On the Relation of Computing to the World

William J. Rapaport

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

`rapaport@buffalo.edu`
`http://www.cse.buffalo.edu/∼rapaport/`

April 21, 2015

Abstract

I survey a common theme that pervades the philosophy of computer science (and philosophy more generally): the relation of computing to the world. Are algorithms merely certain procedures entirely characterizable in an "indigenous", "internal', "intrinsic", "local", "narrow", "syntactic" (more generally: "intra-system") purely Turing-machine language? Or must they interact with the real world, with a purpose that is expressible only in a language with an "external", "extrinsic", "global", "wide", "inherited" (more generally: "extra-" or "inter-"sytem) semantics?

1 Preface

If you begin with Computer Science, you will end with Philosophy.[1]

I was simultaneously surprised and deeply honored to receive the 2015 Covey Award from the International Association for Computing and Philosophy.[2] The honor is due in part to linking me to the illustrious predecessors who have received this award, but also to its having been named for Preston Covey,[3] whom I knew and who inspired me as I began my twin journeys in philosophy and computing.

1.1 From Philosophy to Computer Science, and Back Again

Contrary to the motto above, I began with philosophy, found my way to computer science, and have returned to a mixture of the two. Inspired by Douglas Hofstadter's review [Hofstadter, 1980] of Aaron Sloman's *The Computer Revolution in Philosophy* [Sloman, 1978], which quoted Sloman to the effect that a philosopher of mind who knew no AI was like a philosopher of physics who knew no quantum mechanics,[4] my philosophical interests in philosophy of mind led me to study AI at SUNY Buffalo with Stuart C. Shapiro.[5] This eventually led to a faculty appointment in computer science at Buffalo. (Along the way, my philosophy colleagues and I at SUNY Fredonia published one of the first introductory logic textbooks to use a computational approach [Schagrin et al., 1985].)

At Buffalo, I was amazed to discover that my relatively arcane philosophy dissertation on Alexius Meinong was directly relevant to Shapiro's work in AI, providing an intensional semantics for his SNePS semantic-network processing system (see, e.g., [Shapiro and Rapaport, 1987], [Shapiro and Rapaport, 1991]).[6] And then I realized that the discovery of quasi-indexicals ('he himself', 'she herself', etc.; [Castañeda, 1966]) by my dissertation advisor, Hector-Neri Castañeda

[1] "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" (http://en.wikipedia.org/wiki/Wikipedia:Getting_to_Philosophy). If you begin with "Computer Science", you will end with "Philosophy" (in 12 links).

[2] http://www.iacap.org/awards/

[3] http://en.wikipedia.org/wiki/Covey_Award

[4] "I am prepared to go so far as to say that within a few years, if there remain any philosophers who are not familiar with some of the main developments in artificial intelligence, it will be fair to accuse them of professional incompetence, and that to teach courses in philosophy of mind, epistemology, aesthetics, philosophy of science, philosophy of language, ethics, metaphysics, and other main areas of philosophy, without discussing the relevant aspects of artificial intelligence will be as irresponsible as giving a degree course in physics which includes no quantum theory" [Sloman, 1978, p. 5].

[5] http://www.cse.buffalo.edu/∼shapiro/

[6] I presented some of this work at CAP 1987.

could repair a "bug" in a knowledge-representation theory that Shapiro had developed with another convert to computer science (from psychology), Anthony S. Maida [Maida and Shapiro, 1982]. (This work was itself debugged with the help of yet another convert (from English), my doctoral student Janyce M. Wiebe [Rapaport et al., 1997].)

My work with Shapiro and our SNePS Research Group at Buffalo enabled me to rebut my Covey Award predecessor John R. Searle's Chinese-Room Argument [Searle, 1980] using a philosophical theory of what I call "syntactic semantics" [Rapaport, 1986b], [Rapaport, 1988], [Rapaport, 1995], [Rapaport, 2012].[7] And both of these projects, as well as one of my early Meinong papers [Rapaport, 1981], led me, together with another doctoral student (Karen Ehrlich) and (later) a colleague from Buffalo's Department of Learning and Instruction (Michael W. Kibby) to develop a computational and pedagogical theory of vocabulary acquisition from context [Rapaport and Kibby, 2007], [Rapaport and Kibby, 2014].[8]

1.2 The Philosophy of Computer Science

All of this inspired me to create and teach a course on the philosophy of computer science [Rapaport, 2005b][9] and, now in retirement, to write up my lecture notes as a textbook [Rapaport, 2015].

The course and the text begin with a single question: What is computer science?[10] To answer this, we need to consider a series of questions, each of which leads to another:

- Is computer science a science? (And what is science?) Or is it a branch of engineering? (What is engineering?) (Or is it a combination of these? Or perhaps something completely different, new, *sui generis*?)

- If it is a science, what is it a science of? Of computers? (In that case, what is a computer?) Or of computation? That question leads to many others:

- What is computation? What is an algorithm? Do algorithms differ from procedures? From recipes? (And what are they?) What is the (Church-Turing) Computability Thesis?[11] What is hypercomputation? What is a computer

[7] I presented some of this work at IACAP 2009 and NACAP 2010.

[8] I presented some of this work at NACAP 2006.

[9] Presented at NACAP 2006 in my Herbert A. Simon Keynote Address, http://www.hass.rpi.edu/streaming/conferences/cap2006/nacp_8_11_2006_9_1010.asx

[10] This is the first question, but, because there are two intended audiences—philosophy students and computer-science students—I actually begin with a zeroth question: What is philosophy?

[11] I prefer this name over "Church's Thesis", "Turing's Thesis", or the "Church-Turing Thesis" for reasons given in [Soare, 2009, §§3.5, 12].

program? This question also gives rise to a host of others:

- What is an implementation? What is the relation of a program to that which it models or simulates? (For that matter, what is simulation? And can programs be considered to be (scientific) theories?) What is software, and how does it relate to hardware? (And can, or should, one or both of those be copyrighted or patented?) Can computer programs be (logically) verified?

- There are, of course, issues in the philosophy of AI (What is AI? What is the relation of computation to cognition? Can computers think? What are the Turing Test and the Chinese-Room Argument?).

- Finally, there are issues in computer ethics, but I only touch on two that I think are not widely dealt with in the already voluminous computer-ethics literature: Should we trust decisions made by computers? Should we build "intelligent" computers?

There are many issues that I don't deal with: The nature of information, the role of the Internet in society, the role of computing in education, and so on. But I have to stop somewhere!

My goal in the book is not to be comprehensive, but to provide background on some of the major issues and a guide to some of the major papers, and to raise questions for readers to think about (together with a guide to how to think about them—the text contains a brief introduction to critical thinking and logical evaluation of arguments). For a philosophy textbook, raising questions is more important than answering them. My goal is to give readers the opportunity and the means to join a long-standing conversation and to devise their own answers to some of these questions.

2 A Common Thread: Computing and the World

In the course of writing the book, I have noticed a theme that pervades its topics. In line with my goals for the book, I have not yet committed myself to a position; I am still asking questions and exploring. In this essay, I want to share those questions and explorations with you.

The common thread that runs through most, if not all, of these topics: the relation of computing to the world:

> Is computing about the *world*? Is it "external", "global", "wide", or "semantic"?

> Or is it about *descriptions* of the world? Is it, instead, "internal", "local", "narrow", or "syntactic"?

And I will quickly agree that it might be both! In that case, the question is how the answers to these questions are related.

This theme should be familiar to most of you; I am not announcing some newly discovered philosophical puzzle. But it isn't necessarily familiar or obvious to my book's intended audience, and I do want to recommend it as a topic worth pondering and worth discussing with students.

In this section, we'll survey these issues as they appear in some of the philosophy of computer science questions of §1.2. In subsequent sections, we'll go into more detail. But I only promise to raise questions, not to answer them (in the course and the text: to challenge my students' thinking, not to tell them what to think). Leslie Lamport recently said (quoting cartoonist Richard Guindon), " 'Writing is nature's way of letting you know how sloppy your thinking is.' We think in order to understand what we are doing. If we understand something, we can explain it clearly in writing. If we have not explained it in writing, then we do not know if we really understand it" [Lamport, 2015, 38].[12]

2.1 Some Thought Experiments

Castañeda used to say that philosophizing must begin with data. So let's begin with some data in the form of real and imagined computer programs.

2.1.1 Rey's and Fodor's Chess and War Programs

Jerry Fodor, taking up a suggestion by Georges Rey, asks us to consider a computer that simulates the Six Day War and a computer that simulates (or actually plays?) a game of chess, but which are (are?) such that "the internal career of a machine running one program would be identical, step by step, to that of a machine running the other" [Fodor, 1978, p. 232].

A real example of the same kind is "a method for analyzing x-ray diffraction data that, with a few modifications, also solves Sudoku puzzles".[13] Or consider a computer version of the murder-mystery game Clue that exclusively uses the Resolution rule of inference, and so could be a general-purpose propositional theorem prover instead.[14]

[12] As E.M. Forster is alleged to have said, "How can I know what I think till I see what I say?" https://rjheeks.wordpress.com/2011/04/13/discovery-writing-and-the-so-called-forster-quote/

[13] Cited in the biographical sketch for a book review by Veit Elser [Elser, 2012].

[14] Thanks to Robin Hill, personal communication, for this example.

Similar examples abound, notably in applications of mathematics to science, and these can be suitably "computationalized" by imagining computer programs for each. For example,

> 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].

Other examples, concerning, e.g., the applicability of group theory to physics are discussed in [Frenkel, 2013]. And [Halmos, 1973, p. 384] points out that once aspects of Hilbert spaces are seen to be structurally identical to aspects of quantum-mechanical systems, "the difference between a quantum physicist and a mathematical operator-theorist becomes one of language and emphasis only". Clearly, Eugene Wigner's classic paper [Wigner, 1960] is relevant here. The issues here are also akin to, if not the same as, those that motivate structuralism in the philosophy of mathematics (we'll return briefly to this in §5.2).

In these examples, do we have one algorithm, or two?[15]

2.1.2 Cleland's Recipe for Hollandaise Sauce

Carol Cleland offers an example of a recipe for hollandaise sauce [Cleland, 1993], [Cleland, 2002]. Let's suppose that we have an algorithm (a recipe) that tells us to mix eggs and oil, and that outputs hollandaise sauce.[16] Suppose that, on Earth, the result of mixing the egg and oil is an emulsion that is, in fact, hollandaise sauce. And let us suppose that, on the Moon, 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).

Can a Turing machine make hollandaise sauce? Is making hollandaise sauce computable?

[15]Compare this remark: "Recovering motives and intentions is a principal job of the historian. For without some attrribution of mental attitudes, actions cannot be characterized and decisions assessed. *The same overt behavior, after all, might be described as 'mailing a letter' or 'fomenting a revolution.'* " [Richards, 2009, 415].

[16]Calling a recipe an "algorithm", despite its ubiquity in introductory presentations of computer science, is controversial; see [Preston, 2013, Ch. 1] for some useful discussion of the *non*-algorithmic, improvisational nature of recipes.

2.1.3 A Blocks-World Robot

Consider a blocks-world computer program that instructs a robot how to pick up blocks and move them onto or off of other blocks [Winston, 1977]. 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. [Rapaport, 1995, §2.5.1].

Did the program behave as intended?

2.1.4 A GCD Program

Michael Rescorla [Rescorla, 2013, §4] offers an example that is reminiscent of Cleland's, but less "physical". Here is a Scheme program for computing the greatest common divisor (GCD) of two numbers:

```
(define (gcd a b)
  (if (= b 0)
    a
    (gcd b (remainder a b))))
```

Implement this program on two computers, one (M_{10}) using base-10 notation and one (M_{13}) using base-13 notation. Rescorla argues that only M_{10} executes the Scheme program *for computing GCDs*, even though, in a "narrow" sense, both computers are executing the "same" program. When the *numerals* '115' and '20' are input to M_{10}, it outputs the numeral '5'; "it thereby calculates the greatest common divisor of the corresponding *numbers*" [Rescorla, 2013, p. 688]. But the *numbers* expressed in base-13 by '115' and '20' are 187_{10} and 26_{10}, respectively, and their GCD is 1_{10}, not 5_{10}. So, in a "wide" sense, the two machines are doing "different things".

Are these machines doing different things?

2.1.5 A Spreadsheet

I vividly remember the first semester that I taught a "Great Ideas in Computer Science" course aimed at computer-phobic students. We were going to teach the students how to use a spreadsheet program, something that I had never used! So, with respect to this, I was as naive as any of my students. My TA, who had used spreadsheets before, gave me something like the following instructions:

enter a number in cell_1;
enter a number in cell_2;
enter '=⟨click on cell_1⟩⟨click on cell_2⟩' in cell_3

I had no idea what I was doing. I was blindly following her instructions *and had no idea that I was adding two integers*. Once she told me that that was what I was doing, my initial reaction was "Why didn't you tell me that before we began?".

When I entered those data into the spreadsheet, was I adding two numbers?

2.2 What Is Computer Science?

Science, no matter how conceived, is generally agreed to be a way of *understanding* the world. So, if computer science is a science—and, of course, just because it is called 'computer *science*' does not imply that it *is* a science!—then it should be a way of understanding the world *computationally*.

Engineering, no matter how conceived, is generally agreed to be a way of *changing* the world (preferably by improving it).[17] So, if computer science is an engineering discipline, then it should be a way of changing the world (for the better?)—by implementing algorithms in computer programs that can have physical effects.

Computer science tries to do both: to understand the world computationally, and to change the world by building computational artifacts. In introductory classes, I offer the following definition:

> Computer science is the scientific study[18] of:
>
> - *what* can be computed (narrowly, which mathematical functions are computable; widely, which real-world tasks are automatable [Forsythe, 1968]),
> - *how* to compute such functions or tasks (how to represent the necessary information and how to construct appropriate algorithms),
> - how to compute them *efficiently*,

[17]"[S]science tries to understand the world, whereas engineering tries to change it" [Staples, 2015, §1], paraphrasing Marx: "The philosophers have only interpreted the world, in various ways; the point is to change it" [Marx, 1845, Thesis 11]. Was Marx proposing a discipline of "philosophical engineering"?

[18]Using the adjective 'scientific' instead of the noun 'science' neatly puts the science-vs.-engineering dispute in the background. After all, surely engineering is "scientific", even if it isn't a "science". And, whether or not computer science is a "science", it is surely a systematic, *scientific* field of study.

- and how to *engineer* computers that can implement those computations in the real world.

And here we glimpse the first strand of our thread: Is computation concerned with (a) the internal workings of a computer (both abstractly in terms of the theory of computation—e.g., the way in which a Turing machine works—as well as more concretely in terms of the internal physical workings of a physical computer)? Or (but not necessarily exclusively "or"!) with (b) how those internal workings can reach out to the world in which they are embedded?

And, here, it is also important to point out similarities with a classic issue in the philosophy of mind: How does the mind (or, more materialistically, the brain) reach out to "harpoon" the world?[19]

2.3 What Is a Computer?

Computer science might be the study of comput*ers* or of computa*tion*. If it is the science or even the engineering of comput*ers*, then we need to ask what a computer is. Is a computer an abstract, mathematical entity, in particular, a Turing machine (or a universal Turing machine)? (Or, for that matter, anything logically equivalent to a Turing machine, such as a λ-"calculator" or a recursive-function "machine"?) Or is it any physical, real-world object that implements a (universal) Turing machine? (Or both, of course.) If it is physical, which things in the world are computers (besides the obvious suspects, such as Macs, PCs, iPhones, etc.)? Notably, is the brain a computer? Is the solar system a computer (computing Kepler's laws)? Is the universe a computer?[20]

Here, we see another strand of our thread: Where should we look for an answer to what a computer is? Should we look narrowly to mathematics, or more widely to the real world? (And I won't even mention the related issue of mathematical Platonism, which might respond that looking "narrowly" to math is, indeed, looking even more widely than the merely real world.)

2.4 What Is Computation?

When we ask what computation is, things get more complex. Is computation "narrow", focusing only on the operations of a Turing machine (print, move) or on basic recursive functions (successor, predecessor, projection)? Or "wide", involving, say,

[19] See, e.g., [Castañeda, 1989, p. 114, "The Harpooning Model"].

[20] On the brain, see, e.g., [Searle, 1990]; on the solar system, see, e.g., [Copeland, 1996, §2], [Perruchet and Vinter, 2002, §1.3.4], [Shagrir, 2006, p. 394]; on the universe, see, e.g., [Weinberg, 2002], [Wolfram, 2002], [Lloyd and Ng, 2004].

chess pieces and a chess board (for a chess program) or soldiers and a battlefield (for a wargame simulator)? Is it independent of the world, or is it world-involving?

Are algorithms purely logical? Or are they "intentional" [Hill, 2015] and "teleological" [Anderson, 2015]? Which of these two forms do they take?:

> Do *P*
> (where '*P*' is either a primitive computation, or a set of computations recursively structured by sequence, selection, and repetition, i.e., a "procedure").

Or

> In order to accomplish goal *G*, do *P*

If the former, then computation is "narrow"; if the latter, then "wide". (We'll return to this in §4.)

What *is* a procedure? Are recipes procedures? Is "Make hollandaise sauce" a high-level procedure call? If so, then computation is wide. (We'll return to this in §4.4.)

Is making hollandaise sauce Turing-machine computable? Or is it a task that goes beyond Turing machines? How does interactive (or oracle) computation, relate to Turing-machine computation? Turing-machine computation seems to be "narrow"; the others, "wide". (We'll return to this in §6.)

2.5 What Is a Computer Program?

Is an algorithm an implementation of a function, a computer program an implementation of an algorithm, and a process (i.e., a program being executed on a computer) an implementation of a program? Implementation is a relation between something more or less *abstract* and something else that is more or less *concrete* (at least, that is less abstract). It is the central relation between abstraction and reality (as well as where science meets engineering).

Elsewhere, I have argued that implementation is most usefully understood as (external) semantic interpretation. More precisely, *I* is an implementation, in medium *M*, of "Abstraction" *A* iff *I* is a semantic interpretation or model of *A*, where *A* is some syntactic domain and *M* is the semantic domain [Rapaport, 1999, 128], [Rapaport, 2005a]. Typically (but not necessarily), *I* is a real-world system in some physical medium *M*, and *A* is an abstract or formal system (but both *I* and *A* could be abstract. The theme of the relation of computing to the real world is obviously related to this issue.

It has been claimed that (at least some) computer programs are theories [Simon and Newell, 1962, p. 97], [Johnson-Laird, 1981, pp. 185–186]

[Pylyshyn, 1984, p. 76]. (For the contrasting view, see [Moor, 1978, §4], [Thagard, 1984].) How do theories relate to the world? Do computer programs simulate the world? (We'll return to this a little bit in §7.)

Are computer programs software or hardware? Here we have a computational version of the mind-body problem. And it has legal ramifications in terms of what can be copyrighted and what can be patented.

Can computer programs be verified? This question concerns, at least in part, the correlation of a syntactic description (of the world) with a domain of semantic interpretation (i.e., the world being described). (We'll return to this in §7.)

It is time to delve into some of these.

3 Inputs, Turing Machines, and Outputs

> Any machine is a prisoner of its input and output domains.
> [Newell, 1980, 148]

Let's begin at the beginning, with Turing machines. The tape of a Turing machine records symbols in its "cells", usually '0' or '1'. Is the tape the input-output device of the Turing machine? Or is it the machine's internal memory device?

Given a Turing machine for computing a certain mathematical function, it is certainly true that the function's inputs will be inscribed on the tape at the beginning of the computation, and the outputs will be printed onto the tape by the time that the computation halts. Moreover, the inscriptions on the tape will be used and modified by the machine during the computation, in the same way that a physical computer uses its internal memory for storing intermediate results of a computation. So it certainly looks like the answer to our questions is: both.

But, although Turing's *a*-machines were designed to simulate human computers,[21] Turing doesn't talk about the humans who would *use* them. A Turing machine *doesn't* accept user-supplied input from the external world! It begins with all data pre-stored on its tape and then simply does its own thing, computing the output of a function and leaving the result on the tape. Turing machines don't "tell" anyone in the external world what the answers are, though the answers are there for anyone to read because the "internal memory" of the machine is visible to the external world. Of course, a user would have to be able to *interpret* the symbols on the tape, and thereon hangs a tale.

[21] That is, humans who compute. In some recent writings (see §4.2), they are called 'comput*o*rs' [Sieg, 1994], so as to be able to use a word that sounds like 'computer' without the 21st-century implication that it is something like a Mac or a PC. I prefer to call them 'clerks'.

The question I want to look at here is whether the symbols on the tape are really inputs and outputs in the sense of coming from, and being reported to, the external world.

Are inputs and outputs an essential part of an algorithm? After all, the input-output interface "merely" connects the algorithm with the world. It may seem outrageous to deny that they are essential, but it's been done!

3.1 Are Inputs Needed?

It's outrageous, of course, because algorithms are supposed to be ways of computing mathematical functions, and mathematical functions, by definition, have inputs and outputs. They are, after all, certain sets of ordered pairs of inputs and outputs, and you can't very well have an ordered *pair* that is missing one or both of those. A.A. Markov's informal characterization of algorithm has an "applicability" condition stating that algorithms must have "The possibility of starting from original given objects which can vary within known limits" [Markov, 1954, p. 1]. Those "original given objects" are, presumably, the input.

But Donald Knuth's informal characterization of the notion of algorithm has an "input" condition stating that "An algorithm has *zero or more* inputs" [Knuth, 1973, p. 5; my italics]! He not only doesn't explain this, but he goes on to characterize outputs as "quantities which have a specified relation to the inputs" [Knuth, 1973, p. 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?[22] Knuth is not alone in this: Juris Hartmanis and R.E. Stearns's classic paper on 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 [Hartmanis and Stearns, 1965, p. 288].

One way to understand this is that some programs merely output information, such as prime-number generators. In cases such as this, although there may not be any *explicit* input, there is an *implicit* input (roughly, ordinals: the algorithm outputs the nth prime, without explicitly requesting an n to be input). Another kind of function that might seem not to have any (explicit) inputs is a constant function, but, again, its implicit input could be anything (or anything of a certain type, "varying within known limits", as Markov might have said).

So, what constitutes input? Is it simply the initial data for a computation? Or is it information supplied to the computer from the external world (and interpreted or translated into a representation of that information that the computer can "understand" and manipulate)?

[22] Is this a relation to a non-existent entity in a Meinongian sense? See [Grossmann, 1974, p. 109], [Rapaport, 1986a, §4.5].

3.2 Are Outputs Needed?

What about output? Markov, Knuth, and Hartmanis & Stearns all require at least one output. Markov, for example, has an "effectiveness" condition (using that term slightly differently from others, such as Church or Knuth) stating that an algorithm must "obtain a certain result".

But B. Jack Copeland and Oren Shagrir suggest that a Turing machine's output might be unreadable [Copeland and Shagrir, 2011, pp. 230–231]. Imagine, not a Turing machine with a tape, but a physical computer that literally prints out its results. Suppose that the printer is broken or that it has run out of ink. Or suppose that the programmer failed to include a 'print' command in the program. The computer's program would compute a result but not be able to tell the user what it is. Consider this algorithm from [Chater and Oaksford, 2013, p. 1172, citing an example from [Pearl, 2000]]:

1. input P
2. multiply P by 2; store in Y
3. add 1 to Y; store in Z

This algorithm does not appear to have an output. The computer has computed $2X+1$ and stored it away in Z for safekeeping, but doesn't tell you its answer. There *is* an answer, but it isn't output. ("I know something that you don't!"?)

So, what constitutes "output"? Is it simply the final result of a computation? Or is it some kind of translation or interpretation of the final result that is physically output and implemented in the real world? In the former case, wouldn't both of §2.1.4's base-10 and base-13 GCD computers be doing the same thing? A problem would arise only if they told us what results they got, and we—reading those results—would interpret them, possibly incorrectly.

3.3 When Are Inputs and Outputs Needed?

Here is Gualtiero Piccinini on this topic:

> 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, p. 741, n. 11; my italics]

Or, as Allen Newell put it:

> 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. [Newell, 1980, 148][23]

There you have it: Narrowly conceived, algorithms might not need inputs and outputs. Widely conceived, they do. Any input from the external world would have to be *encoded* by a user into a language "understandable" by the Turing machine (or else the Turing machine would need to be able to *decode* such external-world input). And any output *from* the Turing machine to be reported *to* the external world (e.g., a user) would have to be *encoded* by the Turing machine (or *decoded* by the user). Such codings would, themselves, have to be algorithmic.

In fact, the key to determining which real-world tasks are computable—one of the main questions that computer science is concerned with (§2.2)—is finding coding schemes that allow the sequence of '0's and '1's (i.e., a natural number in binary notation) on a Turing machine's tape to be *interpreted* as a symbol, a pixel, a sound, etc. A mathematical function on the natural numbers is computable iff it is computable by a Turing machine (according to the Church-Turing Computability Thesis); thus, a real-world problem is computable iff it can be encoded as such a computable mathematical function.

But it's that wide conception, requiring algorithmic, semantic interpretations of the inputs and outputs, that leads to various debates.

3.4 Must Inputs and Outputs Be Interpreted Alike?

There is another input-output issue, which has not been discussed much in the literature but which is relevant to our theme. [Rescorla, 2007, p. 254] notes that

> Different textbooks employ different correlations between Turing machine syntax and the natural numbers. The following three correlations are among the most popular:[24]

$$d_1(\underline{n}) = n.$$

$$d_2(\underline{n+1}) = n.$$

$$d_3(\underline{n+1}) = n, \text{as an input.}$$

$$d_3(\underline{n}) = n, \text{as an output.}$$

[23] 'Universal', as Newell uses it here, means being able to "produce an arbitrary input-output function" [Newell, 1980, 147].

[24] [The symbol '$\underline{x}$' represents a sequence of x strokes, where x is a natural number.–WJR]

> 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.

Let's focus on interpretations like d_3 (we'll look at d_1 and d_2 in §5.2). This idea of having different input and output interpretations occurs all the time in the real world. (I don't know how often it is considered in the more rarefied atmosphere of computation theory.) For example, machine-translation systems that use an "interlingua" work this way: Chinese input is encoded into an "interlingual" representation language (often thought of as an internal, "meaning"-representation language that encodes the "proposition" expressed by the Chinese input), and English output is generated from that interlingua (re-expressing in English the proposition that was originally expressed in Chinese).[25] Cognition (assuming that it is computable!) also works this way: Perceptual encodings into the "language" of the biological neural network of our brain surely differ from motor decodings. (Newell's above-quoted examples of hearing and flying are surely different.)

Consider, again, Rescorla's GCD program. And let us consider, not Scheme, but a Common Lisp version. The Common Lisp version will look identical to the Scheme version (the languages share most of their syntax), but the Common Lisp version has two global variables—`*read-base*` and `*print-base*`—that tell the computer how to interpret input and how to display output. By default, `*read-base*` is set to 10. So the Common Lisp read-procedure sees the three-character sequence '115' (for example); decides that it satisfies the syntax of an integer; converts that sequence of characters to an internal representation of type `integer`—*which is represented internally as a binary numeral implemented as bits or switch-settings*—does the same with (say) '20'; and computes their GCD using the algorithm from §2.1.4 on the binary representation. If the physical computer had been an old IBM machine, the computation might have used binary-coded decimal numerals instead, thus computing in base 10. If `*read-base*` had been set to 13, the input characters would have been interpreted as base-13 numerals, and the very same Common Lisp (or Scheme) code would have correctly computed the GCD of 187_{10} and 26_{10}. One could either say that the algorithm computes with *numbers*—not numerals—or with *base-2 numerals as a canonical representation of numbers*, depending on one's view concerning such things as Platonism or nominalism. And similarly for output: The switch-settings containing the GCD of the input are then output as base-10 or base-13 numerals, as pixels on a screen or ink on paper, depending on the value of such things as `*print-base*`. The

[25] My former student Min-Hung Liao used SNePS for this purpose in [Liao, 1998]. For more on the history of interlinguas in computer science, see [Daylight, 2013, §2].

point, once again, with respect to Rescorla's example, is that a single Common Lisp (or Scheme) algorithm is being executed by both M_{10} and M_{13}. Those machines *are* different; they do *not* "have the same local, intrinsic, physical properties" [Rescorla, 2013, 687], because M_{10} has `*read-base*` and `*print-base*` set to 10, whereas M_{13} has `*read-base*` and `*print-base*` set to 13.[26]

The aspect of this situation that I want to remind you of is whether the tape is the *external* input and output device, or is, rather, the machine's *internal* memory.[27] If it is the machine's internal memory, then, in some sense, there is no (visible or user-accessible) input or output (§3). If it is an external input-output device, then the marks on it are for *our* convenience only. In the former case, the only accurate description of the Turing machine's behavior is syntactically in terms of stroke-appending. In the latter case, we can use that syntactic description but we can also embellish it with one in terms of our interpretation of what it is doing. (We'll return to this in §5.2.)

4 Are Algorithms Teleological (Intentional)?

Let's begin untangling our thread with the question of whether the proper way to characterize an algorithm must include an "intentional or teleological preface", so to speak.

4.1 What Is an Algorithm?

The history of computation theory is, in part, an attempt to make mathematically precise the informal notion of an algorithm. Turing more or less "won" the competition. (At least, he tied with Church. Gödel, also in the race, placed his bet on Turing [Soare, 2009]). Many informal characterizations of "algorithm" exist (such as Knuth's and Markov's, mentioned in §3.1); they can be summarized as follows [Rapaport, 2012, Appendix, pp. 69–71]:

> An algorithm (for executor *E*) [to accomplish goal *G*] is:
>
> 1. a procedure *P*, i.e., a finite set (or sequence) of statements (or rules, or instructions), such that each statement *S* is:
>   (a) composed of a finite number of symbols (better: uninterpreted marks) from a finite alphabet
>   (b) and unambiguous (for *E*—i.e.,

[26] I am indebted to Stuart C. Shapiro, personal communication, for the ideas in this paragraph.
[27] [Dresner, 2003] and [Dresner, 2012] discuss this.

i. *E* "knows how" to do *S*,
ii. *E* can do *S*,
iii. *S* can be done in a finite amount of time
iv. and, after doing *S*, *E* "knows" what to do next—),

2. *P* takes a finite amount of time, i.e., halts,
3. [and *P* ends with *G* accomplished].[28]

4.2 Do Algorithms Need a Purpose?

I think that the notion of an algorithm is best understood with respect to an executor. One machine's algorithm might be another's ungrammatical input [Suber, 1988]. We can probably rephrase the above characterization without reference to *E*, albeit more awkwardly, hence the *parentheses* around references to *E*.

But the present issue is whether the *bracketed* comments about the task to be accomplished are essential. My former student Robin K. Hill has recently argued in favor of including *G*, roughly on the grounds that a "prospective user" needs "some understanding of the task in question" over and above the mere instructions ([Hill, 2015, §5]. Algorithms, according to Hill, must be expressed in the form "In order to accomplish *G*, do *P*", not merely "Do *P*".

Stephen C. Kleene's informal characterization of "algorithm" [Kleene, 1995, p. 18] begins as follows:

> [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. *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,…

[28] "[N]ote…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" [Rapaport, 2012, p. 71]. A few other things to note: The characterization of a procedure as a *set* of *statements* (with the parenthetical alternatives of sequence, rules, and instructions) is intended to abstract away from issues about imperative/procedural vs. declarative presentations. Whether the "letters" of the alphabet in which the algorithm is written are considered to be either symbols in the classical sense of mark-plus-semantic-interpretation or else uninterpreted marks (symbols that are not "symbolic of" anyting) is another aspect of our common thread. Talk of "knowing how" does not presume that *E* is a cognitive agent (it might be a CPU), but I do want to distinguish between *E*'s (i) being able *in principle* (i.e., "knowing how") to execute a statement and (ii) being able *in practice* to (i.e., "can") execute it. Similarly, "knowing" what to do next does not presume cognition, but merely having a deteministic way to proceed from one statement to the "next". I am using 'know' and its cognates here in the same (not necessarily cognitive) way that AI researchers use it in the phrase 'knowledge base'.

Note that the procedure has a purpose: "answering any one of a given infinite class of questions". And the procedure depends on that class: Given a class of questions, there is a procedure such that, given "*any* question from the class," the procedure "enable[s] us to recognize that now we have the answer before us and to read it off".

David Marr analyzed information processing into three levels: computational (*what* a system does), algorithmic (*how* it does it), and physical (how it is *implemented*) [Marr, 1982]. I have never liked these terms, preferring 'functional', 'computational', and 'implementational', respectively: Certainly, when one is doing mathematical computation (the kind that Turing was concerned with in [Turing, 1936], one begins with a *mathematical function* (i.e., a certain set of ordered pairs), asks for an algorithm to *compute* it, and then seeks an *implementation* of it, *possibly* in a *physical* system such as a computer or the brain (or perhaps even [Searle, 1982]'s beer cans and levers powered by windmills), but *not necessarily* (e.g., the functionality of an abstract data type such as a stack can be *abstractly* implemented using a list.[29]

In *non*-mathematical fields (e.g., cognition), the set of ordered pairs of input-output *behavior* is expressed in problem-specific language;[30] the algorithmic level will also be expressed in that language; and the implementation level might be the brain or a computer. A recipe for hollandaise sauce developed in this way needs to say more than just something along the lines of "mix these ingredients in this way"; it must take the external environment into account. (We will return to this in §4.4, and we will see how it can take the external world into account in §5.4.)

Marr's "computational" level is rather murky. Frances Egan takes the mathematical functional view just outlined.[31] On that view, Marr's "computational" level is mathematical.

Barton Anderson [Anderson, 2015, §1], on the other hand, says that Marr's "computational" level

> concern[s] the presumed *goal* or *purpose* of a mapping,[32] that is, the specification of the 'task' that a particular computation 'solved.' Algorithmic level questions involve specifying how this mapping was achieved computationally, that is, the formal procedure that transforms an input representation into an output representation.

On this view, Marr's "computational" level is *teleological*. In the formulation "To

[29]This is one reason that I have argued that implementation is semantic interpretation [Rapaport, 1999], [Rapaport, 2005a].

[30]Using an "external" or "inherited" semantics; these terms will be explained below.

[31][Egan, 1991, pp. 196–107], [Egan, 1995, p. 185]; cf. [Shagrir and Bechtel, 2015, §2.2].

[32]Of a mathematical function?

accomplish *G*, do *P*", the "To *G*" preface expresses the teleological aspect of Marr's "computational" level; the "do *P*" seems to express Marr's "algorithm" level.

According to [Bickle, 2015], Marr was trying to counter the then-prevailing methodology of trying to *describe* what neurons were doing (a "narrow", internal, implementation-level description) without having a "wide", external, "computational"-level *purpose* (a "function" in the teleological, not mathematical, sense). Such a teleological description would tell us "why" [Marr, 1982, p. 15, as quoted in [Bickle, 2015]] neurons behave as they do.

Shagrir and William Bechtel suggest that Marr's "computational" level conflates two separate, albeit related, questions: not only "why", but also "what". On this view, Egan is focusing on the "what", whereas Anderson is focusing on the "why" [Shagrir and Bechtel, 2015, §2.2]. We will return to this in a moment.

Certainly, knowing what the goal of an algorithm is makes it easier for a *cognitive agent* executor to follow the algorithm and to have a fuller understanding of what s/he is doing. I didn't understand that I was adding when my TA told me to enter certain data into the cells of the spreadsheet. It was only when she told me that that was how I could add two numbers with a spreadsheet that I understood.

Now, (I like to think that) I am a cognitive agent who can come to understand that entering data into a spreadsheet can be a way of adding. A Turing machine that adds or a Mac running Excel is not such a cognitive agent. It does not understand what addition is or that that is what it is doing. And it does not have to. However, an AI program running on a robot that passes the Turing test would be a very different matter; I have argued elsewhere that such an AI program could, would, and should (come to) understand what it was doing.[33]

The important point is that—despite the fact that understanding what task an algorithm is accomplishing makes it easier to understand the *algorithm* itself—"blind" following of the algorithm is all that is necessary to *accomplish* the task. Understanding the task—the goal of the algorithm—is expressed by the intentional/teleological preface. This is akin to dubbing it with a name that is meaningful to the user, as we will discuss in §5.3.

That computation can be "blind" in this way is what [Fodor, 1980] expressed by his "formality condition" and what Daniel C. Dennett has called

> Turing's. . . strange inversion of reasoning. The Pre-Turing world was one in which computers were people, who had to understand mathematics in order to do their jobs. Turing realised that this was just not necessary: you could take the tasks they performed and squeeze out the last tiny smidgens of understanding, leaving nothing but brute,

[33] [Rapaport, 1988], [Rapaport, 2012]. See my former doctoral student Albert Goldfain's work on how to get AI computer systems to understand mathematics [Goldfain, 2006], [Goldfain, 2008].

> mechanical actions. IN ORDER TO BE A PERFECT AND BEAUTIFUL COMPUTING MACHINE IT IS NOT REQUISITE TO KNOW WHAT ARITHMETIC IS. [Dennett, 2013, p. 570, caps in original][34]

As I read it, the point is that a Turing machine need not "know" that it is adding, but agents who do understand adding can use that machine to add.

Or can they? In order to do so, the machine's inputs and outputs have to be interpreted—understood—by the user as representing the numbers to be added. And that seems to require an appropriate relationship with the external world. It seems to require a "user manual" that tells the user what the algorithm does in the way that Hill prescribes, not in the way that my TA explained what a spreadsheet does. And such a "user manual"—an intention or a purpose for the algorithm—in turn requires an interpretation of the machine's inputs and outputs.

But before pursuing this line of thought, let's take a few more minutes to consider "Turing's inversion", the idea that a Turing machine can be doing something very particular by executing an algorithm without any specification of what that algorithm is "doing" in terms of the external world. Algorithms, on this view, seem not to have to be intentional or teleological, yet they remain algorithms.

Brian Hayes offers two expressions of an algorithm that ants execute [Hayes, 2004]:

teleological version:

To create an ant graveyard, gather all your dead in one place.[35]

non-teleological version:

1. If you see a dead ant,[36] and if you are not carrying one, then pick it up.
2. If you see a dead ant, and if you are carrying one, then put yours down near the other one.

As Hayes notes, the teleological version requires planning and organization skills far beyond those that an ant might have, not to mention conceptual understanding that we might very well be unwilling to ascribe to ants. More to the point, the

[34] See also the more easily accessible [Dennett, 2009, p. 10061].

[35] This is a "fully" teleological version, with a high-level, teleologically formulated execution statement. A "partially" teleological version would simply prefix "To create an ant graveyard" to the following non-teleological version.

[36] Note that testing this condition does not require the ant to have a *de dicto* concept of death; it is sufficient for the ant to sense—either visibly or perhaps chemically—what *we* would describe, *de re*, as a dead ant. For a computational theory of the *de re/de dicto* distinction, see [Wiebe and Rapaport, 1986], [Rapaport et al., 1997].

ant needs none of that. The teleological description helps *us* describe and perhaps understand the ant's behavior; it doesn't help the ant.

The same is true in my spreadsheet example. Knowing that I am adding helps *me* understand what I am doing when I fill the spreadsheet cells with certain values or formulas. But the spreadsheet does its thing without needing that knowledge.

And it is true for Searle in the Chinese Room [Searle, 1980]: Searle-in-the-room might not understand what he is doing, but he *is* understanding Chinese.[37] Was Searle-in-the-room simply told, "Follow the rule book!"? Or was he told, "To understand Chinese, follow the rule book!"? If he was told the former (which seems to be what Searle-the-author had in mind), then, (a) from a narrow, internal, first-person point of view, Searle-in-the-room can truthfully say that he doesn't know what he is doing (in the wide sense). In the narrow sense, he does know that he is following the rule book, just as I didn't know that I was using a spreadsheet *to add*, even though I knew that I was filling certain cells with certain values. And (b) from the wide, external, third-person point of view, the native-Chinese-speaking interrogator can truthfully tell Searle-in-the-room that he *is* understanding Chinese. When Searle-in-the-room is told that he has passed a Turing test for understanding Chinese, he can—paraphrasing Molière's bourgeois gentleman—truthfully admit that he was speaking Chinese but didn't know it.[38]

Here is a nice description of computation that matches the Chinese-Room set-up:

> Consider again the scenario described by Turing: an idealized human computor[39] 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 deter-**

[37] Too much has been written on the Chinese-Room Argument to cite here, but [Cole, 1991], my response to Cole in [Rapaport, 1990], [Rapaport, 2000], and [Rapaport, 2006, 390–397] touch on this particular point.

[38] "*Par ma foi! il y a plus de quarante ans que je dis de la prose sans que j'en susse rien, et je vous suis le plus obligé du monde de m'avoir appris cela.*" "Upon my word! It has been more than forty years that I have been speaking prose without my knowing anything about it, and I am most obligated to you in the world for having apprised me of that." (my translation) (http://en.wikipedia.org/wiki/Le_Bourgeois_gentilhomme).

[39] See explanation of this term in §3.

> **mining 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, pp. 261–262; my boldface, Rescorla's italics]

Without the boldfaced clauses, this is a nice description of the Chinese Room. The difference is that, in the Chinese Room, 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 interpretation of the native speaker outside the room. I also claim that the boldfaced clauses are irrelevant to the computation itself; that is the lesson of Turing's "strange inversion".

These examples suggest that the user-manual/external world interpretation is not necessary. Algorithms *can* be teleological, and their being so can help cognitive agents who execute them to more fully understand what they are doing. But they don't *have to* be teleological.[40]

4.3 Can Algorithms Have More than One Purpose?

In addition to being teleological, algorithms seem to be able to be *multiply* teleological, as in the chess-war example and its kin. That is, there can be algorithms of the form:

> To accomplish G_1, do P.

and algorithms of the form:

> To accomplish G_2, do P.

[40]This touches on another philosophical issue in cognitive science: the role of "embodied cognition". As [Adams, 2010, p. 619] describes it,

> On a non-embodied approach, the sensory system informs the cognitive system and the motor system does the cognitive system's bidding. There are causal relations between the systems but the sensory and motor systems are not constitutive of cognition. For embodied views, the relation to the sensori-motor system to cognition is constitutive, not just causal.

Compare this paraphrase: On a non-teleological approach, inputs from the external world inform a Turing machine and outputs to the external world do the Turing machine's bidding. There are causal or semantic relations between the systems but the semantic interpretations of the inputs and outputs are not constitutive of computation. For teleological views, the relation to the semantic interpretations of the inputs and outputs is constitutive.

where $G_1 \neq G_2$, and where G_2 does not subsume G_1 (or vice versa), although the *P*s are the same. In other words, what if doing *P* accomplishes both G_1 and G_2? How many algorithms do we have in that case? Two? (One that accomplishes G_1, and another that accomplishes G_2, counting teleologically, or "widely"?) Or just one? (A single algorithm that does *P*, counting more narrowly?)

Were de Bruijn and the chemists talking about the same thing? On the teleological (or wide) view, they weren't; on the narrow view, they were. Multiple teleologies are multiple realizations of an algorithm narrowly construed: 'Do *P*' can be seen as a way to algorithmically implement the higher-level "function" (mathematical *or* teleological) of accomplishing G_1 *as well as* G_2. E.g., executing a particular subroutine in a given program might result in checkmate or winning a battle. Viewing multiple teleologies as multiple realizations (multiple implementations) can also account for hollandaise-sauce failures on the Moon, which could be the result of an "implementation-level detail" [Rapaport, 1999] that is irrelevant to the abstract, underlying computation.

4.4 What If *G* and *X* Come Apart?

What if "successfully" executing *P fails* to accomplish goal *G*? This could happen for external, environmental reasons (hence my use of 'wide', above). Does this mean that *G* might not be a computable task even though *P* is?

The blocks-world computer's model of the world was an incomplete, partial model; it assumed that its actions were always successful. I'll have more to say about partial models in §7. For now, the point is that this program lacked feedback from the external world. There was nothing wrong with the environment, as there is in the lunar hollandaise-sauce case; rather, there was incomplete information about the environment.

Rescorla's GCD computers do "different things" *by* doing the "same thing". The difference is not in *how* they are doing what they are doing, but in the interpretations that *we* users of the machines give to their inputs and outputs. Would [Hill, 2015] say that the procedure encoded in that Scheme program was therefore not an algorithm?

What is more central to the notion of "algorithm"?: All of parts 1–3 in our informal characterization in §4 ("To *G*, do *P*"), or just parts 1–2, i.e., without the bracketed goals (just "Do *P*")? Is the algorithm the narrow, non-teleological, "purposeless" (or non-purposed) entity? Or is the algorithm the wide, intentional, teleological (i.e., goal-directed) entity? On the narrow view, the wargame and chess algorithms are just *one* algorithm, the hollandaise-sauce recipe *does* work on the Moon (its computer program might be logically verifiable even if it fails to make hollandaise sauce), and the "two" GCD programs are also just *one* algorithm that does its thing correctly (but only we base-10 folks can *use* it to compute GCDs). On the wide view, the wargame and chess programs are *two*, distinct algorithms, the hollandaise-sauce recipe *fails* on the Moon (despite the fact that the program might have been verified—shades of the Fetzer controversy that we will discuss in §7!), and the Scheme program when fed base-13 numerals is doing something *wrong* (in particular, its "remainder" subroutine is incorrect).[41]

These examples suggest that the wide, goal-directed nature of algorithms teleologically conceived is due to the interpretation of their input and output. As Shagrir & Bechtel put it, Marr's "algorithmic level… is directed to the *inner working* of the mechanism.… The computational level looks *outside*, to identifying the function computed and relating it to the environment in which the mechanism operates" [Shagrir and Bechtel, 2015, §2.3].

There is a way to combine these insights: Hill's formulation of the teleological or intentional nature of algorithms had two parts, a teleological "preface" specifying the task to be accomplished, and a statment of the algorithm that accomplishes it. One way to clarify the nature of Marr's "computational" level is to split it into its "why" and its "what" parts. The "why" part is the task to be accomplished. The "what" part can be expressed "computationally" (I would say "functionally") as a mathematical function (possibly, but not necessarily, expressed in "why" terminology), but it can also be expressed *algorithmically*. Finally, the algorithm can be implemented. So, we can distinguish the following *four* Marr-like levels of analysis:

"Computational"-What Level
Do $f(i) = o$

"Computational"-Why Level
To accomplish G, do $f(i) = o$

Algorithmic Level
To accomplish G, do $A_f(i) = o$

Implementation Level
To accomplish G, do $I(A_f)(i) = o$

where:

- G is the task to be accomplished or explained, expressed in the language of the external world, so to speak;

[41] At least as Rescorla describes it; it does the *right* thing on the Shapiro-Rapaport interpretation discussed in §3.4.

- f is an *i*nput-*o*utput function that accomplishes G, expressed either in the same language or perhaps expressed in purely mathematical language;

- A_f is an algorithm that implements f (i.e., it is an algorithm that has the same input-output behavior as f); and

- and I is an implementation (perhaps in the brain or on some computer) of A).[42]

[Shagrir and Bechtel, 2015, §4] say that "The *what* aspect [of the "computational" level] provides a description of the mathematical function that is being computed. The *why* aspect employs the contextual constraints in order to show how this function matches with the environment." These seem to me to nicely describe the two clauses of what I call the "computational-why" level above.

And this raises another interesting question:

5 Do We Compute with Symbols or with Meanings?

5.1 Where Does Context Belong?

Analogous (if not identical) problems arise elsewhere: Is "3 is an integer power of 2" true? [Stewart, 2000, p. 98] (cf. [Rescorla, 2013]). Not in base-10 arithmetic. But it is true in the integers modulo 5.[43]

One can take two attitudes towards situations like this. First, one can say that the truth-values differ because the external context is different in the two cases, in much the same way that 'I like water' is true on Earth (where 'water' refers to H_2O) but (say) false on Twin Earth (where 'water' refers to XYZ). Alternatively, one can say that the original proposition was not stated carefully enough: It should have been something like: "3_{10} is an integer power of 2_{10}" or, perhaps, "In base-10, 3 is an integer power of 2" (with a corresponding pair of propositions for the $\mathbb{Z}/5$ case).[44] We should have incorporated the information about the external con-

[42][Egan, 1995, p. 187, n. 8], citing Colin McGinn, notes that even what I am calling the "computational"-what level can be phrased intentionally as, e.g., "To compute the Laplacean of a Gaussian, do $f(i,o)$." So perhaps there is a level intermediate between the what- and why-levels, something along these lines: "To accomplish $A_f(i,o)$, do $A_f(i,o)$", where A is expressed in pure Turing-machine language. Note, too, that *both* clauses can vary independently: Not only can f implement many different Gs (as in the chess-wargame example), but G can be implemented by many different A_fs.

[43]Because $(2_{10})^{(1_{10})} = 2_{10} < 3_{10} < (2_{10})^{(2_{10})} = 4_{10}$. But $2^3 = 8 = 3_{\mathbb{Z}/5}$.

[44]There are many other options: "The *number* represented by the *numeral* '3' in base-10 is an integer power of the *number* represented by the *numeral* '2' in base-10". Or "The number canonically representable as $\{\{\{\emptyset\}\}\}$ (etc.)".

text *into* the proposition. In other words, the "external context" should have been "internalized".[45] (We will return to this in §5.4.)

5.2 What Is This Turing Machine Doing?

Here is a question to consider: Are such propositions about numbers? Or are they about numerals? What do Turing machines compute with? For that matter, what do *we* compute with? This is not the place for us to get into a discussion of nominalism in mathematics, though it is worth pointing out that our common thread seems to lead us there.

Both Christopher Peacocke and Rescorla remind us that

> 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.[46] [Rescorla, 2007, p. 253]

Once again, we see that it is necessary to interpret the strokes.

Here is [Peacocke, 1999]'s example: Suppose that we have a Turing machine that outputs a copy of the input appended to itself (thus doubling the number of input strokes): input '/', output '//'; input '//', output '////', and so on. What is our Turing machine doing? Isn't "outputting a copy of the input appended to itself" the most neutral description? After all, that describes *exactly* what the Turing machine is doing, leaving the interpretation up to the observer. If we had come across that Turing machine in the middle of the desert and were trying to figure out what it does, something like that would be the most reasonable answer. *Why* someone might want a copy-appending Turing machine is a different matter that probably *would* require an interpretation of the strokes. But that goes far beyond what the Turing machine is doing.

As we saw in §3.4, Rescorla offered three interpretations of the strokes. Do we really have one machine that does three different things? 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; at least, that's what Cleland said about the hollandaise sauce recipe.

[45] This is something that I have argued is necessary for computational cognition; see, e.g., [Rapaport, 2006, pp. 385–387], [Rapaport, 2012, §3].

[46] Here, by the way, we have an interesting difference between Turing machines and their logical equivalents in the Computability Thesis, such as the λ-calculus or recursive-function theory, because the latter deal with functions and numbers, not symbols for them.

Piccinini says much the same thing:

> 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. [Piccinini, 2006, §2]

By 'individuated', Piccinini is talking about how one decides whether what appear to be two programs (say, one for a wargame 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 wargame 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, 2006, §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 semantic interpretations).

Here is a related issue, showing our thread running through action theory: Given a calculator that I use to add two numbers, how would you describe my behavior? Am I pushing certain buttons in a certain sequence? (A "syntactic", narrow, internal answer: I am "doing *P*".) Or am I adding two numbers? (A teleological, "semantic", wide, external answer: I am accomplishing *G*.) Or am I adding two numbers *by* pushing those buttons in that sequence? (A teleological (etc.) answer, together with a syntactic description of how I am doing it: I am accomplishing *G*, by doing *P*.) [Rapaport, 1990], [Rapaport, 1993]. This is the same situation that we saw in the spreadsheet example (and we will see it again when we discuss named subroutines in §5.3).

Clearly, in some sense, all of these answers are correct, merely(?) focusing on different aspects of the situation. But there is a further question: *Why* (or how) does a Turing machine's printing and moving thus and so, or my pushing certain calculator buttons thus and so, result in adding two numbers? And the answer to that seems to require a semantic interpretation. This is the kind of question that Marr's "computational" level is supposed to respond to.

If I want to know which Turing machine this is, I should look at the internal mechanism (roughly, Dennett's "design" stance [Dennett, 1971]) for the answer [Piccinini, 2006]. But if I'm interested in buying a chess program (rather than a wargame simulator), then I need to look at the external/inherited/wide semantics [Cleland, 1993].

> 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, 2014, p. 181]

On this view, the hollandaise-sauce-making computer does its thing whether it's on 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. (I have been told that packages of cake mix that are sold in mile-high Denver come with alternative directions).

Here is another nice example:

> 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 woven. In other words, the weaving process is insensitive to the properties of the input. [Piccinini, 2008, p. 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 aspect of our thread reappears 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? For a survey, see [Horsten, 2015, §4].)

5.3 Syntactic Semantics

'Syntax' is usually used in the narrow sense of the grammar of a language, and 'semantics' is usually understood as the meanings of the morphemes, words, and sentences of a language. Following [Morris, 1938, pp. 6–7], I prefer to understand 'syntax' very broadly to be the study of the properties of, and relations among, the elements of a single set (or formal system), and 'semantics' very broadly to be the study of the relations between any two sets whatsoever (each with its own syntax).[47] Syntax is concerned with "intra-system" properties and relations; semantics is concerned with "extra-system" relations (where the "system" in question is the "syntactic" domain), or, viewed "*sub specie aeternitatis*", it is concerned with "inter-system" relations (i.e., relations between two domains, one of which is taken as the syntactic domain and the other as a semantic domain).

So, one way to answer the questions at the end of §5.2 is by using an external semantic interpretation: These Turing-machine operations or those button presses (considered as being located in a formal, syntactic system of Turing-machine operations or button pressings) can be associated with numbers and arithmetical operations on them (considered as being located in a distinct, Platonic (or at least external) realm of mathematical entities).[48] In the formulation "To *G*, do *P*", *P can* be *identified* syntactically (at the "computational-what" level), but *G needs* to be identified semantically—and then *P* can be *interpreted* semantically in *G*'s terms (at the "computational-why" level).

But there is another way to answer these questions, by using an "internal" kind of semantics, the kind that I have called "syntactic semantics". Syntactic semantics arises when the semantic domain of an external semantic interpretation is "internalized" into the syntactic domain. In that way, the previous semantic relations between the two previously independent domains have become relations within the new unified domain, turning them into syntactic relations.[49] Syntactic semantics is akin to (if not identical with) what Rescorla has called "indigenous semantics" [Rescorla, 2012], [Rescorla, 2014]. One possible difference between them is that

[47] So, the grammar of a language—syntax in the narrow sense—is the study of the properties of, and relations among, its words and sentences. And their referential meanings are given by a semantic interpretation relating the linguistic items to concepts or objects. For more details, see [Rapaport, 2012, §3.2].

[48] Which realm, incidentally, has its own organization in terms of properties and relations among its entities, i.e., its ontology, which can be thought of as its own "syntax" [Rapaport, 2006, p. 392]. Similarly, the organization of the syntactic realm, in terms of the properties and relations among its entities, can be considered, at least in the mental case, as an "epistemological ontology" [Rapaport, 1986a]. Ontology is syntax (by the definition of 'syntax' given here). Relations between *two* domains, each with its own syntax (or ontology) is semantics.

[49] [Rapaport, 1988], [Rapaport, 1995], [Rapaport, 2006], [Rapaport, 2012]; see also [Kay, 2001].

my version emphasizes the importance of *conceptual-role* semantics (to be distinguished from, but including, *inferential-role* semantics) [Rapaport, 2002], whereas Rescorla's version emphasizes *causal* relations.[50]

Without going into details (some of which are spelled out in the cited papers), let me note here that one way to give this kind of semantics is in terms of (named) subroutines (which accomplish subtasks of the overall algorithm). We can identify collections of statements in a program that "work together", then package them up, name the package, and thus identify subtasks.

E.g., the following Logo program draws a unit square by moving forward 1 unit, then turning 90 degrees right, and doing that 4 times:

```
repeat 4 [forward 1 right 90]
```

But Logo won't "know" what it means to draw a square unless we tell it that

```
to square
repeat 4 [forward 1 right 90]
end
```

Of course, the Logo program still has no understanding in the way *we* do of what a square is. It is now capable only of associating that newly defined symbol ('square') with a certain procedure. The symbol's meaning for *us* is its *external* semantics; the word's meaning (or "meaning"?) for the Logo program is its internal "syntactic semantics" due to its relationship with the body of that program. Another example is the sequence of instructions "turnleft; turnleft; turnleft", in *Karel the Robot* [Pattis et al., 1995], which can be packaged up and named "turnright":

```
DEFINE-NEW-INSTRUCTION turnright AS
BEGIN
turnleft;turnleft;turnleft
END
```

Notice here that Karel still can't "turn *right*" (i.e., 90 deg clockwise); it can only turn left three times (i.e., 270 deg counterclockwise).

There is a caveat: Merely *naming* a subroutine does not automatically endow it with the meaning of that name, as [McDermott, 1980] makes clear. But the idea that connections (whether conceptual, inferential, or causal) can be "packaged" together is a way of providing "syntactic" or "indigenous" semantics. If the name is associated with objects that are *external* to the program, then we have external/wide/inherited/extra-system semantics. If it is associated with objects

[50][Egan, 1995, p. 181]'s "structural properties" and [Bickle, 2015, esp. §5]'s description of "causal-mechanistic explanations" in neuroscience may also be "syntactic/indigenous" semantics.

internal to the program, then we have internal/narrow/syntactic/indigenous/intra-system semantics. *Identifying* subroutines is syntactic; *naming* them leads to semantics: If the name is externally meaningful to a user, because the user can associate the name with other external concepts, then we have semantics in the ordinary sense (subject to the caveat in the previous footnote); if it is internally meaningful to the computer, because the computer can associate the name with other internal names, then we have "syntactic" or "indigenous" semantics.

5.4 Internalization

Another way to get syntactic semantics is by "internalization". External semantic relations between the elements of two domains (a "syntactic" domain described syntactically and a "semantic" domain also described syntactically (or, if you prefer, "ontologically") can be turned into internal syntactic relations ("syntactic semantics") by internalizing the semantic domain into the syntactic domain. After all, if you take the union of the syntactic and semantic domains, then all formerly external semantic relations are now internal syntactic ones.

And one way that this happens for us cognitive (and arguably computational) agents is by sensory perception, which is a form of input encoding. For animal brains, perception interprets signals from the external world into a biological neural network. For a computer that accepts input from the external world, the interpretation of external or user input as its internal switch settings (or inscriptions on a Turing-machine tape) constitutes a form of perception. Both are forms of what I am calling "internalization". As a result, the interpretation becomes part of the computer's or the brain's intra-system, syntactic/indigenous semantics. (See, e.g., [Rapaport, 2012] for further discussion and refereces.)

My colleague Stuart C. Shapiro advocates internalization in the following form:[51]

Shapiro's Internalization Tactic

Algorithms *do* take the teleological form, "To accomplish *G*, do *P*",
but *G* must include *everything* that is relevant:

- To make hollandaise sauce *on Earth*, do *P*.
- To find the GCD of 2 integers *in base-10*, do *Q*.
- To play chess, do *R*, *where R's variables range over chess pieces and a chess board*.

[51] Personal communication. A similar point was made in [Smith, 1985, p. 24]: "as well as modelling the artifact itself, you have to model the relevant part of the world in which it will be embedded."

- To simulate a wargame battle, do *R*, *where R's variables range over soldiers and a battlefield*.

And the proper location for these teleological clauses is in the preconditions and postconditions of the program. Once they are located there, they can be used in the formal verification of the program, which proceeds by proving that, *if the preconditions are satisfied*, then the program will accomplish its goal *as articulated in the postconditions*. This builds the external world (and any attendant external semantics) *into* the algorithm. As Lamport has said, "There is no easy way to ensure a blueprint stays with a building, but a specification can and should be embedded as a comment within the code it is specifying" [Lamport, 2015, 41]. The separability of blueprint from building is akin to the separabiiity of G from P; embedding a specification into code as (at least) a comment is to internalize it as a pre- or postcondition.[52]

As I suggested in §4.1, we can avoid having Cleland's hollandaise-sauce recipe output a messy goop by limiting its execution to one location (Earth, say) without guaranteeing that it will work elsewhere (on the Moon, say). This is no different from a partial mathematical function that is silent about what to do with input from outside its domain, or from an algorithm for adding two integers that specifies no particular behavior for non-numerical input.[53] Another way is to use the "Denver cake mix" strategy: The recipe or algorithm should be expressed conditionally: If location = Earth, then do P; if location = Moon, then do Q (where Q might be the output of an error message).

A third way is to write an algorithm (a recipe) that tells us to mix eggs and oil *until an emulsion is produced*, and that outputs hollandaise sauce. On Earth, an emulsion is indeed produced, and hollandaise sauce is output. But on the Moon, 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 (cf. [Cooper, 2012, p. 78]). 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, to which we now turn.

[52] Note the similarity of (a) internalizing external/inherited semantics into internal/syntactic semantics to (b) the Deduction Theorem in logic, which can be thought of as saying that a premise of an argument can be "internalized" as an antecedent of the argument's conclusion:
$P \vdash C \;\Leftrightarrow\; \vdash (P \rightarrow C)$.

[53] "Crashing" is a well-defined behavior if the program is silent about illegal input. More "well-behaved" behavior requires some kind of error handling.

6 Interactive (Hyper?)computation

There are many varieties of hypercomputation, i.e., "the computation of functions that cannot be" computed by a Turing machine [Copeland, 2002, p. 461]. The notion is put forth as a challenge to the Computability Thesis. Many varieties of hypercomputation involve such aracana as computers operating in Malament-Hogarth spacetime.[54] Here, I want to focus on one form of hypercomputation that is more down to earth. It goes under various names (though whether there is a single *it* with multiple names, or multiple *it*s is an issue that I will ignore here): 'interactive computation' [Wegner, 1995, p.45], 'reactive computation' (Amir Pneuli's term; see [Hoffmann, 2010]), or 'oracle computation' (Turing's term; useful expositions are in [Feferman, 1992], [Davis, 2006], [Soare, 2009], and [Soare, 2013]).

Remember that Turing machines do not really *accept* input from the external world; the input to a function computed by a Turing machine is *pre-stored*—already printed on its tape; Turing machines work "offline". Given such a tape, the Turing machine computes (and, hopefully, halts). A student in an introductory programming course who is asked to write an *interactive* program that takes as input two integers *chosen randomly by a user* and that produces as output their GCD has not written a Turing-machine program. The student's program begins with a Turing machine that prints a query on its tape and halts; the user then does the equivalent of supplying a new tape with the user's input pre-stored on it; and then a(nother) Turing machine uses that tape to compute a GCD, query another input, and (temporarily) halt.

Each run of the student's program, however, could be considered to be the run of a Turing machine. But the read-loop in which the GCD computation is embedded in that student's program takes it out of the realm of a Turing machine, strictly speaking.

That hardly means that our freshman student has created a hypercomputer that computes something that a Turing machine cannot compute. Such interactive computations, which are at the heart of modern-day computing, were mathematically modeled by Turing using his concept of an oracle. Our freshman's computer program's query to the user to input another pair of integers is nothing but a call to an oracle that provides unpredictable, and possibly uncomputable, values. (Computer users who supply input are oracles!)

Now, many interactive computations can be *simulated* by Turing machines, simply by supplying all the actual inputs on the tape at the start. The Kleene Substitution Property[55] states that data can be stored effectively (i.e., algorithmically)

[54] http://en.wikipedia.org/wiki/Hypercomputation

[55] Also called the Kleene Recursion Theorem [Case, nd].

in programs; the data need not be input from the external world. A typical interactive computer might be an ATM at the bank. No one can predict what kind of input will be given to that ATM on any given day; but, at the end of the day, all of the day's inputs are known, and that ATM can be simulated by a TM.

But that is of no help to anyone who wants to use an ATM on a daily basis. Computation in the wild must allow for input from the external world (including oracles).

And that is where our thread re-appears: Computation must interact with the world. A computer without physical transducers that couple it to the environment [Sloman, 2002, §5, #F6, pp. 17–18] would be solipsistic. The transducers allow for perception of the external world (and thereby for interactive computing), and they allow for manipulation of the external world (and thereby for computers—robots, including computational chefs—that can make hollandaise sauce). But the computation and the external-world interaction (the interpretation of the computer's output in the external world) are separable and distinct. And there can, therefore, be slippage between them (leading to such things as blocks-world and hollandaise-sauce failures), mulitiple interpretations (chess vs. wargame), etc.

7 Program Verification and the Limits of Computation

Let's consider programs that specify physical behaviors a bit further. In a classic and controversial paper, James H. Fetzer [Fetzer, 1988] argued to the effect that, given a computer program, you might be able to logically prove that its (primitive) instruction RINGBELL will be executed, but you cannot logically prove that the physical bell will actually ring (a wire connecting computer and bell might be broken, the bell might not have a clapper, etc.). Similarly, we might be able to logically prove that the hollandaise-sauce program will execute correctly, but not that hollandaise sauce will actually be produced. For Fetzer and Cleland, it's the bells and the sauce that matter in the real world: Computing is about *the world.*

Donald Mackenzie agrees [MacKenzie, 1992, p. 1066]:

> . . . 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".

Similarly, a formal proof of a theorem only establishes its truth *relative to* the truth of its axioms, not its "absolute" truth [Rapaport, 1984, p. 613].

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, p. 1066].

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!

Lamport has recently said much the same thing:

> 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]

The goal of an algorithm can be expressed in its specification. This is why you wouldn't *have* to read the code to find out what it does. Of course, if the specification has been internalized into the code, you might be able to. But it's also why you can have a chess program that is also a wargame simulator: They might have different specifications but the same code.

> . . . 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].

This is the case for reasons that are related to the Church-Turing Computability Thesis: 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 Computability Thesis, the problem concerns the informality of the language of algorithms versus the formality of the language of Turing machines. 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 of the hardware is correct; and that can't be had.

It can't be had, because, to have it, we would need *another* formal description of the hardware to compare with the formal description that we were trying to verify. And that leads to a Zeno- or Bradley-like infinite regress. (We can come "close, but no cigar".)

Brian Cantwell Smith has articulated this problem most clearly [Smith, 1985]. For him, computing is about a *model of* the world. 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 is the opposite of implementation. It is the removal of "irrelevant" implementation details. His point is that computers only deal with *their representations of* these *abstract models of* the real world. They are *twice* removed from reality.[56]

All models are necessarily "partial", hence abstract. But action is *not* abstract: You *and* the computer must act *in* the complex, real world, and in real time. Yet such real-world action must be based on *partial models* of the real world and inferences based on incomplete and noisy information (cf. Herbert Simon's notion of "bounded rationality" and the need for "satisficing" [Simon, 1996]). 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 our 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). 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, on Smith's view, is that mathematical model theory only discusses the relation between two *descriptions*: the model itself (which is a partial description of the world) and a description of the model. It does not discuss the relation between the model and the world; there is an unbridgeable gap. In Kantian fashion, a model is like eyeglasses for the computer, through which it sees the world, and it cannot see the world without those glasses. The model *is* the world as far as the computer can see. The model is the world *as* the computer sees it.

Both Smith and Fetzer agree that the program-verification project fails, but for slightly different reasons: For Fetzer (and Cleland), computing is about the *world*; it is external and contextual. Thus, computer programs can't be verified, because

[56]"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].

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.[57] Note that Smith also believes that computers must act *in* the real world, but it is their abstract narrowness that isolates them from the concrete, real world at the same time that they must act in it.

The debate over 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, p. 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 Computability Thesis problem.

Smith's gap is due, in part, to the fact that specifications are abstractions. How does one know if something that has been omitted from the specification is important or not? This is why "abstraction is an art", as Lamport said, and why there's no guarantee that the model is correct (in the sense that it matches reality).

8 Conclusion

Where do we stand? First, I have not attempted in this overview to resolve these issues. I am still struggling with them, and my goal was to convince you that they are interesting, and perhaps important, issues that are widespread throughout the philosophy of computer science and beyond, to issues in the philosophy of mind, philosophy of language, and the ethics and practical uses of computers. But I think we can see opportunities for some possible resolutions.

We can distinguish between the question of which Turing machine a certain computation is and the question of what goal that computation is trying to accomplish. Both questions are important, and they can have very different answers. Two computations might implement the same Turing machine, but be designed to accomplish different goals.

And we can distinguish between two kinds of semantics: wide/external/extrinsic/-inherited and narrow/internal/intrinsic/"syntactic"/indigenous. Both kinds exist, have interesting relationships and play different, albeit complementary, roles.

[57]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, pp. 163–164].

Algorithms narrowly construed (minus the teleological preface) is what is studied in the mathematical theory of computation. To decide whether a task is computable, we need to find an algorithm that can accomplish it. Thus, we have two separate things: an algorithm (narrowly construed, if you prefer) and a task. Some algorithms can accomplish more than one task (depending on how their inputs and outputs are interpreted by external/inherited semantics). Some algorithms may fail, not because of a buggy, narrow algorithm, but because of a problem at the real-world interface. That interface is the (algorithmic) coding of the algorithm's inputs and outputs, typically through a *sequence* of transducers at the real-world end (cf. [Smith, 1987]). Physical signals from the external world must be transduced (encoded) into the computer's switch-settings (the physical analogues of a Turing machine's '0's and '1's), and the output switch-settings have to be transduced (decoded) into such real-world things as displays on a screen or physical movements by a robot.

At the real-world end of this continuum, we run into Smith's gap. From the narrow algorithm's point of view, so to speak, it might be able to asymptotically approach the real world, in Zeno-like fashion, without closing the gap. But, just as someone trying to cross a room by only going half the remaining distance at each step *will* eventually cross the room (though not because of doing it that way), so the narrow algorithm implemented in a physical computer *will* do something in the real world. Whether what it accomplishes was what its programmer intended is another matter. (In the real world, there are no "partial functions"!)

One way to make teleological algorithms more likely to be successful is by Shapiro's strategy: Internalizing the external, teleological aspects into the pre- and post-conditions of the (narrow) algorithm, thereby turning the external/inherited semantic interpretation of the algorithm into an internal/indigenous syntactic semantics.

What Smith shows is that the external semantics for an algorithm is never a relation directly with the real world, but only to a *model* of the real world. That is, the real-world semantics has been internalized. But that internalization is necessarily partial and incomplete.

There are algorithms *simpliciter*, and there are algorithms *for accomplishing a particular task*. Alternatively, *all* algorithms accomplish a particular task, but some tasks or more "interesting" than others. The algorithms whose tasks are not currently of interest may ultimately *become* interesting when an application is found for them, as in the case of non-Euclidean geometry. Put otherwise, the algorithms that do not accomplish tasks may ultimately be used to accomplish a task.

Algorithms that *explicitly* accomplish a(n interesting) task can be converted into algorithms whose tasks are *not* explicit in that manner by internalizing the task into the algorithm narrowly construed. This can be done by internalizing the task, perhaps in the form of pre-and postconditions, perhaps in the form of named subroutines (modules). Both involve syntactic or indigenous semantics.

As promised, I have raised more questions than I have answered. But that's what philosophers are supposed to do![58]

References

[Adams, 2010] Adams, F. (2010). Embodied cognition. *Phenomenology and Cognitive Science*, 9:619–628. DOI 10.1007/s11097-010-9175-x.

[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, 2004].

[Anderson, 2015] Anderson, B. L. (2015). Can computational goals inform theories of vision? *Topics in Cognitive Science*. DOI: 10.111/tops.12136.

[Bickle, 2015] Bickle, J. (2015). Marr and reductionism. *Topics in Cognitive Science*. DOI: 10.1111/TOPS.12134.

[Case, nd] Case, J. (nd). Motivating the proof of the Kleene recursion theorem. http://www.eecis.udel.edu/∼case/papers/krt-self-repo.pdf. Accessed 20 March 2015.

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

[Castañeda, 1989] Castañeda, H.-N. (1989). Direct reference, the semantics of thinking, and guise theory (constructive reflections on David Kaplan's theory of indexical reference). In Almog, J., Perry, J., and Wettstein, H., editors, *Themes from Kaplan*, pages 105–144. Oxford University Press, New York.

[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.

[Cleland, 1993] Cleland, C. E. (1993). Is the Church-Turing thesis true? *Minds and Machines*, 3(3):283–312.

[58] I am grateful to Robin K. Hill and to Stuart C. Shapiro for discussion and comments on earlier drafts.

[Cleland, 2002] Cleland, C. E. (2002). On effective procedures. *Minds and Machines*, 12(2):159–179.

[Cole, 1991] Cole, D. (1991). Artificial intelligence and personal identity. *Synthese*, 88(3):399–417.

[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, 2002] Copeland, B. J. (2002). Hypercomputation. *Minds and Machines*, 12(4):461–502.

[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.

[Davis, 2006] Davis, M. (2006). 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).

[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).

[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.

[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.

[Egan, 1991] Egan, F. (1991). Must psychology be individualistic? *Philosophical Review*, 100(2):179–203.

[Egan, 1995] Egan, F. (1995). Computation and content. *Philosophical Review*, 104(2):181–203.

[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).

[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 Exporations*, pages 314–348. Walter de Gruyter, Berlin.

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

[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.

[Forsythe, 1968] Forsythe, G. E. (1968). Computer science and education. *Proceedings IFIP 68 Cong*, pages 92–106.

[Frenkel, 2013] Frenkel, E. (2013). *Love and Math: The Heart of Hidden Reality*. Basic Books, New York.

[Goldfain, 2006] Goldfain, A. (2006). Embodied enumeration: Appealing to activities for mathematical explanation. In Beetz, M., Rajan, K., Thielscher, M., and Rusu, R. B., editors, *Cognitive Robotics: Papers from the AAAI Workshop*

(CogRob2006), Technical Report WS-06-03, pages 69–76. AAAI Press, Menlo Park, CA.

[Goldfain, 2008] Goldfain, A. (2008). A computational theory of early mathematical cognition. PhD dissertation (Buffalo: SUNY Buffalo Department of Computer Science & Engineering), http://www.cse.buffalo.edu/sneps/Bibliography/GoldfainDissFinal.pdf.

[Grossmann, 1974] Grossmann, R. (1974). *Meinong*. Routledge and Kegan Paul, London.

[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).

[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.

[Hayes, 2004] Hayes, B. (2004). Small-town story. *American Scientist*, pages 115–119. https://www.americanscientist.org/issues/pub/small-town-story (accessed 14 March 2015).

[Hill, 2015] Hill, R. K. (2015). What an algorithm is. *Philosophy and Technology*. DOI 10.1007/s13347-014-0184-5.

[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).

[Hofstatder, 1980] Hofstatder, D. R. (1980). Review of [Sloman, 1978]. *Bulletin of the American Mathematical Society*, 2(2):328–339. http://projecteuclid.org/download/pdf_1/euclid.bams/1183546241.

[Horsten, 2015] Horsten, L. (2015). Philosophy of mathematics. In Zalta, E. N., editor, *Stanford Encyclopedia of Philosophy*. http://plato.stanford.edu/entries/philosophy-mathematics/#StrNom.

[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.

[Kay, 2001] Kay, K. (2001). Machines and the mind. *The Harvard Brain*, Vol. 8 (Spring). http://www.hcs.harvard.edu/∼hsmbb/BRAIN/vol8-spring2001/ai.htm (accessed 17 April 2015).

[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.

[Knuth, 1973] Knuth, D. E. (1973). *The Art of Computer Programming, Second Edition*. Addison-Wesley, Reading, MA.

[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).

[Liao, 1998] Liao, M.-H. (1998). Chinese to English machine translation using SNePS as an interlingua. http://www.cse.buffalo.edu/sneps/Bibliography/tr97-16.pdf. Unpublished doctoral dissertation, Department of Linguistics, SUNY Buffalo.

[Lloyd and Ng, 2004] Lloyd, S. and Ng, Y. J. (2004). Black hole computers. *Scientific American*, 291(5):52–61.

[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.

[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.

[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.

[Marx, 1845] Marx, K. (1845). Theses on Feuerbach. https://www.marxists.org/archive/marx/works/1845/theses/theses.htm (accessed 14 March 2015).

[McDermott, 1980] McDermott, D. (1980). Artificial intelligence meets natural stupidity. In Haugeland, J., editor, *Mind Design: Philosophy, Psychology, Artificial Intelligence*, pages 143–160. MIT Press, Cambridge, MA. http://www.inf.ed.ac.uk/teaching/courses/irm/mcdermott.pdf.

[Moor, 1978] Moor, J. H. (1978). Three myths of computer science. *British Journal for the Philosophy of Science*, 29(3):213–222.

[Moor, 2003] Moor, J. H., editor (2003). *The Turing Test: The Elusive Standard of Artificial Intelligence*. Kluwer Academic, Dordrecht, The Netherlands.

[Morris, 1938] Morris, C. (1938). *Foundations of the Theory of Signs*. University of Chicago Press, Chicago.

[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).

[Newell, 1980] Newell, A. (1980). Physical symbol systems. *Cognitive Science*, 4:135–183. Accessed 15 March 2014 from http://tinyurl.com/Newell1980.

[Pattis et al., 1995] Pattis, R. E., Roberts, J., and Stehlik, M. (1995). *Karel the Robot: A Gentle Introduction to the Art of Programming, Second Edition*. John Wiley & Sons, New York.

[Peacocke, 1999] Peacocke, C. (1999). Computation as involving content: A response to Egan. *Mind & Language*, 14(2):195–202.

[Pearl, 2000] Pearl, J. (2000). *Causality: Models, Reasoning, and Inference*. Cambridge University Press, Cambridge, UK.

[Perruchet and Vinter, 2002] Perruchet, P. and Vinter, A. (2002). The self-organizing consciousness. *Behavioral and Brain Sciences*, 25(3):297–388.

[Piccinini, 2004] Piccinini, G. (2004). 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, 2006] Piccinini, G. (2006). Computation without representation. *Philosophical Studies*, 137(2):204–241. Accessed 29 April 2014 from: http://www.umsl.edu/∼piccininig/Computation_without_Representation.pdf.

[Piccinini, 2008] Piccinini, G. (2008). Computers. *Pacific Philosophical Quarterly*, 89:32–73.

[Piccinini, 2011] Piccinini, G. (2011). The physical Church-Turing thesis: Modest or bold? *British Journal for the Philosophy of Science*, 62:733–769.

[Preston, 2013] Preston, B. (2013). *A Philosophy of Material Culture: Action, Function, and Mind*. Routledge, New York.

[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).

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

[Rapaport, 1984] Rapaport, W. J. (1984). Critical thinking and cognitive development. *American Philosophical Associtation 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, 1986a] Rapaport, W. J. (1985-1986a). Non-existent objects and epistemological ontology. *Grazer Philosophische Studien*, 25/26:61–95.

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

[Rapaport, 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, 1990] Rapaport, W. J. (1990). Computer processes and virtual persons: Comments on Cole's 'Artificial intelligence and personal identity'. Technical Report 90-13, SUNY Buffalo Department of Computer Science, Buffalo. http://www.cse.buffalo.edu/∼rapaport/Papers/cole.tr.17my90.pdf.

[Rapaport, 1993] Rapaport, W. J. (1993). Because mere calculating isn't thinking: Comments on Hauser's 'Why isn't my pocket calculator a thinking thing?'. *Minds and Machines*, 3:11–20. Preprint online at http://www.cse.buffalo.edu/∼rapaport/Papers/joint.pdf.

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

[Rapaport, 2000] Rapaport, W. J. (2000). How to pass a Turing test: Syntactic semantics, natural-language understanding, and first-person cognition. *Journal of Logic, Language, and Information*, 9(4):467–490. Reprinted in [Moor, 2003, 161–184].

[Rapaport, 2002] Rapaport, W. J. (2002). Holism, conceptual-role semantics, and syntactic semantics. *Minds and Machines*, 12(1):3–59.

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

[Rapaport, 2005b] Rapaport, W. J. (2005b). Philosophy of computer science: An introductory course. *Teaching Philosophy*, 28(4):319–341. http://www.cse.buffalo.edu/∼rapaport/philcs.html.

[Rapaport, 2006] Rapaport, W. J. (2006). How Helen Keller used syntactic semantics to escape from a Chinese room. *Minds and Machines*, 16:381–436. See reply to comments, in [Rapaport, 2011].

[Rapaport, 2011] Rapaport, W. J. (2011). Yes, she was! Reply to Ford's 'Helen Keller was never in a Chinese room'. *Minds and Machines*, 21(1):3–17.

[Rapaport, 2012] Rapaport, W. J. (2012). Semiotic systems, computers, and the mind: How cognition could be computing. *International Journal of Signs and Semiotic Systems*, 2(1):32–71.

[Rapaport, 2015] Rapaport, W. J. (2015). Philosophy of computer science. Current draft in progress at http://www.cse.buffalo.edu/∼rapaport/Papers/phics.pdf.

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

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

[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.

[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, 2012] Rescorla, M. (2012). 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, 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, 2014] Rescorla, M. (2014). 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).

[Richards, 2009] Richards, R. J. (2009). The descent of man. *American Scientist*, 97(5):415–417. http://www.americanscientist.org/bookshelf/pub/the-descent-of-man (accessed 20 April 2015).

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

[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, 1990] Searle, J. R. (1990). Is the brain a digital computer? *Proceedings and Addresses of the American Philosophical Association*, 64(3):21–37.

[Shagrir, 2006] Shagrir, O. (2006). 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 and Bechtel, 2015] Shagrir, O. and Bechtel, W. (2015). Marr's computational level and delineating phenomena. *Cognitive Science*, forthcoming. Preprint (accessed 23 March 2015) at http://philsci-archive.pitt.edu/11224/1/shagrir_and_bechtel.Marr's_Computational_Level_and_Delineating_Phenomena.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.

[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).

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

[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.

[Sloman, 1978] Sloman, A. (1978). *The Computer Revolution in Philosophy: Philosophy, Science and Models of Mind*. Humanities Press, Atlantic Highlands, NJ. http://www.cs.bham.ac.uk/research/projects/cogaff/crp/.

[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.

[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.

[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, 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].

[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).

[Stewart, 2000] Stewart, I. (2000). Impossibility theorems. *Scientific American*, pages 98–99.

[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).

[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.

[Thagard, 1984] Thagard, P. (1984). Computer programs as psychological theories. In Neumaier, O., editor, *Mind, Language and Society*, pages 77–84. Conceptus-Studien, Vienna.

[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.

[Wegner, 1995] Wegner, P. (1995). Interaction as a basis for empirical computer science. *ACM Computing Surveys*, 27(1):45–48.

[Weinberg, 2002] Weinberg, S. (2002). Is the universe a computer? *New York Review of Books*, 49(16).

[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.

[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).

[Winston, 1977] Winston, P. H. (1977). *Artificial Intelligence*. Addison-Wesley, Reading, MA.

[Wolfram, 2002] Wolfram, S. (2002). *A New Kind of Science*. Wolfram Media.