10-1-2007

Computing Mechanisms

Gualtiero Piccinini
University of Missouri-St. Louis, piccininig@umsl.edu

Computing Mechanisms*

Gualtiero Piccinini†‡

This paper offers an account of what it is for a physical system to be a computing mechanism—a system that performs computations. A computing mechanism is a mechanism whose function is to generate output strings from input strings and (possibly) internal states, in accordance with a general rule that applies to all relevant strings and depends on the input strings and (possibly) internal states for its application. This account is motivated by reasons endogenous to the philosophy of computing, namely, doing justice to the practices of computer scientists and computability theorists. It is also an application of recent literature on mechanisms, because it assimilates computational explanation to mechanistic explanation. The account can be used to individuate computing mechanisms and the functions they compute and to taxonomize computing mechanisms based on their computing power.

1. Introduction. This paper contains an account of what it is for a physical system to be a computing mechanism—a system that performs computations. My motivation, namely, doing justice to the practices of computer scientists and computability theorists, is endogenous to the philosophy of computing. My account is also an application of recent literature on mechanisms (e.g., Machamer, Darden, and Craver 2000; Glennan 2002; Tabery 2004; Craver, forthcoming), because it assimilates computational explanation to mechanistic explanation. The *mechanistic account* I offer can be used to individuate computing mechanisms and the functions they

*Received August 2004; revised May 2007.

†To contact the author, please write to: University of Missouri—St. Louis, 599 Lucas Hall (MC 73), 1 University Blvd, St. Louis, MO 63121; e-mail: piccininig@umsl.edu.

‡Thanks to the many people who discussed computing mechanisms with me. For comments on previous drafts, I'm especially indebted to Carl Craver,, John Gabriel, Peter Machamer, Corey Maley, the editor, and members of the St. Louis Philosophy of Science Reading Group. Between 2002 and 2005, ancestors of this paper were presented to the Canadian Society for the History and Philosophy of Science, Computing and Philosophy (CAP@CMU), the Second Reichenbach Conference, the SSPP, the University of Pittsburgh, the University of Georgia, and Georgia State University. Thanks to the audiences for their feedback. The writing of this paper was supported in part by a grant from the University of Missouri—St. Louis.

Philosophy of Science, 74 (October 2007): 501–526. 0031-8248/2007/7404-0004$10.00

compute and to taxonomize computing mechanisms based on their computing power.

At the origin of the mechanistic account are two central theses. First, computation does not presuppose representation. Unlike most accounts in the philosophical literature, the mechanistic account does not appeal to semantic properties to individuate computing mechanisms and the functions they compute. In other words, the mechanistic account keeps the question whether something is a computing mechanism and what it computes separate from the question whether something has semantic content and what it represents. I defend this thesis in Piccinini (2004a, 2007a).

Second, computing systems are mechanisms. I explicate computational explanation in terms of mechanistic explanation. I construe mechanistic explanation as in engineering and biology. Roughly, a mechanistic explanation involves a partition of a mechanism into parts, an assignment of functions and organization to those parts, and a statement that a mechanism's capacities are due to the way the parts and their functions are organized. Given this construal, computational explanation—the explanation of a mechanism's outer capacities in terms of the inner computations it performs—is a species of mechanistic explanation. I defend this thesis in Piccinini (2004b, 2007b).

The mechanistic account flows naturally from these theses. Computing mechanisms, such as calculators and computers, are analyzed in terms of their component parts (processors, memory units, input devices, and output devices), their functions, and their organization. Those components are also analyzed in terms of their component parts (e.g., registers and circuits), their functions, and their organization. Those, in turn, are analyzed in terms of primitive computing components (logic gates), their functions, and their organization. Primitive computing components can be further analyzed mechanistically, but not by computational explanation; hence their analysis does not illuminate the notion of a computing mechanism.

I believe I have motivated the above theses sufficiently well in the cited works. In this occasion, I will articulate and defend the mechanistic account on largely independent grounds. I will argue that the mechanistic account has six desirable features. In contrast to extant philosophical accounts of computing mechanisms, these features allow the present account to explicate adequately both our ordinary language about computing mechanisms and the language and practices of computer scientists and computability theorists.

2. Features. An account of computing mechanisms that does justice to the sciences of computation should have at least the following features:

1. *Objectivity*. An account with objectivity is such that whether a system

performs a particular computation is a matter of fact. Contrary to objectivity, some authors have suggested that computational descriptions are vacuous—a matter of free interpretation rather than fact. The alleged reason is that any system may be described as performing any computation, and there is no further fact of the matter as to whether one computational description is more accurate than another (Putnam 1988; Searle 1992). This conclusion may be informally derived as follows.

Assume that a computing mechanism is a system with a mapping between a sequence of states individuated by a computational description and a sequence of states individuated by a physical description of the system (Putnam 1960, 1967, 1988). Assume, along with Putnam and Searle, that there are no constraints on which mappings are acceptable, so that any sequence of computational states may map onto any sequence of physical states of the same cardinality. If the sequence of physical states has larger cardinality, the computational states may map onto either equivalence classes or a subset of the physical states. Since physical variables can generally take real numbers as values and there are uncountably many of those, physical descriptions generally give rise to uncountably many states and state transitions. But ordinary computational descriptions contain only countably many states and state transitions. Hence, there is a mapping from any (countable) sequence of computational state transitions onto either equivalence classes or a subset of physical states belonging to any (uncountable) sequence of physical state transitions. Hence, generally, any physical system performs any computation.

If this result is sound, then empirical facts about concrete systems make no difference to what computations they perform. Both Putnam (1988) and Searle (1992) take results of this sort to trivialize the empirical import of computational descriptions. Both conclude that computationalism—the view that the brain is a computing mechanism—is vacuous. But as usual, one person's *modus ponens* is another person's *modus tollens*. I take Putnam and Searle's result to refute their assumptions about what counts as a computing mechanism.[1]

Computer scientists and engineers appeal to empirical facts about the systems they study to determine which computations are performed by which mechanisms. They apply computational descriptions to concrete mechanisms in a way entirely analogous to other bona fide scientific descriptions. Furthermore, many psychologists and neuroscientists are in the business of discovering which computations are performed by minds and brains. When they disagree, they address their opponents by mustering empirical evidence about the systems they study. Unless the prima

1. For related criticisms of Putnam and Searle, see Chrisley (1995), Chalmers (1996), Copeland (1996), and Scheutz (1999).

facie legitimacy of those scientific practices can be explained away, a good account of computing mechanisms should entail that there is a fact of the matter as to which computations are performed by which mechanisms.

2. *Explanation*. Inner computations may explain outer behaviors. Ordinary digital computers are said to execute programs, and their outer behavior is normally explained by appealing to the programs they execute. The literature on computational theories of mind contains explanations that appeal to the computations performed by the mind or brain. The same literature also contains claims that psychological capacities ought to be explained computationally, and more specifically, by program execution (e.g., Fodor 1968; Cummins 1977). A good account of computing mechanisms should say how appeals to program execution, and more generally to computation, explain the behavior of computing mechanisms. It should also say how program execution relates to the general notion of computation: whether they are the same and if not, how they are related.

3. *The right things compute*. A good account of computing mechanisms should entail that paradigmatic examples of computing mechanisms, such as digital computers, calculators, both universal and nonuniversal Turing machines, and finite state automata, compute.

4. *The wrong things do not compute*. A good account of computing mechanisms should entail that all paradigmatic examples of noncomputing mechanisms and systems, such as planetary systems, hurricanes, and digestive systems, do not perform computations.

Contrary to feature 4, many authors maintain that everything performs computations (e.g., Chalmers 1996, 331; Scheutz 1999, 191; Shagrir 2006). But contrary to their view as well as feature 3, there are accounts of computation so restrictive that under them, even many paradigmatic examples of computing mechanisms turn out *not* to compute. For instance, according to Jerry Fodor and Zenon Pylyshyn, a necessary condition for something to perform computations is that the steps it follows be caused by internal representations of rules for those steps (Fodor 1968, 1975, 1998, 10–11; Pylyshyn 1984). But nonuniversal Turing machines and finite state automata do not represent rules for the steps they follow. Hence, according to Fodor and Pylyshyn's account, they do not compute.

The accounts just mentioned lack feature 3 or 4. Why? What is wrong with that? There are borderline cases, such as lookup tables and so called analog computers. Whether those things really compute may be open to debate, and in some cases it may be open to stipulation. But there are plenty of clear cases. Digital computers, calculators, Turing machines, and finite state automata are paradigmatic computing mechanisms. They constitute the subject matter of computer science and computability theory. Planetary systems, the weather, and digestive systems are paradig-

matic noncomputing systems;[2] at the very least, it is not obvious how to explain their behavior computationally. If we can find an account that works for the clear cases, the unclear ones may be left to fall wherever the account says they do—"spoils to the victor" (Lewis 1986, 203; cf. Collins, Hall, and Paul 2004, 32).

Insofar as the assumptions of computer scientists and computability theorists ground the success of their science as well as the appeal of their notion of computation to practitioners of other disciplines, they ought to be respected. By having features 3 and 4, a good account of computing mechanisms draws a principled distinction between systems that compute and systems that do not, and it draws it in a place that fits the presuppositions of good science.

5. *Miscomputation*. Computations can go wrong. A mechanism m miscomputes just in case m is computing function f on input i, $f(i) = o_1$, m outputs o_2, and $o_2 \neq o_1$.[3] Although miscomputation has been ignored by philosophers to date, a good account of computing mechanisms should explain how it is possible for a physical system to miscompute. This is desirable because miscomputation, or more informally, making computational 'mistakes', plays an important role in computer science and its applications. Those who design and use computing mechanisms devote a large portion of their efforts to avoiding miscomputations and devising techniques for preventing them. To the extent that an account of computing mechanisms makes no sense of that effort, it is unsatisfactory.

6. *Taxonomy*. Different classes of computing mechanisms have different capacities. Logic gates can perform only trivial operations on pairs of bits. Nonprogrammable calculators can compute a finite but considerable number of functions for inputs of bounded size. Ordinary digital computers can compute any computable function on any input until they run out of memory or time. Different capacities relevant to computing play an important role in computer science and computing applications. Any account of computing mechanisms whose conceptual resources explain or shed light on those differences is preferable to an account that is blind to those differences.

To illustrate, consider Robert Cummins's account. According to Cum-

2. For evidence, see Fodor (1968, 632; 1975, 74), Dreyfus (1979, 68, 101–102), and Searle (1980, 37–38; 1992, 208).

3. Here o_1 and o_2 represent any possible outcome of a computation, including the possibility that the function is undefined for a given input, which corresponds to a nonhalting computation. Miscomputation is analogous to misrepresentation (Dretske 1986), but it is not the same. Something (e.g., a sorter) may compute correctly or incorrectly regardless of whether it represents or misrepresents anything. Something (e.g., a painting) may represent correctly or incorrectly regardless of whether it computes or miscomputes anything.

mins (1983), for something to compute, it must execute a program. He also maintains that executing a program amounts to following the steps described by the program. This leads to paradoxical consequences. Consider that many paradigmatic computing mechanisms (such as nonuniversal Turing machines and finite state automata) are not characterized by computer scientists as executing programs, and they are considerably less powerful than the mechanisms that are so characterized (i.e., universal Turing machines and idealized digital computers). Accordingly, we might conclude that nonuniversal Turing machines, finite state automata, etc., do not really compute. But this conclusion violates feature 3 (the right things compute). Alternatively, we might observe along with Cummins that all these systems do follow the steps described by a program. Hence, by Cummins's light, they execute a program, and hence they compute. But now we find it difficult to explain why they are so much less powerful than ordinary digital computers. For under Cummins's account, we cannot say that unlike digital computers, those other mechanisms lack the flexibility that comes with the capacity to execute programs. The difference between computing mechanisms that execute programs and those that do not is important to computer science and computing applications, and it should make a difference to theories of mind. We should prefer an account of computing mechanisms that honors that kind of difference to one that is blind to it.

With these features as landmarks, I proceed to formulate the mechanistic account of computing mechanisms and argue that it possesses them.

3. The Mechanistic Account.

3.1. Mechanistic Explanation. The central idea is to explicate computing mechanisms as systems subject to mechanistic explanation. By mechanistic explanation of a system X, I mean a description of X in terms of spatiotemporal components of X, their functions, and their organization, to the effect that X possesses its capacities because of how X's components and their functions are organized.[4] To distinguish systems whose capacities are subject to mechanistic explanation in the present sense from other systems, I call them mechanisms. To identify the components, functions, and organization of a system, I defer to the relevant community of scientists.

Biologists ascribe functions to types of organs (e.g., the pumping func-

4. For a similar notion of mechanistic explanation, see Bechtel and Richardson (1993), Machamer, Darden, and Craver (2000), and Glennan (2002). For a detailed and systematic account of the same notion, with emphasis on neural mechanisms, see Craver (2001, forthcoming).

tion of hearts) and engineers ascribe them to types of artifacts (e.g., the cooling function of refrigerators). Scientists differentiate between functions and other causal properties (e.g., making noise or breaking under pressure). Organs and artifacts that do not perform their functions are said to malfunction or be defective. The philosophical analysis of function ascription in biology and engineering is a controversial matter on which, for present purposes, I remain neutral.[5] Suffice it to say that biologists and engineers determine the functions of the systems and components they analyze by empirical considerations, and they use the functions and malfunctions of components to explain the activities of the containing systems.

This notion of mechanistic explanation applies to ordinary computers and other computing systems in a way that matches the language and practices of computer scientists and engineers.[6] Computing mechanisms, including computers, are mechanisms whose function is computing. Like other mechanisms, computing mechanisms and their components perform their activities *ceteris paribus*, as a matter of their function. In the rest of our discussion, we will mostly focus on their normal operation, but it is important to keep in mind that they can malfunction, break, or be malformed or defective. This will help demonstrate that the mechanistic account has feature 5 (miscomputation).

A similar notion of mechanistic explanation applies to abstract computing systems, such as (unimplemented) Turing machines. Turing machines consist of a tape divided into squares and a processing device. The tape and processing device are explicitly defined as spatiotemporal components. They have functions (storing letters; moving along the tape; reading, erasing, and writing letters on the tape) and an organization (the processing device moves along the tape one square at a time, etc.). Finally, the organized activities of their components explain the computations they perform. Abstract computing mechanisms stand to concrete ones in roughly the same relation that the triangles of geometry stand relative to concrete triangular objects. Abstract computing mechanisms may be ide-

5. William Lycan (personal communication) has called this notion of function *teleological*, and warned against confusing the teleological notion of function with any particular account of teleology, such as the etiological account. I am committed to the former (in a weak sense of 'teleology') but not the latter. Accounts of function that suit my purposes are Boorse's (2002) and Wimsatt's (2002). But I prefer to stay neutral as to the correct account of function. Representatives and discussions of the main competing accounts can be found in Allen, Bekoff, and Lauder (1998), Preston (1998), Schlosser (1998), Buller (1999), Ariew, Cummins, and Perlman (2002), and Christensen and Bickhard (2002).

6. For a standard introduction to computer organization and design, see Patterson and Hennessy (1998).

alized in various ways: they may be defined so as to (i) never break down and (ii) have properties that may be impossible to implement (such as having tapes of unbounded length).

Mechanistic explanation, unlike causal explanation simpliciter, distinguishes between a system's successes and its failures. It also distinguishes between the conditions relevant to explaining successes and failures and those that are irrelevant. This gives us the resources to distinguish the properties of a mechanism that are relevant to its computational capacities from those that are irrelevant. But mechanistic structure per se is not enough to distinguish between mechanisms that compute and mechanisms that do not. For instance, both digestive systems and computers are subject to mechanistic explanation, but it appears that only the latter compute. The main challenge for the mechanistic account is to specify mechanistic explanations that are relevant to computation.

I will now propose a way to single out computations from other capacities of mechanisms, thereby differentiating between computing mechanisms and other mechanisms. To do so, I will assume that the relevant community of scientists can identify a mechanism's functionally relevant components and properties. I will suggest criteria that should be met by the functionally relevant components and properties of a mechanism for it to count as performing computations in a nontrivial sense, and hence as being a computing mechanism. The resulting account is not intended as a list of necessary and sufficient conditions, but as an explication of the properties that are most central to computing mechanisms.

3.2. Abstract Computation. Mathematically, a computation, in the sense most directly relevant to computability theory and computer science, is defined in terms of two things: strings of letters from a finite alphabet and a list of instructions for generating new strings from old strings. The list of instructions is called a 'program'. The instructions are typically deterministic, specifying how to modify a string to obtain its successor. (In special cases, instructions may be nondeterministic, specifying which of several modifications may be made.) Given an alphabet and a list of pertinent instructions, a computation is a sequence of strings—sometimes called 'snapshots'—such that each member of the sequence is derived from its predecessor by some instruction in the list.[7]

Letters and strings thereof are often called 'symbols', because they are typically assigned semantic interpretations. But interpreting strings is not necessary for individuating them. A letter is simply a type of entity that

7. For an introduction to computability theory, including a more precise definition of computation, see Davis, Sigal, and Weyuker (1994). For the mathematical theory of strings, see Corcoran, Frank, and Maloney (1974).

(i) is distinct from other letters and (ii) may be concatenated to other letters to form lists, called 'strings'. A string is an ordered sequence of letters—it is individuated by the types of letter that compose it, their number, and their order within the string.[8]

Many interesting computations depend not only on an input string of *data*, but also on the *internal state* of the (abstract) mechanism that is said to be responsible for the computation. At least in paradigmatic cases, internal states may also be defined as strings of letters from a finite alphabet. Thus, the strings over which computations are defined—the snapshots—may specify not only the computational data, but also the relevant internal states. If internal states are relevant, at each step in the computation at least one letter in a snapshot—together, perhaps, with its position in the snapshot—specifies the internal state of the mechanism. Typically, an abstract computation begins with one initial string (input plus initial internal state), includes some intermediate strings (intermediate data plus intermediate internal states), and terminates with a final string (output plus final internal state).

For all strings from an alphabet and any relevant list of instructions, there is a general rule that specifies which function is computed by acting in accordance with a program. In other words, the rule specifies which relationship obtains between the outputs produced by modifying snapshots in accordance with the program and their respective inputs. For example, a rule may say that the outputs are a series of input words arranged in alphabetical order. Such a rule has two important features. First, it is general, in that it applies to all inputs and outputs from a given alphabet without exception. Second, it is input-specific, in that it depends on the composition of the input (the letters that compose it and their order) for its application. The rule need not return an output value for all inputs; when it does not, the (partial) function being computed is undefined for that input. Absent a way to formulate the rule independently of the program, the program itself may count as the rule.[9]

It is important to notice that mathematically, a computation is a specific type of sequence defined over a specific type of entity. Many sets do not count as alphabets (e.g., the set of natural numbers is not an alphabet

8. The standard one-dimensional notion of string can be generalized to a two-dimensional notion using graph theory (Sieg and Byrnes 1996).

9. In Turing's original formulation, all computations begin with the same input (the empty string), but there are still rules that specify which strings are produced by each computation (Turing 1936–37). More generally, it is possible to define "computations" analogous to ordinary computations but without inputs, without outputs, or without both. Since these "computations" are of little interest for present purposes, I will ignore them.

because it is infinite) and many operations do not count as computations in the relevant sense (e.g., integrating a function over a domain with uncountably many values, or generating a random string of letters [Church 1940]). The mathematical notion of computation is clear enough, but it applies directly only to abstract mechanisms. The remaining question is how to apply it to concrete mechanisms.

3.3. Digits and Primitive Computing Components. To show how a concrete mechanism can perform computations, the first step is finding a concrete counterpart to the formal notion of letter from a finite alphabet. I will call such an entity a 'digit'. A digit may be a component or state of a component of the mechanism that processes it. It may enter the mechanism, be processed or transformed by the mechanism, and exit the mechanism (to be transmitted, perhaps, to another mechanism). While inside a mechanism, a digit may be implemented as either a state or a particular that belongs to one among a finite number of mechanistically relevant types. If it is a state, at any given time it is a state of a specific component of a mechanism, such as a memory cell.

A system of digits is individuated by the digits' functional roles within a mechanism. Under normal conditions of operation, the mechanism must process tokens of the same digit type in the same way and tokens of different digit types in different ways. This condition may be characterized as follows.

It is convenient to consider strings of one digit first, leaving strings of multiple digits for later. A logic gate is a device that takes one or two input digits and returns one or two output digits as a function of its input. Logic gate computations are so trivial that they cannot be analyzed into simpler computations. For this reason, I call logic gates 'primitive computing components'. Logic gates are the computational building blocks of modern computing technology.[10]

Digits are permutable in the sense that normally, any token of any digit type may be replaced by a token of any other digit type. Functionally speaking, the components that bear digits of one type are also capable of bearing digits of any other type. For example, ordinary computer

10. Some computing mechanisms, such as old mechanical calculators, are not made out of logic gates. Their simplest computing components may manipulate strings of multiple digits, as opposed to a few separate digits, as inputs and outputs. Their treatment requires the notion of a concrete string, which is introduced below. Without loss of generality, we may consider primitive components with two inputs and one output, since primitive components with a larger number of inputs and outputs are reducible to components with only two inputs and one output. (This condition may not hold in the case of hypercomputation, which will be briefly mentioned in the next section. Here, we are focusing on ordinary, recursive, computation.)

memory cells must be able to stabilize on states corresponding to either of the two digit types—usually labeled '0' and '1'—that are manipulated by a computer. If memory cells lost the capacity to stabilize on one of the digit types, they would cease to function as memory cells and the computer would cease to work.

In a computing mechanism, under normal conditions, digits of the same type affect primitive components of a mechanism in sufficiently similar ways that their dissimilarities make no difference to the resulting output. For instance, if two inputs to a NOT gate are sufficiently close to a certain voltage (labeled type '0'), the outputs from the gate in response to the two inputs must be of voltages different from the input voltages but sufficiently close to a certain other value (labeled type '1') that their difference does not affect further processing by other logic gates.

Furthermore, normally, digits of different types affect primitive components of a computing mechanism in sufficiently different ways that their similarities make no difference to the resulting outputs. That is not to say that for any two input types, a primitive component always generates outputs of different types. On the contrary, it is common for two computationally different inputs to give rise to the same computational output. For instance, in an AND gate, all of input types '0,0', '0,1', and '1,0' give rise to outputs of type '0'. But it is still crucial that the AND gate can give different responses to tokens of different types, so as to respond differently to '1,1' than to other input types. Thus, in all cases when two inputs of different types are supposed to generate different output types (such as the case of input type '1,1' in the case of an AND gate), the differences between digit types must suffice for the component to differentiate between them, so as to yield the correct outputs.

Which differences and similarities are relevant to a given mechanism depends on the technology used to build the mechanism. At different times, variants of mechanical, electro-mechanical, and electronic technologies have been used in computing applications. Newer technologies, such as optical, DNA, and quantum computing, are under development. It would be illuminating to study the details of different technologies, the specific similarities and differences between digits that are relevant to each, and the considerable engineering challenges that must be overcome to build mechanisms that reliably differentiate between different digit types. Since each technology poses specific challenges, however, no general treatment can be given.

For now, I hope the example of electronic logic gates is enough to grasp the basic idea. I will add some more pertinent observations below. Provided that the relations just discussed hold, a mechanism may be described as performing elementary (atomic) computations, because its inputs and outputs are digits, and the relation between inputs and outputs may be

characterized by a simple logical relation. But elementary computations are trivial. When we talk about computing, we are generally interested in computations over strings (of nontrivial length). For that, we need to introduce a concrete notion of string, which requires a concrete ordering of the digits.

3.4. Strings of Digits and Complex Computing Components. Any relation between digits that possesses the abstractly defined properties of concatenation may constitute a concrete counterpart to abstract concatenation. The simplest examples of ordering relations are spatial contiguity between digits, temporal succession between digits, or a combination of both.

For instance, suppose you have a literal physical implementation of a simple Turing machine. The tape has a privileged square, *s*. Before the machine begins, the input string is written on the tape. The digit written on *s* is the first digit in the string, the one on its right is the next, and so forth. When the machine halts, the output string is written on the tape, ordered in the same direction as the input. This is a spatial ordering of digits into a string.

An example of temporal concatenation is given by finite state automata. Since they have no tape, they simply take inputs one letter at a time. Any literal physical implementation of a finite state automaton will receive one letter at a time. The first digit to go in counts as the first in the string, the second as the second in the string, and so forth.

Real computers and other computing mechanisms may exploit a combination of these two strategies. Primitive components, such as logic gates, may be wired together to form *complex* components, which may in turn be wired together to form more complex components. This process must be iterated several times before one obtains an entire digital computer.

In designing computing mechanisms, not any wiring between components will do. The components must be arranged so that it is clear where the input digits come in and where the output digits come out. In addition, for the inputs and outputs to constitute strings, the components must be arranged so as to respect the desired relations between the digits composing the strings. What those relations are depends on which computation is performed by the mechanism.

For example, consider a circuit that adds two four digit strings.[11] A simple way to perform binary addition is the following: add each pair of bits; if there is a carry from the first two bits, add it to the second two

11. Addition is normally understood as an arithmetical operation, defined over numbers. In this case, it should be understood as a string-theoretic operation, defined over strings of numerals written in binary notation.

bits; after that, if there is a carry from the second two bits, add it to the third two bits; and so forth until the last two bits. A circuit that performs four bit addition in this way must be functionally organized so that the four digits in the input strings are manipulated in the way just specified, in the correct order. The first two bits may simply be added. If they generate a carry, that must be added to the second two bits, and so forth. The resulting wiring diagram will be asymmetric: different input digits will be fed to different components, whose exact wiring to other components depends on how their respective digits must be processed, and in what order. Implicit in the spatial, temporal, and functional relations between the components of the whole circuit as well as the way the circuit is connected to other circuits is the order defined on input and output digits.

An important aspect of digit ordering is synchrony between components. When a computing mechanism is sufficiently large and complex, there needs to be a way to ensure that all digits belonging to a string are processed during the same functionally relevant time interval. What constitutes a functionally relevant time interval depends on the technology used, but the general point is independent of technology. The components of a mechanism interact over time, and given their physical characteristics, there is only a limited amount of time during which their interaction can yield the desired result, consistent with the ordering of digits within strings.

Consider again our four bit adder. If the digits that are intended to be summed together enter the mechanism at times that are sufficiently far apart, they will not be added correctly, even if they are correctly received by the components that are supposed to process them. If a carry from the first two bits is added to the second two bits too late, it will fail to affect the result. And so on. Concrete computation has temporal aspects, which must be taken into account in designing and building computing mechanisms. When this is done correctly, it contributes to implementing the relation of concatenation between digits. When it is done incorrectly, it prevents the mechanism from working properly.

Unlike a simple four bit adder, which yields its entire output at once, there are computing components that generate different portions of their output at different times. When this is the case, the temporal succession between groups of output digits may constitute (an aspect of) the ordering of digits into strings.

Yet other functional relations may be used to implement concatenation. Within modern, stored program computers, computation results are stored in large memory components. Within such memories, the concatenation of digits into strings is realized neither purely spatially nor purely temporally. Rather, there is a system of memory registers, each of which has a label, called an 'address'. If a string is sufficiently long, a memory register

may contain only a portion of it. To keep track of a whole string, the computer stores the addresses where the string's parts are stored. The order of register names within the list corresponds to the relation of concatenation between the parts of the string that is stored in the named registers. By exploiting this mechanism, a computer can store very large strings and keep track of the digits' order without needing to possess memory components of corresponding length.

In short, just as an abstract algorithm is sensitive to the position of a letter within a string of letters, a concrete computing mechanism—via the functional relations between its components—is sensitive to the position of a digit within a string of digits. Thus, when an input string is processed by a mechanism, normally the digit types, their number, and their order within the string make a difference to what output string is generated.

3.5. Components, Functions, and Organization. As we have seen, digits and strings thereof are equivalence classes of physical entities or states. For instance, all voltages sufficiently close to two set values count as token digits of two different types; all voltages sufficiently far from those values do not count as digits at all. But voltage values could be grouped in many ways. Why is one grouping privileged within a computing mechanism? The answer has to do with the components of the mechanism, their functional properties, and their organization.

Some components of computing mechanisms do not manipulate digits. Their functions include storing energy (battery), keeping the temperature below a certain value (fan), or protecting the mechanism (case). They can be ignored here, because we are focusing on components that participate in computations. Components that manipulate digits are such that they stabilize only on states that count as digits. Finding components with such characteristics and refining them until they operate reliably is an important aspect of computer design. In ordinary computing technology, the components that manipulate digits can be classified as follows.

Input devices have the function of turning external stimuli into strings of digits. Memory components have the function of storing digits and signaling their state upon request. Their state constitutes either data strings or the physical implementation of abstract internal states. Processing components have the function of taking strings of digits as inputs and returning others as outputs according to a fixed rule defined over the strings. Output devices have the function of taking the final digits produced by the processing components and yielding an output to the environment. Finally, some components simply transmit digits between the other components.

Given their special functional characteristics, digits can be labeled by letters and strings of digits by strings of letters. As a consequence, the

same formal operations and rules that define abstract computations over strings of letters can be used to characterize concrete computations over strings of digits. Within a concrete computing mechanism, the components are connected so that the inputs from the environment, together with the digits currently stored in memory, are processed by the processing components in accordance with a set of instructions. During each time interval, the processing components transform the previous memory state (and possibly, external input) in a way that corresponds to the transformation of each snapshot into its successor. The received input and the initial memory state implement the initial string of an abstract computation. The intermediate memory states implement the intermediate strings. The output returned by the mechanism, together with the final memory state, implement the final string.

To show in detail how computing mechanisms can implement complex computations goes beyond the scope of this essay. I have room for only a few brief remarks. Abstract computations can be reduced to elementary operations over individual pairs of letters. Letters may be implemented as digits by the state of memory cells. Elementary operations on letters may be implemented as operations on digits performed by logic gates. Logic gates and memory cells can be wired together so as to correspond to the composition of elementary computational operations into more complex operations. Provided that (i) the components are wired so that there is a well-defined ordering of the digits being manipulated and (ii) the components are synchronized and functionally organized so that their processing respects the ordering of the digits, the behavior of the resulting mechanism can be accurately described as a sequence of snapshots. Hence, under normal conditions, such a mechanism processes its inputs and internal states in accordance with a program; the relation between the mechanism's inputs and its outputs is captured by a computational rule.

The synchronization provision is crucial and often underappreciated. Components must be synchronized so that they update their state or perform their operations within appropriate time intervals. Such synchronization is necessary to individuate digits appropriately, because synchronization screens off irrelevant values of the variables some values of which constitute digits. For example, when a memory component changes its state, it evolves through all values in between those that constitute well-defined digits. Before it stabilizes on a new value, it takes on values that constitute no digits at all. This has no effect on either the proper taxonomy of digits or the mechanism's computation. The reason is that memory state transitions occur during well-defined time intervals, during which memory components' states do not affect the rest of the mechanism. This contributes to making the equivalence classes that constitute digits functionally well defined. Thus, synchronization is a necessary aspect of

the computational organization of ordinary computing technology. Without synchronization, there would be no complex computations, because there would be no well-defined digits.

In short, any system whose correct mechanistic explanation ascribes to it the function of generating output strings from input strings (and possibly internal states), in accordance with a general rule that applies to all strings and depends on the input strings (and possibly internal states) for its application, is a computing mechanism. The mechanism's ability to perform computations is explained mechanistically in terms of its components, their functions, and their organization. By providing a mechanistic explanation of computing mechanisms, it is thus possible to individuate computing mechanisms, the functions they compute, and their computing power, and to explain how they perform their computations.

4. The Mechanistic Account and the Six Features. I will now argue that the mechanistic account possesses all the features listed in Section 1.

1. *Objectivity*. Given the mechanistic account, computational descriptions are neither vacuous nor trivial. The account relies on the analysis of mechanisms into their components, functions, and organization by the relevant community of scientists. As a result, whether a concrete system is a (nontrivial) computing mechanism and what it computes are matters of empirical fact.

Mechanistic descriptions are sometimes said to be perspectival, in the sense that the same component or activity may be seen as part of different mechanisms depending on which phenomenon is being explained (e.g., Craver 2001). For instance, the heart may be said to be for pumping blood as part of an explanation of blood circulation, or it may be said to be for generating rhythmic noises as part of an explanation of physicians who diagnose patients by listening to their hearts. This kind of perspectivalism does not trivialize mechanistic descriptions. Once we fix the phenomenon to be explained, the question of what explains the phenomenon has an objective answer. This applies to computations as well as other capacities of mechanisms. A heart makes the same noises regardless of whether a physician is interested in hearing it or anyone is interested in explaining medical diagnosis.

What we want to avoid is observers who share the same mechanistic perspective and yet ascribe different computations to the same system. Under the mechanistic account, this is not an option any more than it is an option for different observers to attribute different noises to the same heart. For example, either something is a memory register or not, an arithmetic-logic unit or not, etc., depending on what it contributes to its containing mechanism. It is certainly possible to label the digits processed by a computing mechanism using different letters. But these do not con-

stitute alternative computational descriptions of the mechanism; they are merely notational variants, all of which attribute the same computation to the mechanism. In short, the mechanistic description of a computing mechanism is no less objective than any other mechanistic description in biology or engineering. What is computed by which mechanism is a matter of fact.

2. *Explanation*. According to the mechanistic account, computational explanation is a form of mechanistic explanation. As a long philosophical tradition has recognized, mechanistic explanation is explanation in terms of a system's components, functions, and organization. Computational explanation is the form taken by mechanistic explanation when the activity of a mechanism can be accurately described as the processing of strings of digits in accordance with appropriate rules.

Traditionally, many philosophers assimilate computational explanation to explanation by program execution (Fodor 1968; Cummins 1977, 1983). The mechanistic account resists this assimilation. According to the mechanistic account, explanation by program execution is the special kind of mechanistic explanation that applies to all soft-programmable mechanisms—namely, mechanisms controlled by concrete instructions—regardless of whether such mechanisms perform computations. Program execution is a process by which a certain component or state of a mechanism, the concrete program, affects another component of the mechanism, a processing component, so as to perform different sequences of operations. But program execution need not be computation.

For instance, some automatic looms operate by executing programs, and yet they do not perform computations (in the sense relevant here). The difference between program-executing *computers* and other types of program-executing mechanisms is in the inputs they process and the way their processes are responsive to their inputs. Only the inputs (and memory states) of computers are genuine strings of digits, because only the processes executed by computers are defined over, and responsive to, both their finitely many input types and their order. Put another way, program-executing looms perform the same operations regardless of the properties of the inputs they process (unless, say, the inputs are such as to break the loom), and even regardless of whether they have any inputs to process.

Program execution is an interesting capacity of certain mechanisms, including computing mechanisms, and it is explained mechanistically. When combined with the capacity to perform computations, which is also explained mechanistically, program execution results in a powerful kind of computing mechanism, soft-programmable computers, whose computations are explained in part by program execution.

3. *The right things compute*. All paradigmatic examples of computing mechanisms, such as digital computers, calculators, Turing machines, and

finite state automata, have the function of generating certain output strings from certain input strings and internal states according to a general rule that applies to all strings and depends on the inputs and internal states for its application. According to the mechanistic account, then, they all perform computations. Thus, the mechanistic account properly counts all paradigmatic examples of computing mechanisms as such.

The mechanistic account also counts most connectionist systems as performing computations. Connectionist systems can be decomposed into units with functions and an organization, and hence they are mechanisms in the present sense. They take input strings of digits and return output strings of digits in accordance with an appropriate rule, and hence they are computing mechanisms. Unlike ordinary computing mechanisms, the units of connectionist systems need not be logic gates. Their capacities still have a mechanistic explanation, but such an explanation does not involve the decomposition of their computations into simpler computations performed by their components. Like logic gates, paradigmatic connectionist systems are computationally primitive.

The units of some connectionist systems have activation values that vary along a continuum, so such activation values may appear to be something different from digits. But in fact, in most of connectionist formalisms, these activation values are "read" as inputs to and outputs from the whole system only when they approximate certain standard values at functionally significant times. In this respect, they are not different from the activation values of the components of digital computers, which also vary along a continuum but are functionally significant only when they approximate certain standard values at functionally significant times. As a result, the functionally significant activation values of input and output units of connectionist systems constitute digits, and the activation values of whole input and output layers of units constitute strings of digits. An appropriate rule can then be used to characterize the relationship between the input and output strings. In fact, this is precisely how connectionist systems are described when theorists study the functions they compute (cf. Hopfield 1982; Rumelhart and MacClelland 1986; Minsky and Papert 1988; Siegelmann 1999; Piccinini 2007c).

In formulating the mechanistic account, I purposefully did not say whether the rule specifying the computation performed by a mechanism is recursive (or equivalently, computable by a Turing machine). This is because computability theorists define recursive as well as nonrecursive (abstract) computations. Both recursive and nonrecursive computations may be defined in terms of instructions for manipulating strings of letters or rules connecting input strings to output strings. Thus, both fall under the present account.

The only functions that are known to be physically computable are the

recursive ones. There is an ongoing controversy over the physical possibility of hypercomputers—mechanisms that compute nonrecursive functions (Copeland 2002; Cotogno 2003). I do not have room to address the hypercomputation controversy here. But that controversy should not be resolved by stipulating that hypercomputers do not perform computations. A good account of computing mechanisms should be able to accommodate hypercomputers. This highlights another advantage of the mechanistic account.

Traditional accounts are formulated in terms of either a canonical formalism, such as Turing machines (Putnam 1967), or the standard notion of computer program (Fodor 1975; Cummins 1983). Since standard computer programs and formalisms can compute only recursive functions, accounts based on them cannot accommodate hypercomputers. The mechanistic account, by contrast, is formulated in terms of rules that relate input strings of digits to output strings of digits. If those rules are recursive, we obtain the usual class of computing mechanisms. If those rules are not recursive, we obtain various classes of hypercomputers.

The mechanistic account, however, distinguishes between genuine and spurious hypercomputers. Genuine hypercomputers are mechanisms that have the function of generating output strings of digits from input strings of digits in accordance with a nonrecursive rule. Alan Turing's o-machines are an example (Turing 1939; cf. Copeland [2000] and Piccinini [2003] for discussion). Spurious hypercomputers are physical processes that are nonrecursive in some way, but do not have the function of generating strings of digits in accordance with a nonrecursive rule. Genuinely random processes are an example.

The distinction between genuine and spurious hypercomputers clarifies the debate over hypercomputation, where considerations pertaining to spurious hypercomputers are often mixed up with considerations pertaining to genuine hypercomputers. If one does not draw this distinction, it is relatively easy to show that 'hypercomputation' is possible. Any genuinely random process will do. But this is not an interesting result, for a random process cannot be used to generate the values of a desired function. The interest in hypercomputation is due to the hope for machines that yield strings of output digits that are related to their inputs in a nonrecursive way. For something to be a genuine hypercomputer, it must be possible to specify the rule relating the inputs to the outputs without waiting for the physical process to take place.

Given the generality of the mechanistic account, it may be surprising that it excludes so called analog computers (in the sense of Pour-el [1974]). Analog computers do not manipulate strings of digits. Rather, they manipulate real (i.e., continuous) variables. Hence, they are left out of the present account. But analog computers can be given their own mechanistic

account in terms of their components, functions, and organization (Piccinini nd). The fact that both classes of mechanisms are called computers should not blind us to their deep differences. After all, their bearing the same name is to some extent an historical accident. Before the invention of digital computers, analog computers used to be called 'differential analyzers'. They were later renamed 'analog computers' probably because for some time they shared important applications with digital computers and competed within the same market. Once again, thinking about computation in mechanistic terms allows us to appreciate and sharpen important distinctions. Digital computers and analog 'computers' operate on different vehicles by means of different processes. Determining their analogies and disanalogies is a nontrivial problem, which deserves more philosophical attention than it has received.

4. *The wrong things do not compute*. Let me grant from the outset that all systems may be given computational descriptions, which describe the behavior of the system to some degree of approximation. But giving a computational description of a system is not the same as asserting that the system itself performs computations.[12] The mechanistic account explains why paradigmatic examples of noncomputing systems do not compute by invoking their mechanistic explanation (or lack thereof), which is different from that of computing mechanisms. Different considerations apply to different classes of systems.

To begin with, most systems—including planetary systems and the weather—are not mechanisms in the present sense, because they are not collections of functional components organized to exhibit specific capacities.[13] Also, most systems—including, again, planetary systems and the weather—do not receive inputs from an external environment, process them, and return outputs distinct from themselves. It is not difficult to cook up notions of input that apply to all systems. For instance, sometimes initial conditions or time instants are described as inputs. But these are not entities or states that can enter the system, persist within the system, and finally exit the system. Hence, they do not count as computational inputs in the present sense.

In addition, most mechanisms—including mechanisms that manipulate inputs and return outputs distinct from themselves and their states—do

12. Noncomputing systems may be said to be "computational" in some looser senses than genuine computing mechanisms. I offered a taxonomy of those looser senses in Piccinini (2007b).

13. To be sure, there are accounts of function according to which planetary systems and the weather have functions. But no one disputes that they lack *teleological* functions. As I pointed out above, I am working with a teleological notion of function, while remaining neutral on the correct account of teleology.

not manipulate strings of digits. Digestive systems are a good example. As a preliminary observation, there is no prominent scientific theory according to which digestion is a computational process. In other words, the current science of digestion does not identify finitely many types of input digits and an ordering between them, let alone processes that manipulate those inputs in accordance with a rule defined over the input types and their place within strings. Instead, the science of digestion identifies food types first and foremost by the *family* of macromolecules to which its constituents belong (carbohydrates, fats, or proteins). Different families are processed differently. But what matters to digestion are not the details of how molecules of different chemical types, or belonging to different families, form pieces of food. On the contrary, digestion mixes and breaks down pieces of food by mechanical and chemical means, obliterating temporal, spatial, and chemical connections between molecules until the resulting products can be either absorbed by the body or discarded.

Now, suppose that someone wished to develop a computational explanation of digestion. She would have to find a plausible candidate for input and output strings. The most obvious place to start for the role of input seems to be bites of food, and the most obvious candidate for the role of output seems to be the nutrients absorbed by the body plus the feces. Finally, the most obvious candidate for a concatenation relation seems to be the temporal order of bites and digestive products. A first difficulty in formulating the theory is that the outputs of digestion are of a kind so different from the inputs that, unlike ordinary computational outputs, they cannot be fed back into the system for further computational processing. This is not a devastating objection, as computations may be defined so that outputs belong to a different alphabet than the inputs. Perhaps feces belong to a different alphabet than food.

A more serious difficulty is that the most important taxonomy of inputs for the science of digestion has little to do with food bites. Bites of food come in indefinitely many sizes, shapes, and compositions, but the processes that take place during digestion are not defined in terms of the size, shape, or composition of food bites. Furthermore, even if bites of food could somehow be classified into finitely many functionally relevant types, their temporal order would not constitute a string of digits. For digestion, unlike computation, is largely indifferent to the order in which organisms ingest their food bites.[14] On one hand, the products of digestion always come out in roughly the same order, regardless of how the inputs

14. There may be partial exceptions: for instance, ingesting a certain substance before or after another may facilitate or hinder digestion. These exceptions seem unlikely to warrant a computational explanation of digestion.

came in. On the other hand, the first operations typically performed by organisms on the food they ingest eliminate most differences between bites of food. Upon being ingested, food is chewed, mixed with saliva, swallowed, and mixed with digestive fluid. The result, far from being responsive to any obvious differences between bites of food or their order, is a relatively uniform bolus.

The purpose of these quick observations is not to prove that digestion is not computational. Ultimately, according to the present account, whether digestion is computational is an empirical question, to be answered by the science of digestion. What I have shown is that under the present account, treating the digestive system as a computing mechanism faces considerable challenges. The common intuition that digestion is not computational might be wrong, but it is prima facie plausible.

Finally, not all mechanisms that manipulate strings of symbols do so in accordance with a general rule that applies to all strings and depends on the input strings for its application. We have already mentioned genuine random processes. A genuine random 'number' (or more precisely, numeral) generator produces a string of digits, but it does not do so by computing, because there is no rule for specifying which digit it will produce at which time. Thus, a genuinely random 'number' generator does not count as a computing mechanism. (Of course, random strings of digits, whether or not they are genuinely random, may play important roles in a computational process.)

This does not decide all the cases. There is still a grey area at the boundary between mechanisms that compute and mechanisms that do not. Anything that takes two kinds of input and generates one output that stands in a definite logical relation to its inputs can be described as a logic gate. Since the computations performed by logic gates are trivial, the fact that many things are describable as logic gates does not trivialize the mechanistic account of computing mechanisms. But if this point could be generalized, and too many mechanisms could be described as performing nontrivial computations, and perhaps even as computing by executing programs, then the mechanistic account would risk being trivialized. This is a fair concern, and it can be addressed head on.

Primitive computing components, such as logic gates, can be wired together to form computing mechanisms, whose computations can be logically analyzed into the operations performed by their components. But not every collection of entities, even if they may be described as logic gates when they are taken in isolation, can be connected together to form a computing mechanism. For that to happen, each putative logic gate must take inputs and generate outputs of the same type, so that outputs from one gate can be transmitted as inputs to other gates. In addition, even having components of the right kind is not enough to build a complex

computing component. For the components must be appropriately organized. The different gates must be connected together appropriately, provided with a source of energy, and synchronized. To turn a collection of logic gates into a functioning computer takes an enormous amount of regimentation. The logic gates must be appropriately organized to constitute complex computing components, which in turn must be appropriately organized to constitute full blown computing mechanisms. Building genuine computers requires overcoming many technical challenges.

In conclusion, how many things taken in isolation constitute a logic gate, or other primitive computing components, is a matter that can be left vague. For primitive computing components in isolation perform computations that cannot be decomposed into simpler computations performed by their components. The mechanistic account has the most interesting things to say about mechanisms that are computationally decomposable. And unlike computers and other nontrivial computing mechanisms, most systems, including most mechanisms, are not computationally decomposable. Since the paradigmatic examples of noncomputing systems do not appear to be subject to the relevant kind of mechanistic explanation, the mechanistic account properly counts them as systems that do not compute in the relevant sense.

5. *Miscomputation*. The mechanistic account of computing mechanisms explains what it means for a computing mechanism to make a mistake. Miscomputations are a kind of malfunction, i.e., an event in which a functional system fails to fulfill its function. In the case of computing mechanisms, whose function is to compute, functional failure results in a computational mistake.

There are many kinds of miscomputation. The most obvious is hardware failure, i.e., failure of a hardware component to perform its function (as specified by the mechanistic explanation of the mechanism). Hardware failure may be due to the failure of a computing component, such as a logic gate, or of a noncomputing component, such as a battery. Another kind of miscomputation, which applies to artifacts, may be due to a mistake in computer design, so that the designed mechanism does not in fact compute the function it was intended to compute. Again, the design mistake may be due to a computing component that does not compute what it was intended to compute or to a noncomputing component that does not fulfill its function (e.g., a clock with too short a cycle time). Other kinds of miscomputation pertain to the intentions of the programmers and users of a machine, rather than its designers. Some may be due to a programming error, whereby instructions are either ungrammatical (and hence cannot be executed) or do not play their intended role within the program. Yet another kind may be due to the accumulation of round off errors in the finite precision arithmetic that computer pro-

cessors employ. Finally, miscomputations may be due to faulty interaction between hardware and software. A familiar example of this last type occurs when the execution of a program requires more memory than the computer has available. When no more memory is available, the computer "freezes" without being able to complete the computation.[15]

6. *Taxonomy*. The mechanistic account of computing mechanisms explains why only some computing mechanisms qualify as computers properly so called: only genuine computers are programmable, stored program, and computationally universal. These properties of computers are functional properties, which can be explained mechanistically in terms of the relevant components, their functions, and their organization (Piccinini nd). Computing mechanisms that have only a subset of these capacities may also be called computers, but they can be distinguished from ordinary computers, and from one another, based on their specific functional properties. Computing mechanisms that lack all of these capacities deserve other names, such as "calculators," "arithmetic-logic units," etc.; they can still be differentiated from one another based on their computing power, which is determined by their functional organization.

5. Conclusion. The mechanistic account of computing mechanisms is a viable account of what it means for a physical system to compute. It has many appealing features. It allows us to formulate the question whether a mechanism computes as an empirical question, to be answered by a correct mechanistic explanation. It allows us to formulate a clear and useful taxonomy of computing mechanisms and compare their computing power. I submit that the mechanistic account of computing mechanisms constitutes an improvement over existing accounts of computation.

REFERENCES

Allen, C., M. Bekoff, and G. Lauder (eds.) (1998), *Nature's Purposes: Analysis of Function and Design in Biology*. Cambridge, MA: MIT Press.

Ariew, A., R. Cummins, and M. Perlman (eds.) (2002), *Functions: New Essays in the Philosophy of Psychology and Biology*. Oxford: Oxford University Press.

Bechtel, W., and R. C. Richardson (1993), *Discovering Complexity: Decomposition and Localization as Scientific Research Strategies*. Princeton, NJ: Princeton University Press.

Boorse, C. (2002), "A Rebuttal on Functions", in A. Ariew, R. Cummins, and M. Perlman (eds.), *Functions: New Essays in the Philosophy of Psychology and Biology*. Oxford: Oxford University Press, 63–112.

Buller, D. J. (ed.) (1999), *Function, Selection, and Design*. Albany: State University of New York Press.

15. For an early discussion of several kinds of computing mistakes by computing mechanisms, see Goldstine and von Neumann (1946). For a modern treatment, see Patterson and Hennessy (1998).

Chalmers, D. J. (1996), "Does a Rock Implement Every Finite-State Automaton?", *Synthese* 108: 310–333.
Chrisley, R. L. (1995), "Why Everything Doesn't Realize Every Computation", *Minds and Machines* 4: 403–430.
Christensen, W. D., and M. H. Bickhard (2002), "The Process Dynamics of Normative Function", *Monist* 85: 3–28.
Church, A. (1940), "On the Concept of a Random Sequence", *Bulletin of the American Mathematical Society* 46: 130–135.
Collins, J., N. Hall, and L. A. Paul (eds.) (2004), *Causation and Counterfactuals*. Cambridge, MA: MIT Press.
Copeland, B. J. (1996), "What Is Computation?", *Synthese* 108: 224–359.
——— (2000), "Narrow versus Wide Mechanism: Including a Re-examination of Turing's Views on the Mind-Machine Issue", *Journal of Philosophy* 96: 5–32.
——— (2002), "Hypercomputation", *Minds and Machines* 12: 461–502.
Corcoran, J., W. Frank, and M. Maloney (1974), "String Theory", *Journal of Symbolic Logic* 39: 625–637.
Cotogno, P. (2003), "Hypercomputation and the Physical Church-Turing Thesis", *British Journal for the Philosophy of Science* 54: 181–223.
Craver, C. (2001), "Role Functions, Mechanisms, and Hierarchy", *Philosophy of Science* 68: 53–74.
——— (forthcoming), *Explaining the Brain*. Oxford: Oxford University Press.
Cummins, R. (1977), "Programs in the Explanation of Behavior", *Philosophy of Science* 44: 269–287.
——— (1983), *The Nature of Psychological Explanation*. Cambridge, MA: MIT Press.
Davis, M., R. Sigal, and E. J. Weyuker (1994), *Computability, Complexity, and Languages*. Boston: Academic Press.
Dretske, F. (1986), "Misrepresentation", in R. Bogdan (ed.), *Belief: Form, Content and Function*. New York: Oxford University Press, 17–36.
Dreyfus, H. L. (1979), *What Computers Can't Do*. New York: Harper & Row.
Fodor, J. A. (1968), "The Appeal to Tacit Knowledge in Psychological Explanation", *Journal of Philosophy* 65: 627–640.
——— (1975), *The Language of Thought*. Cambridge, MA: Harvard University Press
——— (1998), *Concepts*. Oxford: Clarendon Press.
Glennan, S. (2002), "Rethinking Mechanistic Explanation", *Philosophy of Science* 64: S342–S353.
Goldstine, H., and J. von Neumann (1946), "On the Principles of Large Scale Computing Machines", Princeton, NJ: Institute for Advanced Studies.
Hopfield, J. (1982), "Neural Networks and Physical Systems with Emergent Collective Computational Abilities", *Proceedings of the National Academy of Sciences* 79: 2554–2558.
Lewis, D. (1986), "Postscript to 'Causation'", in *Philosophical Papers*, vol. 2. New York: Oxford University Press, 172–213.
Machamer, P. K., L. Darden, and C. Craver (2000), "Thinking about Mechanisms", *Philosophy of Science* 67: 1–25.
Minsky, M. L., and S. A. Papert (1988), *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA: MIT Press.
Patterson, D. A., and J. L. Hennessy (1998), *Computer Organization and Design: The Hardware/Software Interface*. San Francisco: Morgan Kauffman.
Piccinini, G. (2003), "Alan Turing and the Mathematical Objection", *Minds and Machines* 13: 23–48.
——— (2004a), "Functionalism, Computationalism, and Mental Contents", *Canadian Journal of Philosophy* 34: 375–410.
——— (2004b), "Functionalism, Computationalism, and Mental States", *Studies in the History and Philosophy of Science* 35: 811–833.
——— (2007a), "Computation without Representation", forthcoming in *Philosophical Studies*.
——— (2007b), "Computational Modeling vs. Computational Explanation: Is Everything

a Turing Machine, and Does It Matter to the Philosophy of Mind?", *Australasian Journal of Philosophy* 85: 93–115.
——— (2007c), "Connectionist Computation", forthcoming in *Proceedings of the 2007 International Joint Conference on Neural Networks*.
——— (nd), "Computers". St. Louis: University of Missouri.
Pour-El, M. B. (1974), "Abstract Computability and Its Relation to the General Purpose Analog Computer (Some Connections between Logic, Differential Equations and Analog Computers)", *Transactions of the American Mathematical Society* 199: 1–28.
Preston, B. (1998), "Why Is a Wing like a Spoon? A Pluralist Theory of Function", *Journal of Philosophy* 95: 215–254.
Putnam, H. (1960), "Minds and Machines", in S. Hook (ed.), *Dimensions of Mind: A Symposium*. New York: Collier, 138–164.
——— (1967), "Psychological Predicates", in *Art, Philosophy, and Religion*. Pittsburgh: University of Pittsburgh Press.
——— (1988), *Representation and Reality*. Cambridge, MA: MIT Press.
Pylyshyn, Z. W. (1984), *Computation and Cognition*. Cambridge, MA: MIT Press.
Rumelhart, D. E., and J. M. McClelland (1986), *Parallel Distributed Processing*. Cambridge, MA: MIT Press.
Scheutz, M. (1999), "When Physical Systems Realize Functions . . .", *Minds and Machines* 9: 161–196.
Schlosser, G. (1998), "Self-Re-production and Functionality: A Systems-Theoretical Approach to Teleological Explanation", *Synthese* 116: 303–354.
Searle, J. R. (1980), "Minds, Brains, and Programs", *Behavioral and Brain Sciences* 3: 417–457.
——— (1992), *The Rediscovery of the Mind*. Cambridge, MA: MIT Press.
Shagrir, O. (2006), "Why We View the Brain as a Computer", *Synthese* 153: 393–416.
Sieg, W., and J. Byrnes (1996), "K-Graph Machines: Generalizing Turing's Machines and Arguments", in P. Hájek (ed.), *Gödel '96*. Berlin: Springer-Verlag, 98–119.
Siegelmann, H. T. (1999), *Neural Networks and Analog Computation: Beyond the Turing Limit*. Boston: Birkhäuser.
Tabery, J. (2004), "Synthesizing Activities and Interactions in the Concept of a Mechanism", *Philosophy of Science* 71: 1–15.
Turing, A. M. (1936–37 [1965]), "On Computable Numbers, with an Application to the Entscheidungsproblem", in M. Davis (ed.), *The Undecidable*. Ewlett, NY: Raven Press, 116–154.
——— (1939), "Systems of Logic Based on Ordinals", *Proceedings of the London Mathematical Society*, series 2, 45: 161–228.
Wimsatt, W. C. (2002), "Functional Organization, Analogy, and Inference", in A. Ariew, R. Cummins, and M. Perlman (eds.), *Functions: New Essays in the Philosophy of Psychology and Biology*. Oxford: Oxford University Press, 173–221.