communications of the ACM*, 34(10):23–28.

[Sanford, 1967] Sanford, N. (1967). *Where Colleges Fail*. Jossey-Bass, San Francisco.

[Saussure, 1959] Saussure, F. d. (1959). *Course in General Linguistics*. Philosophical Library, New York. Charles Bally, Albert Sechehaye, & Albert Reidlinger (eds.).

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

[Schank, 1983] Schank, R. C. (1983). The current state of ai: One man's opinion. *AI Magazine*, 4(1):3–8.

[Scherlis and Scott, 1983] Scherlis, W. L. and Scott, D. S. (1983). First steps towards inferential programming. In Mason, R., editor, *Information Processing 83*, pages 199–212. Elsevier North-Holland. Technical-report version accessed 27 July 2014 from: http://repository.cmu.edu/cgi/viewcontent.cgi?article=3542&context=compsci.

[Scheutz, 2001] Scheutz, M. (2001). Computational versus causal complexity. *Minds and Machines*, 11:543–566. Preprint accessed 23 October 2014 from: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.21.8293&represent=rep1&type=pdf.

[Scheutz, 2012] Scheutz, M. (2012). What it is not to implement a computation: A critical analysis of chalmers' notion of implemention. *Journal of Cognitive Science (South Korea)*, 13(1):75–106. http://j-cs.org/gnuboard/bbs/board.php?bo_table=__vol013i1&wr_id=4 (accessed 9 November 2014).

[Schwitzgebel, 2012] Schwitzgebel, E. (9 January 2012). For all x, there's philosophy of x. http://schwitzsplinters.blogspot.com/2012/01/for-all-x-theres-philosophy-of-x.html (accessed 12 February 2013).

[Schyns et al., 2008] Schyns, P. G., Goddelin, F., and Smith, M. L. (2008). Information processing algorithms in the brain. *Trends in Cognitive Sciences*, 13(1):20–26. Accessed 31 March 2014 from http://www.mapageweb.umontreal.ca/gosselif/schynsetal_TICS.pdf.

[Seabrook, 2007] Seabrook, J. (2007). Fragmentary knowledge. *The New Yorker*, pages 94–102.

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

[Searle, 1982] Searle, J. R. (1982). The myth of the computer. *New York Review of Books*, pages 3–6. Cf. correspondence, same journal, 24 June 1982, pp. 56–57.

[Searle, 1983] Searle, J. R. (1983). *Intentionality: An Essay in the Philosophy of Mind*. Cambridge University Press, Cambridge, UK.

[Searle, 1990] Searle, J. R. (1990). Is the brain a digital computer? *Proceedings and Addresses of the American Philosophical Association*, 64(3):21–37.

[Searle, 1995] Searle, J. R. (1995). *The Construction of Social Reality*. Free Press, New York.

[Sellars, 1962] Sellars, W. (1962). Philosophy and the scientific image of man. In Colodny, R., editor, *Frontiers of Science and Philosophy*, pages 35–78. University of Pittsburgh Press, Pittsburgh. reprinted in Sellars, Wilfrid (1963), *Science, Perception and Reality* (London: Routledge & Kegan Paul); online at: http://www.ditext.com/sellars/psim.html (accessed 12 February 2013).

[Shadmehr and Wise, 2005] Shadmehr, R. and Wise, S. (2005). *The Computational Neurobiology of Reaching and Pointing: A Foundation for Motor Learning*. MIT Press, Cambridge, MA.

[Shagrir, 1999] Shagrir, O. (1999). What is computer science about? *The Monist*, 82(1):131–149.

[Shagrir, 2006a] Shagrir, O. (2006a). Gödel on turing on computability. In Olszewski, A., Woł enski, J., and Janusz, R., editors, *Church's Thesis after 70 Years*, pages 393–419. Ontos-Verlag. http://edelstein.huji.ac.il/staff/shagrir/papers/Goedel_on_Turing_on_Computability.pdf (accessed 11 December 2013); page references are to the online version.

[Shagrir, 2006b] Shagrir, O. (2006b). Why we view the brain as a computer. *Synthese*, 153:393–416. Preprint at http://edelstein.huji.ac.il/staff/shagrir/papers/Why_we_view_the_brain_as_a_computer.pdf (accessed 25 March 2014).

[Shagrir, 2010] Shagrir, O. (2010). Brains as analog-model computers. *Studies in History and Philosophy of Science*, 41(3):271–279.

[Shagrir, 2012] Shagrir, O. (2012). Can a brain possess two minds? *Journal of Cognitive Science (South Korea)*, 13(2):145–165. http://j-cs.org/gnuboard/bbs/board.php?bo_table=__vol013i2&wr_id=2 (accessed 9 November 2014).

[Shannon, 1937] Shannon, C. E. (1937). A symbolic analysis of relay and switching circuits. Technical report, MIT Department of Electrical Engineering, Cambridge, MA. MS thesis, 1940; http://hdl.handle.net/1721.1/11173.

[Shannon, 1950] Shannon, C. E. (1950). Programming a computer for playing chess. *Philosophical Magazine, Ser. 7*, 41(314). online at: http://vision.unipv.it/IA1/ProgrammingaComputerforPlayingChess.pdf (accessed 5 March 2013).

[Shapiro and Benenson, 2006] Shapiro, E. and Benenson, Y. (2006). Bringing dna computers to life. *Scientific American*, 294(5):44–51. Accessed 24 March 2014 from http://www.wisdom.weizmann.ac.il/~udi/papers/ShapiroBenensonMay06.pdf.

[Shapiro, 1985] Shapiro, F. R. (1985). Debugging etymologies. *New York Times*. letter to the editor; online at: http://www.nytimes.com/1985/03/24/books/l-debugging-etymologies-114833.html (accessed 7 November 2013).

[Shapiro, 2000a] Shapiro, F. R. (2000a). Origin of the term software: Evidence form the jstor electronic journal archive. *IEEE Annals of the History of Computing*, 22(2):69–71.

[Shapiro, 1993] Shapiro, S. (1993). Understanding church's thesis, again. *Acta Analytica*, 11:59–77.

[Shapiro, 1996] Shapiro, S. (1996). Space, number and structure: A tale of two debates. *Philosophia Mathematica*, 4(2):148–173.

[Shapiro, 2000b] Shapiro, S. (2000b). Remarks on the development of computability. *History and Philosophy of Logic*, 4(1):203–220.

[Shapiro, 2009] Shapiro, S. (2009). We hold these truths to be self-evident: But what do we mean by that? *Review of Symbolic Logic*, 2(1):175–207. Accessed 19 February 2014 from: http://www.pgrim.org/philosophersannual/29articles/shapirowehold.pdf.

[Shapiro, 1992] Shapiro, S. C. (1992). Artificial intelligence. In Shapiro, S. C., editor, *Encyclopedia of Artificial Intelligence, 2nd edition*, pages 54–57. John Wiley & Sons, New York.

[Shapiro, 2001] Shapiro, S. C. (2001). Computer science: The study of procedures. Technical report, Department of Computer Science and Engineering, University at Buffalo, Buffalo, NY. Accessed 30 September 2011 from: http://www.cse.buffalo.edu/~shapiro/Papers/whatiscs.pdf.

[Shapiro and Rapaport, 1987] 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 and Rapaport, 1991] 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.

[Shapiro et al., 1992] Shapiro, S. C., Srihari, S., and Jayaraman, B. (December 1992). email discussion. http://www.cse.buffalo.edu/~rapaport/scs.txt.

[Shelley, 1818] Shelley, M. W. (1818). *Frankenstein; or, the Modern Prometheus*. online. http://www.literature.org/authors/shelley-mary/frankenstein/.

[Shepherdson and Sturgis, 1963] Shepherdson, J. and Sturgis, H. (1963). Computability of recursive functions. *Journal of the ACM*, 10(2):217–255.

[Sheraton, 1981] 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 (accessed 13 November 2013).

[Sieg, 1994] Sieg, W. (1994). Mechanical procedures and mathematical experience. In George, A., editor, *Mathematics and Mind*, pages 71–117. Oxford University Press, New York. http://repository.cmu.edu/cgi/viewcontent.cgi?article=1248&context=philosophy (accessed 10 December 2013).

[Sieg, 2000] Sieg, W. (2000). Calculations by man and machine: Conceptual analysis. Technical Report CMU-PHIL-104, Carnegie-Mellon University Department of Philosophy, Pittsburgh, PA. http://repository.cmu.edu/philosophy/178.

[Sieg, 2006] Sieg, W. (2006). Gödel on computability. *Philosophia Mathematica*, 14:189–207. page references to preprint version, accessed 6 November 2012 from: http://www.hss.cmu.edu/philosophy/sieg/A%20G_del%20on%20Computability.pdf.

[Simon, 1962] Simon, H. A. (1962). The architecture of complexity. *Proceedings of the American Philosophical Society*, 106(6):467–482. reprinted as Ch. 8 of Simon 1996.

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

[Simon and Newell, 1958] Simon, H. A. and Newell, A. (1958). Heuristic problem solving: The next advance in operations research. *Operations Research*, 6(1):1–10.

[Simon and Newell, 1962] Simon, H. A. and Newell, A. (1962). Simulation of human thinking. In Greenberger, M., editor, *Computers and the World of the Future*, pages 94–114. MIT Press, Cambridge, MA.

[Skidelsky, 2014] Skidelsky, R. (2014). The programmed prospect before us. *New York Review of Books*, 61(6):35–37. Accessed 28 April 2014 from: http://www.skidelskyr.com/site/article/the-programmed-prospect-before-us/.

[Skow, 2007] Skow, B. (2007). Are shapes intrinsic? *Philosophical Studies*, 133:111–130.

[Slagle, 1971] Slagle, J. R. (1971). *Artificial Intelligence: The Heuristic Programming Approach*. McGraw-Hill, New York.

[Sloman, 1998] Sloman, A. (1998). Supervenience and implementation: Virtual and physical machines. http://www.cs.bham.ac.uk/research/projects/cogaff/Sloman.supervenience.and.implementation.pdf. Accessed 21 October 2014.

[Sloman, 2002] Sloman, A. (2002). The irrelevance of Turing machines to AI. In Scheutz, M., editor, *Computationalism: New Directions*, pages 87–127. MIT Press, Cambridge, MA. http://www.cs.bham.ac.uk/research/projects/cogaff/sloman.turing.irrelevant.pdf (accessed 21 February 2014). Page references are to the online preprint.

[Sloman, 1989] Sloman, A. (6 August 1989). Contribution to newsgroup discussion of "is there a definition of ai? Article 4702 of comp.ai. http://www.cse.buffalo.edu/~rapaport/defs.txt.

[Sloman, 2010] Sloman, A. (7 January 2010). Why symbol-grounding is both impossible and unnecessary, and why symbol-tethering based on theory-tethering is more powerful anyway. http://www.cs.bham.ac.uk/research/cogaff/talks/#models.

[Sloman and Croucher, 1981] Sloman, A. and Croucher, M. (1981). Why robots will have emotions. In *Proceedings of IJCAI 1981*, page PGS. PUB, ADDR.

[Smith, 1985] Smith, B. C. (1985). Limits of correctness in computers. *ACM SIGCAS Computers and Society*, 14–15(1–4):18–26. Also published as *Technical Report* CSLI-85-36 (Stanford, CA: Center for the Study of Language & Information); reprinted in Charles Dunlop & Rob Kling (eds.), *Computerization and Controversy*

(San Diego: Academic Press, 1991): 632–646; reprinted in Timothy R. Colburn, James H. Fetzer, & Terry L. Rankin (eds.), *Program Verification: Fundamental Issues in Computer Science* (Dordrecht, Holland: Kluwer Academic Publishers, 1993): 275–293.

[Smith, 1987] Smith, B. C. (1987). The correspondence continuum. Technical Report CSLI-87-71, Center for the Study of Language & Information, Stanford, CA.

[Smith, 1996] Smith, B. C. (1996). *On the Origin of Objects*. MIT Press, Cambridge, MA.

[Smith, 2002] Smith, B. C. (2002). The foundations of computing. In Scheutz, M., editor, *Computationalism: New Directions*, pages 23–58. MIT Press, Cambridge, MA.

[Smith, 2010] Smith, P. (4 December 2010). Answer to 'what good is it to study philosophy?'. AskPhilosophers.com. http://www.askphilosophers.org/question/3710 (accessed 12 February 2013).

[Soare, 2009] Soare, R. I. (2009). Turing oracle machines, online computing, and three displacements in computability theory. *Annals of Pure and Applied Logic*, 160:368–399. Preprint at http://www.people.cs.uchicago.edu/∼soare/History/turing.pdf; published version at http://ac.els-cdn.com/S0168007209000128/1-s2.0-S0168007209000128-main.pdf?_tid=8258a7e2-01ef-11e4-9636-00000aab0f6b&acdnat=1404309072_f745d1632bb6fdd95f711397fda63ee2 (both accessed 2 July 2014). A slightly different version appears as [Soare, 2013].

[Soare, 2012] Soare, R. I. (2012). Formalism and intuition in computability. *Philosophical Transactions of the Royal Society A*, 370:3277–3304. doi:10.1098/rsta.2011.0335.

[Soare, 2013] Soare, R. I. (2013). Interactive computing and relativized computability. In Copeland, B. J., Posy, C. J., and Shagrir, O., editors, *Computability: Turing, Gödel, Church, and Beyond*, pages 203–260. MIT Press, Cambridge, MA. A slightly different version appeared as [Soare, 2009].

[Sprevak, 2010] Sprevak, M. (2010). Computation, individuation, and the received view on representation. *Studies in History and Philosophy of Science*, 41(3):260–270.

[Spruit and Tamburrini, 1991] Spruit, L. and Tamburrini, G. (1991). Reasoning and computation in leibniz. *History and Philosophy of Logic*, 12:1–14.

[Squires, 1970] Squires, R. (1970). On one's mind. *Philosophical Quarterly*, 20(81):347–356.

[Staples, 2014] Staples, M. (2014). Critical rationalism and engineering: Ontology. *Synthese*, 191(10):2255–2279. Preprint accessed 10 February 2015 from: http://www.nicta.com.au/pub?doc=7397.

[Staples, 2015] Staples, M. (2015). Critical rationalism and engineering: Methodology. *Synthese*, 192(1):337–362. Preprint at http://www.nicta.com.au/pub?doc=7747 (accessed 11 February 2015).

[Steed, 2013] Steed, S. (2013). Harnessing human intellect for computing. *Computing Research News*, 25(2). Accessed 21 July 2014 from: http://www.humancomputation.com/2013/CRN-Feb2013.pdf.

[Sternberg, 1990] Sternberg, R. J. (1990). *Metaphors of Mind: Conceptions of the Nature of Intelligence*. Cambridge University Press, New York.

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

[Stevens, 2001] Stevens, Phillip, J. (2001). Magical thinking in complementary and alternative medicine. *Skeptical Inquirer*, 25(6):32–37. http://www.csicop.org/si/show/magical_thinking_in_complementary_and_alternative_medicine/, accessed 30 September 2011.

[Stewart, 1994] Stewart, I. (1994). A subway named turing. *Scientific American*, pages 104, 106–107.

[Stewart, 2000] Stewart, I. (2000). Impossibility theorems. *Scientific American*, pages 98–99.

[Stewart, 2001] Stewart, I. (2001). Easter is a quasicrystal. *Scientific American*, pages 80, 82–83. Accessed 11 December 2013 from: http://www.whydomath.org/Reading_Room_Material/ian_stewart/2000_03.html.

[Strevens, 2013] Strevens, M. (2013). Looking into the black box. *New York Times Opinionator*. http://opinionator.blogs.nytimes.com/2013/11/24/looking-into-the-black-box (accessed 3 December 2013).

[Suber, 1988] Suber, P. (1988). What is software? *Journal of Speculative Philosophy*, 2(2):89–119. http://www.earlham.edu/~peters/writing/software.htm (accessed 21 May 2012).

[Suber, 1997a] Suber, P. (1997a). Formal systems and machines: An isomorphism. http://www.earlham.edu/~peters/courses/logsys/machines.htm.

[Suber, 1997b] Suber, P. (1997b). Turing machines ii. http://www.earlham.edu/~peters/courses/logsys/turing2.htm (accessed 21 January 2014).

[Swade, 1993] Swade, D. D. (1993). Redeeming charles babbage's mechanical computer. *Scientific American*, pages 86–91.

[Tanenbaum, 2006] Tanenbaum, A. S. (2006). *Structured Computer Organization, Fifth Edition*. Pearson Prentice Hall, Upper Saddle River, NJ. Accessed 24 July 2014 from:
http://e-book.az/download?id=821.

[Tedre and Sutinen, 2008] Tedre, M. and Sutinen, E. (2008). Three traditions of computing: What educators should know. *Computer Science Education*, 18(3):153–170.

[Tenenbaum and Augenstein, 1981] Tenenbaum, A. M. and Augenstein, M. J. (1981). *Data Structures using Pascal*. Prentice-Hall, Englewood Cliffs, NJ.

[Teuscher and Sipper, 2002] Teuscher, C. and Sipper, M. (2002). Hypercomputation: Hype or computation? *Communications of the ACM*, 45(8):23–30. Accessed 28 June 2014 from:
http://www.cs.bgu.ac.il/∼sipper/papabs/hypercomp.pdf.

[Thagard, 1978] Thagard, P. (1978). Why astrology is a pseudoscience. *PSA: Proceedings of the Biennial Meeting of the Philosophy of Science Association*, 1:223–234.

[Thagard, 1984] Thagard, P. (1984). Computer programs as psychological theories. In Neumaier, O., editor, *Mind, Language and Society*, pages 77–84. Conceptus-Studien, Vienna.

[Thagard, 2006] Thagard, P. (2006). *Hot Thought: Mechanisms and Applications of Emotional Cognition*. MIT Press, Cambridge, MA.

[Thagard, 2012] Thagard, P. (4 December 2012). Eleven dogmas of analytic philosophy. Psychology Today. http://www.psychologytoday.com/blog/hot-thought/201212/eleven-dogmas-analytic-philosophy (accessed 13 February 2013).

[*The Economist*, 2013] *The Economist* (2013). Unreliable research: Trouble at the lab. *The Economist*. online at: http://www.economist.com/news/briefing/21588057-scientists-think-science-self-correcting-alarming-degree-it-not-trouble   (accessed 4 November 2013).

[Thomason, 2003] Thomason, R. H. (2003). Dynamic contextual intensional logic: Logical foundations and an application. In Blackburn, P., editor, *CONTEXT 2003: Lecture Notes in Artificial Intelligence 2680*, pages 328–341. Springer-Verlag, Berlin. http://link.springer.com/chapter/10.1007/3-540-44958-2_26#page-1 (accessed 3 December 2013).

[Tingley, 2013] Tingley, K. (2013). The body electric. *The New Yorker*, pages 78–80, 82, 86–86.

[Toussaint, 1993] Toussaint, G. (1993). A new look at euclid's second proposition. *The Mathematical Intelligencer*, 15(3):12–23. http://cgm.cs.mcgill.ca/∼godfried/publications/euclid.pdf (accessed 6 December 2013).

[Tucker et al., 2003] Tucker et al., A. (2003). A model curriculum for k–12 computer science: Final report of the acm k–12 task force curriculum committee, second edition. Computer Science Teachers Association and Association for Computing Machinery. online at:
http://www.acm.org/education/education/curric_vols/k12final1022.pdf (accessed 1 July 2013).

[Tukey, 1958] Tukey, J. W. (1958). The teaching of concrete mathematics. *American Mathematical Monthly*, 65(1):1–9.

[Turing, 1936] 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, 1937] Turing, A. M. (1937). Computability and λ-definability. *Journal of Symbolic Logic*, 2(4):153–163.

[Turing, 1938] Turing, A. M. (1938). On computable numbers, with an application to the entscheidungsproblem: A correction. *Proceedings of the London Mathematical Society, Ser. 2*, 43(1):544–546. http://cs.stanford.edu/people/trevisan/cs172/turing36.pdf (accessed 21 January 2014).

[Turing, 1948] Turing, A. M. (1948). Intelligent machinery. In Copeland, B., editor, *The Essential Turing*, pages 410–432. Oxford University Press, Oxford. Editorial commentary on pp. 395–409. Typescript accessed 26 February 2014 from: http://www.alanturing.net/intelligent_machinery/.

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

[Turing, 1951] Turing, A. M. (1951). Intelligent machinery, a heretical theory. *Philosophia Mathematica*, 4(3):256–260. Alternate version accessed 21 May 2014 from:
http://commonsenseatheism.com/wp-content/uploads/2011/02/Turing-Intelligent-Machinery-a-heretical-theory.pdf; typescripts and further bibliographic information accessed 21 May 2014 from:
http://www.turing.org.uk/sources/biblio1.html.

[Tversky and Kahneman, 1974] Tversky, A. and Kahneman, D. (1974). Judgment under uncertainty: Heuristics and biases. *Science*, 185(4157):1124–1131. http://psiexp.ss.uci.edu/research/teaching/Tversky_Kahneman_1974.pdf (accessed 20 April 2015).

[Uebel, 2012] Uebel, T. (2012). Vienna circle. In Zalta, E. N., editor, *Stanford Encyclopedia of Philosophy (Summer 2012 Edition)*. online. http://plato.stanford.edu/archives/sum2012/entries/vienna-circle/.

[Uglow, 2010] Uglow, J. (2010). The other side of science. *New York Review of Books*, pages 30–31, 34.

[Unger, 1979a] Unger, P. (1979a). I do not exist. In Macdonald, G. F., editor, *Perception and Identity*. Cornell University Press, Ithaca, NY.

[Unger, 1979b] Unger, P. (1979b). Why there are no people. In ED, editor, *Studies in Metaphysics (Midwest Studies in Philosophy, Vol. 4)*, pages 177–222. University of Minnesota Press, Minneapolis. (online at: http://www.thatmarcusfamily.org/philosophy/Course_Websites/Readings/Unger%20-%20No%20People.pdf (accessed 6 February 2013)).

[Vahid, 2003] Vahid, F. (2003). The softening of hardware. *[IEEE] Computer*, 36(4):27–34. Accessed 29 July 2014 from: http://www.cs.ucr.edu/~vahid/pubs/comp03_softerhw.pdf.

[van Leeuwen and Wiedermann, 2013] van Leeuwen, J. and Wiedermann, J. (2013). The computational power of turing's non-terminating circular *a*-machines. In Cooper, S. B. and van Leeuwen, J., editors, *Alan Turing: His Work and Impact*, pages 80–85. Elsevier, Amsterdam.

[Vardi, 2011] Vardi, M. Y. (2011). Computing for humans. *Communications of the ACM*, 54(12):5.

[Vardi, 2012] Vardi, M. Y. (2012). What is an algorithm? *Communications of the ACM*, 55(3):5. Accessed 26 February 2014 from: http://cacm.acm.org/magazines/2012/3/146261-what-is-an-algorithm/fulltext.

[Veblen, 1908] Veblen, T. (1908). The evolution of the scientific point of view. *The University of California Chronicle: An Official Record*, 10(4):395–416. (online at: http://archive.org/details/universitycalif08goog (accessed 6 February 2013).

[von Neumann, 1945] von Neumann, J. (1945). First draft report on the edvac. *IEEE Annals of the History of Computing*, 15(4):27–75. edited by Michael D. Godfrey; accessed 30 October 2013 from http://virtualtravelog.net.s115267.gridserver.com/wp/wp-content/media/2003-08-Th eFirstDraft.pdf; page references to online version.

[von Neumann, 1966] von Neumann, J. (1966). *Theory of Self-Reproducing Automata*. University of Illinois Press, Urbana, IL. Arthur W. Burks (ed.).

[Wadler, 1997] Wadler, P. (1997). How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263. Preprint accessed 26 May 2014 from: https://wiki.ittc.ku.edu/lambda/images/3/3b/Wadler_-_How_to_Declare_an_Imperative.pdf.

[Wainer, 2007] Wainer, H. (2007). The most dangerous equation. *American Scientist*, 95(3):249ff. http://www.americanscientist.org/issues/feature/2007/3/the-most-dangerous-equation/ (accessed 20 April 2015).

[Wainer, 2012] Wainer, H. (2012). The survival of the fittists. *American Scientist*, 100:358–361. online at: http://www.americanscientist.org/issues/pub/the-survival-of-the-fittists/1 (accessed 23 July 2013).

[Wallich, 1997] Wallich, P. (1997). Cracking the u.s. code. *Scientific American*, page 42.

[Walsh, 2014] Walsh, T. (2014). Candy crush's puzzling mathematics. *American Scientist*, 102(6):430–433. Accessed 17 October 2014 from: http://www.americanscientist.org/issues/feature/2014/6/candy-crushs-puzzling-mathematics/.

[Wang et al., 2013] Wang, Z., Busemeyer, J. R., Atmanspacher, H., and Pothos, E. M., editors (2013). *Topics in Cognitive Science 5(4) (October): The Potential of Using Quantum Theory to Build Models of Cognition*. Cognitive Science Society, CITY.

[Wangsness and Franklin, 1966] Wangsness, T. and Franklin, J. (1966). "algorithm" and "formula". *Communications of the ACM*, 9(4):243.

[Wartofsky, 1966] Wartofsky, M. W. (1966). The model muddle: Proposals for an immodest realism. In Wartofsky, M. W., editor, *Models: Representation and the Scientific Theory of Understanding*, pages 1–11. D. Reidel, 1979, Dordrecht, The Netherlands.

[Wartofsky, 1979] Wartofsky, M. W. (1979). Introduction. In Wartofsky, M. W., editor, *Models: Representation and the Scientific Theory of Understanding*, pages xiii–xxvi. D. Reidel, Dordrecht, The Netherlands.

[Weatherson, 2012] Weatherson, B. (18 July 2012). What could leave philosohy? http://tar.weatherson.org/2012/07/18/what-could-leave-philosophy/ (accessed 12 February 2013).

[Wegner, 1976] Wegner, P. (1976). Research paradigms in computer science. In *ICSE '76 Proceedings of the 2nd International Conference on Software Engineering*, pages 322–330. IEEE Computer Society Press, Los Alamitos, CA.

[Wegner, 1995] Wegner, P. (1995). Interaction as a basis for empirical computer science. *ACM Computing Surveys*, 27(1):45–48.

[Wegner, 1997] Wegner, P. (1997). Why interaction is more powerful than algorithms. *Communications of the ACM*, 40(5):80–91.

[Weinberg, 2002] Weinberg, S. (2002). Is the universe a computer? *New York Review of Books*, 49(16).

[Weizenbaum, 1976] Weizenbaum, J. (1976). *Computer Power and Human Reason*. W.H. Freeman, New York.

[Wheeler, 2013] Wheeler, G. (2013). Models, models, and models. *Metaphilosophy*, 44(3):293–300. Preprint accessed 6 May 2014 from: http://philsci-archive.pitt.edu/9500/1/Models.pdf.

[Wiebe and Rapaport, 1986] Wiebe, J. M. and Rapaport, W. J. (1986). Representing *de re* and *de dicto* belief reports in discourse and narrative. *Proceedings of the IEEE*, 74:1405–1413.

[Wiener, 1964] Wiener, N. (1964). *God and Golem, Inc.: A Comment on Certain Points Where Cybernetics Impinges on Religion*. MIT Press, Cambridge, MA. http://www.scribd.com/doc/2962205/God-and-Golem-Inc-Wiener or http://simson.net/ref/1963/God_And_Golem_Inc.pdf.

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

[Wigner, 1960] Wigner, E. (1960). The unreasonable effectiveness of mathematics in the natural sciences. *Communications in Pure and Applied Mathematics*, 13(1). https://www.dartmouth.edu/~matc/MathDrama/reading/Wigner.html, http://www.maths.ed.ac.uk/~aar/papers/wigner.pdf (both accessed 7 April 2015).

[Wilford, 2006] Wilford, J. N. (2006). Early astronomical 'computer' found to be technically complex. *New York Times*.

[Wilford, 2008] Wilford, J. N. (2008). A device that was high-tech in 100 b.c. (discovering how greeks computed in 100 b.c.). *New York Times*, page A12.

[Williams, 1998] Williams, B. (1998). The end of explanation? *The New York Review of Books*, 45(18):40–44.

[Williamson, 2011] Williamson, T. (2011). What is naturalism? *New York Times Opinionator: The Stone*. online at: http://opinionator.blogs.nytimes.com/2011/09/04/what-is-naturalism/ (accessed 22 July 2013).

[Wilson and Papadopoulos, 2004] Wilson, D. G. and Papadopoulos, J. (2004). *Bicycling Science, Third Edition*. MIT Press, Cambridge, MA.

[Wilson and Frenkel, 2013] Wilson, E. and Frenkel, E. (2013). Two views: How much math do scientists need? *Notices of the AMS*, 60(7):837–838. http://dx.doi.org/10.1090/noti1032 (accessed 21 January 2014).

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

[Winsberg, 2001] Winsberg, E. (2001). Simulations, models, and theories: Complex physical systems and their representations. *Philosophy of Science*, 68(3):S442–S454. Supplement: Proceedings of the 2000 Biennial Meeting of the Philosophy of Science Association; Part I: Contributed Papers.

[Wirth, 1971] Wirth, N. (1971). Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227. Reprints online at: http://oberoncore.ru/_media/library/wirth_program_development_by_stepwise_refinement2.pdf, http://sunnyday.mit.edu/16.355/wirth-refinement.html, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.115.9376&rep=rep1&type=pdf (all accessed 5 March 2015).

[Wittgenstein, 1921] Wittgenstein, L. (1921). *Tractatus Logico-Philosophicus*. Humanities Press, New York. Second Edition, with corrections, 1972.

[Wittgenstein, 1958] Wittgenstein, L. (1958). *Philosophical Investigations, Third Edition*. Macmillan, New York.

[Wittgenstein, 1980] Wittgenstein, L. (1980). *Remarks on the Philosophy of Psychology, Vol. I*. University of Chicago Press, Chicago.

[Wolfram, 2002] Wolfram, S. (2002). *A New Kind of Science*. Wolfram Media.

[Worrall, 1989] Worrall, J. (1989). Structural realism: The best of both worlds? *Dialectica*, 43(1–2):99–124.

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

[Young, 2008] Young, J. O. (2008). The coherence theory of truth. In Zalta, E. N., editor, *Stanford Encyclopedia of Philosophy (Fall 2008 Edition)*. http://plato.stanford.edu/archives/fall2008/entries/truth-coherence/.

[Zeigler, 1976] Zeigler, B. P. (1976). *Theory of Modeling and Simulation*. Wiley Interscience, New York.

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

[Zhai and Kristensson, 2012] Zhai, S. and Kristensson, P. O. (2012). The word-gesture keyboard: Reimagining keyboard interaction. *Communications of the ACM*, 55(9):91–101.

[Zobrist, 2000] Zobrist, A. L. (2000). Computer games: Traditional. In Ralston, A., Reilly, E. D., and Hemmendinger, D., editors, *Encyclopedia of Computer Science, 4th edition*, pages 364–368. Grove's Dictionaries, New York.

[Zupko, 2011] Zupko, J. (2011). John Buridan, §7. In Zalta, E. N., editor, *Stanford Encyclopedia of Philosophy (Fall 2011 Edition)*. online. http://plato.stanford.edu/archives/fall2011/entries/buridan/.