
	P	A	P	Q	3	0	9	<b>Operator:</b> Xiaohua Zhou		<b>Dispatch:</b> 21.12.07	<b>PE:</b> Roy See
	Journal Name			Manuscript No.			<b>Proofreader:</b> Wu Xiuhua	<b>No. of Pages:</b> 42		<b>Copy-editor:</b>	

---

# COMPUTERS

BY

GUALTIERO PICCININI

---

**Abstract:** I offer an explication of the notion of the computer, grounded in the practices of computability theorists and computer scientists. I begin by explaining what distinguishes computers from calculators. Then, I offer a systematic taxonomy of kinds of computer, including hard-wired versus programmable, general-purpose versus special-purpose, analog versus digital, and serial versus parallel, giving explicit criteria for each kind. My account is mechanistic: which class a system belongs in, and which functions are computable by which system, depends on the system's mechanistic properties. Finally, I briefly illustrate how my account sheds light on some issues in the history and philosophy of computing as well as the philosophy of mind.

What exactly is a digital computer? (Searle, 1992, p. 205)

In our everyday life, we distinguish between things that compute, such as pocket calculators, and things that don't, such as bicycles. We also distinguish between different *kinds* of computing device. Some devices, such as abaci, have parts that need to be moved by hand. They may be called *computing aids*. Other devices contain internal mechanisms that, once started, produce a result without further external intervention. They may be called *computing mechanisms*. Among computing mechanisms, we single out a special class and call them *computers*. At the very least, computers are special because they are more versatile than other computing mechanisms – or any other mechanisms, for that matter. Computers can do arithmetic but also graphics, word processing, Internet browsing, and myriad other things. No other artifact comes even close to having so many capacities. Computer versatility calls for an explanation, which I will sketch in due course.

In contrast with the above intuitive picture, some authors suggest that there is nothing distinctive about computers. Consider the following passage:

[T]here is no intrinsic property necessary and sufficient for all computers, just the interest-relative property that someone sees value in interpreting a system's states as representing

*Pacific Philosophical Quarterly* 89 (2008) 32–73

© 2008 The Author

Journal compilation © 2008 University of Southern California and Blackwell Publishing Ltd.

1 states of some other system, and the properties of the system support such an interpretation. . . .  
2 Conceivably, sieves and threshing machines could be construed as computers if anyone has  
3 reason to care about the specific function reflected in their input-output behavior. (Church-  
4 land and Sejnowski, 1992, p. 65–66)  
5

6 If these authors are correct, then the distinctions between (i) mechanisms  
7 that compute and mechanisms that don't and (ii) computers and other  
8 computing mechanisms, are ill conceived. For according to these authors,  
9 there is no fact of the matter whether something is a computer, some  
10 other computing mechanism, or something that performs no computations  
11 at all. Whether anything is said to be a computer, for these authors, depends  
12 solely on how we look at it.

13 In the absence of a viable account of computers, this *computational*  
14 *nihilism* is tempting. But computational nihilists have a lot to explain  
15 away. They should explain away our intuition that there is something  
16 distinctive about being a computer and our consequent practice of applying  
17 the term 'computer' only to some mechanisms – such as our desktop and  
18 laptop machines – and not others. They should explain why we think that  
19 the invention of computers was a major intellectual breakthrough and  
20 there are special sciences – computer science and computer engineering –  
21 devoted to studying the specific properties of computers. Finally, they  
22 should explain why the systems we ordinarily call computers, but not  
23 other computing mechanisms that pre-date them, inspired the hypothesis  
24 that minds or brains are computers (von Neumann, 1958; Fodor, 1975).

25 Explaining all of this away, I submit, is hopeless. For there is a fact of  
26 the matter: a computer is *a calculator of "large capacity,"* and a calculator  
27 is *a mechanism whose function is to perform a few computational operations*  
28 *on inputs of bounded but nontrivial size.*<sup>1</sup> Most of this paper may be seen  
29 as a detailed articulation and defense of these theses, including a specification  
30 of what I mean by 'large capacity'. While doing so, I will develop a  
31 systematic taxonomy of kinds of computer, including hard-wired versus  
32 programmable, general-purpose versus special-purpose, analog versus  
33 digital, and serial versus parallel, giving explicit criteria for each kind.

34 The topic of this paper is of interest for at least three reasons. First,  
35 computers are sufficiently important, both practically and conceptually,  
36 that understanding what they are is valuable in its own right. Second, a  
37 proper distinction between computers and other mechanisms and  
38 between different classes of computers is part of the foundations of  
39 computer science. In computer science there is no universally accepted  
40 notion of computer, and there have been controversies over whether  
41 certain machines count as computers of one kind or another. Some of  
42 those controversies have significant legal consequences (cf. Burks, 2002).  
43 Resolving those controversies requires clear and cogent criteria for what  
44 counts as a computer of any significant kind. This paper provides such

1 criteria. In Section 4, I will give an example of how my account illuminates  
2 a historical controversy. Third, as I will illustrate in Sections 5 and 6, a  
3 robust notion of computer gives substance to theories according to which  
4 the brain is a computer.

### 7 **1. *Computing mechanisms***<sup>2</sup>

8  
9 Like other mechanisms, computing mechanisms' capacities are explained  
10 in terms of their components, their components' functions, and the way  
11 their components and functions are organized. Computing mechanisms  
12 are distinguished from other (non-computing) mechanisms in that their  
13 capacities are explained in terms of specific processes: computations. A  
14 *computation*, in turn, is the generation of output strings of digits from  
15 input strings of digits in accordance with a general rule that depends  
16 on the properties of the strings and (possibly) on the internal state of  
17 the system. Finally, a *string of digits* is an ordered sequence of discrete  
18 elements of finitely many types, where each type is individuated by the  
19 different effects it has on the mechanism that manipulates the strings.  
20 Under this account, strings of digits are entities that are individuated in  
21 terms of their functional properties within a mechanistic explanation of  
22 a system.

23 Although strings of digits can be and usually are assigned semantic  
24 interpretations, the present notion of string is the one employed in  
25 computability theory and computer science, where semantic properties  
26 are not part of the identity conditions of strings (Piccinini forthcoming  
27 b). This is a functional, non-semantic notion of string. It is explicated in  
28 terms of how each type of digit, together with its position within a string,  
29 affects the computing components of a mechanism.<sup>3</sup> Thus, this notion of  
30 string does not depend on an appeal to semantic properties.

31 In analyzing computers, I will distinguish between three classes of  
32 strings on the basis of the function they fulfill: (1) *data*, whose function is  
33 to be manipulated during a computation; (2) *results*, whose function is to  
34 be the final string of a computation; and (3) *instructions*, whose function  
35 is to cause appropriate computing mechanisms to perform specific operations  
36 on the data. Lists of instructions are called *programs*. I will also appeal to  
37 four kinds of large-scale components: (1) *processing units*, whose function  
38 is to execute any of a finite number of primitive operations on data; (2)  
39 *memory units*, whose function is to store data, intermediate results, final  
40 results, and possibly instructions; (3) *input devices*, whose function is to  
41 receive strings of digits from the environment and deliver them to memory  
42 units, and (4) *output devices*, whose function is to take strings of digits  
43 from memory units and deliver them to the external environment. Some  
44 processing units can be further analyzed into datapaths, whose function is

1 to perform operations on data, and control units, whose function is to set  
2 up datapaths to perform the operation specified by any given instruction.  
3  
4

## 5 2. Calculators

6  
7 Sometimes, the terms ‘calculator’ and ‘computer’ are used interchangeably.  
8 Following the more common practice, I reserve ‘computer’ to cover a  
9 special class of calculators. Given this restricted usage, a good way to  
10 introduce computers is to contrast them with their close relatives, ordinary  
11 calculators.

12 A calculator is a computing mechanism made out of four kinds of  
13 (appropriately connected) components: input devices, output devices,  
14 memory units, and processing units. In this section, I only discuss non-  
15 programmable calculators. So-called “programmable calculators,” from a  
16 mechanistic perspective, are special purpose computers with small memory,  
17 and are subsumed within the next section.<sup>4</sup>

18 Input devices of calculators receive two kinds of input from the environ-  
19 nment: (i) data and (ii) a command that causes the processing unit to  
20 perform a certain operation on the data. Commands are usually inserted  
21 in calculators by pressing appropriate buttons. An operation is a transfor-  
22 mation of data into results. Calculators’ memory units hold the data,  
23 and possibly intermediate results, until the operation is performed.  
24 Calculators’ processing units perform one operation on the data. Which  
25 operation is performed depends only on which command was inserted  
26 through the input device. After the operation is performed, the output  
27 devices of a calculator return the results to the environment. The results  
28 returned by calculators are computable (recursive) functions of the data.  
29 Since calculators manipulate strings of digits according to a general rule,  
30 they – unlike the sieves and threshing machines mentioned by Churchland  
31 and Sejnowski – are genuine computing mechanisms. The computing power  
32 of ordinary calculators is limited, however, by three important factors.

33 First, (*ceteris paribus*) an ordinary calculator’s result is the value of one  
34 of a fixed finite number of functions of the data, and the set of those  
35 functions cannot be augmented (e.g. by adding new instructions or programs  
36 to the calculator). The set of functions that can be computed by a calculator  
37 is determined by the primitive operations on strings that can be performed  
38 by the processing unit. Which of those functions is computed at any given  
39 time is determined by the command that is inserted through the input  
40 device. The command sets the calculator on one of a finite number of initial  
41 states, which correspond to the functions the calculator can compute. In  
42 short, calculators (in the present sense) are not programmable.

43 Second, an ordinary calculator performs only one operation on its  
44 data, after which it outputs the result and stops. It has no provision for

1 performing several of its primitive operations in a specified order, so as to  
2 follow an algorithm automatically. Of course, each primitive operation of  
3 a calculator is performed by following a pseudo-algorithm (that is, an  
4 algorithm defined over finitely many inputs). But a calculator – unlike a  
5 computer – cannot follow an algorithm or pseudo-algorithm defined in  
6 terms of its primitive operations. In other words, a calculator has no control  
7 structure besides the insertion of commands through the input device.<sup>5</sup>  
8 The operations that a calculator can compute on its data can be combined  
9 in sequences, but only by inserting successive commands through the  
10 input device after each operation is performed.

11 Finally, the range of values that calculators can compute is limited  
12 by the size of their memory, input devices, and output devices. Ordinary  
13 calculator memories are of fixed size and cannot be increased in a modular  
14 fashion. Also, ordinary calculator input and output devices only take  
15 data and deliver results of bounded size. Calculators can only operate  
16 within the size of those data and results, and cannot outrun their limited  
17 memory capacity.

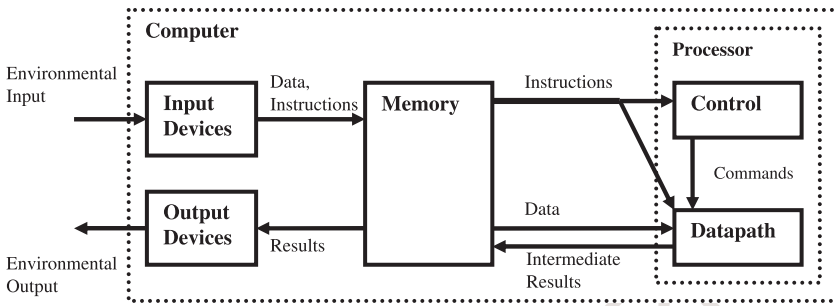
18 As a consequence of these limitations, ordinary calculators lack  
19 computers' most interesting functional properties: calculators have no  
20 "virtual memory" and do not support "complex notations" and "complex  
21 operations." In short, they have no "functional hierarchy." (These terms  
22 are explained in the next section.)

23 Calculators are computationally more powerful than simpler computing  
24 mechanisms, such as logic gates or arithmetic-logic units. Nevertheless,  
25 the computing power of ordinary calculators is limited in a number of  
26 ways. Contrasting these limitations with the power of computers sheds  
27 light on why computers are so special and why computers, rather than  
28 calculators, have been used as a model for the brain.

### 3. *Computers*

33 Like ordinary calculators, computers are made out of four types of  
34 components: input devices, output devices, memory units, and processing  
35 units. The processing units of modern computers are called *processors*  
36 and can be analyzed as a combination of datapaths and control units. A  
37 schematic representation of the functional organization of a modern  
38 computer is shown in Figure 1.

39 The difference between (ordinary) calculators and computers lies in  
40 their mechanistic properties and organization. Computers' processors are  
41 capable of branching behavior and can be set up to perform any number  
42 of their primitive operations in any order (until they run out of memory  
43 or time). Computers' memory units are orders of magnitude larger than  
44 those of ordinary calculators, and often they can be increased in a modular



**Figure 1** The main components of a computer and their functional relations.

fashion if more storage space is required. This allows today's computers to take in data and programs, and yield results, of a size that has no well-defined upper bound. So computers, unlike calculators, are programmable, capable of branching, and capable of taking data and yielding results of "unbounded" size. Because of these characteristics, today's computers are called "programmable", "stored-program", and computationally "universal". (These terms are defined more explicitly below.)

If we were to define 'computer' as something with all the characteristics of today's computers, we would certainly obtain a robust notion. We would also make it difficult to distinguish between many classes of computing mechanisms that lack some of those characteristics, yet are significantly more powerful than ordinary calculators, are similar to modern computers in important ways, and were often called "computers" when they were built. Because of this, I recommend using the term 'computer' so as to encompass more than today's computers, while introducing functional distinctions among different classes of computers. Ultimately, our understanding does not depend on how restrictively we use a term; it depends on how careful and precise we are in classifying computing mechanisms based on their relevant functional properties.

Until at least the 1940s, the term 'computer' was used to designate people whose job was to perform calculations, usually with the help of a calculator or abacus. Unlike the calculators they used, these computing humans could string together a number of primitive operations (each of which was performed by the calculator) in accordance with a fixed plan, or algorithm, so as to solve complicated problems defined over strings of digits. By analogy, any machine with an analogous capacity may also be called a 'computer'.

To a first approximation, then, a computer is a computing mechanism with a control unit that can string together a sequence of primitive operations, each of which can be performed by the processing unit(s), so as to follow an algorithm or pseudo-algorithm (i.e. an algorithm defined

1 over finitely many inputs). Among computers, there is wide variation in  
2 how many operations their control unit can string together in a sequence  
3 and how complex an algorithm their control unit can follow (and consequently,  
4 how complex a problem a computer can solve). For instance,  
5 some machines that were built in the first half of the 20<sup>th</sup> century – such  
6 as the IBM 601 – could string together a handful of arithmetical operations.  
7 They were barely more powerful than ordinary calculators, and a  
8 computing human could easily do anything that they did. Other machines  
9 – such as the Atanasoff-Berry Computer (ABC) – could perform long  
10 sequences of operations on their data; a computing human could not  
11 solve the problems that they solved without taking a prohibitively long  
12 amount of time.

13 The ABC, which was completed in 1942 and was designed to solve  
14 systems of up to 29 linear algebraic equations in 29 unknowns, appears to  
15 be the first machine that was called ‘computer’ by its inventor.<sup>6</sup> So, a  
16 good place to draw the boundary between calculators and computers  
17 might lie somewhere between machines that can be replaced by computing  
18 humans, such as ordinary calculators and the IBM 601, and machines  
19 that outperform computing humans at solving at least some problems,  
20 such as the ABC.<sup>7</sup> The exact boundary is best left vague. What matters is  
21 not how many machines we honor with the term ‘computer,’ but that we  
22 identify mechanistic properties that make a difference in computing  
23 power, and that whether a machine possesses any of these properties is a  
24 matter of fact, not interpretation. We’ve seen that one of those properties  
25 is the capacity to follow an algorithm defined in terms of the primitive  
26 operations that a machine’s processing unit(s) can perform. Other  
27 important mechanistic properties of computers are discussed in the rest  
28 of this section.

29 Computers and their components can also be classified according to  
30 the technology they use (mechanical, electro-mechanical, electronic, etc.)  
31 as well as according to other characteristics (size, speed, cost, etc.). These  
32 differences don’t matter for our purposes, because they don’t affect which  
33 functions can in principle be computed by different classes of computing  
34 mechanisms. Historically, however, the introduction of electronic technology,  
35 and the consequent enormous increase in computation speed, made a  
36 huge difference in making computers practically useful.

### 37 38 3.1 PROGRAMMABILITY

39  
40 A computing human can do more than follow *one* algorithm to solve a  
41 problem. She can follow *any* algorithm, which is typically given to her in  
42 the form of instructions, and thus she can compute any function for  
43 which she has an algorithm. More generally, a human can be instructed  
44 to perform the same activity (e.g. knitting or playing the piano) in many

1 different ways. Any machine that can be easily modified to yield different  
2 output patterns may be called 'programmable'. In other words, 'being  
3 programmable' means being modifiable so as to perform relatively long  
4 sequences of different operations in a different way depending on the  
5 modification.

6 Computers are not the only programmable mechanisms. This is  
7 especially relevant in light of the persistent tendency by some philosophers  
8 to identify computation with program execution (e.g. Cummins, 1989, pp.  
9 91–92; Roth, 2005, p. 458). On the contrary, some types of non-computing  
10 mechanisms, such as certain looms and music boxes, execute programs  
11 too. Computers' characteristic activity is computing, so a programmable  
12 computer is a computer that can be modified to compute in different  
13 ways. Furthermore, as will soon be clear, the kind of programmability  
14 that comes with the ability to execute computer programs is only one species  
15 of computer programmability among others, just as programmable  
16 computation is only one species of computation among others.

17 The simplest kind of programmability involves the performance of  
18 specified sequences of operations on an input, without the characteristics  
19 of the input making any difference on what operations must be performed.  
20 For instance, (*ceteris paribus*) a loom programmed to weave a certain  
21 pattern will weave that pattern regardless of what kinds of thread it is  
22 weaving. The properties of the threads make no difference to the pattern  
23 being weaved. In other words, the weaving process is insensitive to the  
24 properties of the input. (Though of course, whether the pattern is easy to  
25 observe and how it appears will depend on whether relevant threads have  
26 different colors, and what colors they have.) This kind of programmability  
27 is insufficient for computing, because computing is an activity that (*ceteris*  
28 *paribus*) is sensitive to relevant differences in input. One reason for this is  
29 that a mathematical function typically yields different values given different  
30 arguments. Any mechanism that is computing that function must respond  
31 differentially to the different input arguments so as to generate the cor-  
32 rect output values. The way to do this is for computing mechanisms  
33 to respond differentially to the different types of digit that make up the  
34 input and to the order in which they are concatenated into strings.  
35 Accordingly, programming a computer requires specifying how it must  
36 respond to different strings of digits by specifying how it must respond to  
37 different types of digit and different positions within a string.<sup>8</sup>

38 What counts as a legitimate modification for the purpose of programming  
39 a computer depends on the context of use and the means that are available  
40 to make the modification. In principle, one could modify a calculator by  
41 taking it apart and rewiring its circuits or adding new parts, and then it  
42 would compute new functions. But this kind of modification would ordin-  
43 arily be described as building a new machine rather than programming  
44 the old one. So we should relativize the notion of programmability to the

1 way a machine is ordinarily used. This will not make a difference in the  
2 end, especially since the kind of programmability that matters for most  
3 purposes is soft-programmability (see below), where what counts as a  
4 relevant modification is determined unambiguously by the functional  
5 organization of the computer.

6 Programmability comes in two main forms, hard and soft, depending  
7 on the type of modification that needs to be made for the machine to  
8 behave in a different way.

### 9 10 3.1.1 *Hard programmability*

11 Some early computers – including the celebrated ENIAC – had switches,  
12 plug-ins, and wires with plugs, which sent signals to and from their  
13 computing components.<sup>9</sup> Flipping the switches or plugging the wires in  
14 different configurations had the effect of connecting the computing  
15 components of the machine in different ways, to the effect that the  
16 machine would perform different series of operations. I call any computer  
17 whose modification involves the mechanical modification of its functional  
18 organization *hard-programmable*. Hard programming requires that users  
19 change the way the components of a computer are spatially joined  
20 together, which changes the number of operations that are performed or  
21 the order in which they are performed, so that different functions are  
22 computed as a result. Hard programming is relatively slow and cumbersome.  
23 Programming is easier and faster if a machine is soft-programmable.

### 24 25 3.1.2 *Soft programmability*

26 Modern computers contain computing components that are designed to  
27 respond differentially to different sequences of digits, so that different  
28 operations are performed. These sequences of digits, then, act as instructions  
29 to the computer, and lists of instructions are called *programs*. In order to  
30 program these machines, it is enough to supply the appropriate arrange-  
31 ment of digits, without manually rewiring any of the components. I call  
32 any computer whose modification involves the supply of appropriately  
33 arranged digits (instructions) to the relevant components of the machine  
34 *soft-programmable*.

35 Soft programmability allows the use of programs of unbounded size,  
36 which are given to the machine as part of its input. Since the whole  
37 program is written down as a list of instructions, in principle a soft-  
38 programmable machine is not bound to execute the instructions in the  
39 order in which they are written down in the program; it can execute  
40 arbitrary instructions in the list at any particular time. This introduces  
41 the possibility of conditional branch instructions, which require the  
42 machine to jump from one instruction to an arbitrary instruction in the  
43 program based on whether a certain condition – which can be checked by  
44 the processing unit(s) – obtains. Conditional branch instructions are the

1 most useful and versatile way of using intermediate results of a computation  
2 to influence control, which in turn is a necessary condition for computing  
3 all computable functions.<sup>10</sup> Since soft programmable computers can execute  
4 programs of unbounded size and use intermediate results to influence  
5 control, they can be computationally universal (see below).

6 There are two kinds of soft programmability: external and internal. A  
7 machine that is *externally* soft-programmable requires the programs to  
8 be inserted into the machine through an input device, and it does not  
9 have components that copy and store the program inside the machine.  
10 For example, universal Turing machines and some early punched  
11 cards computers – such as the famous Harvard Mark I – are externally  
12 soft-programmable.<sup>11</sup>

13 A machine that is *internally* soft-programmable contains components  
14 whose function is to copy and store programs inside the machine and to  
15 supply instructions to the machine's processing units. Internal soft  
16 programmability is by far the most flexible and efficient form of pro-  
17 grammability. It even allows the computer to modify its own instructions  
18 based on its own processes, which introduces a final and most sophisticated  
19 level of computational control.<sup>12</sup>

20 Given the ability to modify their instructions, internally soft-programmable  
21 computers have a special, seldom-appreciated property. In principle, a  
22 soft-programmable computer may be able to increase its computational  
23 power by operations that are not computable by a Turing machine.<sup>13</sup> If an  
24 internally soft-programmable computer has the ability to modify its  
25 program(s) in a non-computable way (e.g. at random), then in principle it  
26 may modify its program so as to compute an increasingly larger number  
27 of functions. This increase in computational power need not be governed  
28 by any algorithm.

29 Internal soft-programmability has become so standard that other forms  
30 of programmability are all but forgotten or ignored. One common name  
31 for machines that are internally soft-programmable, which include all of  
32 today's desktop and laptop computers, is 'stored-program computer'. For  
33 present purposes, however, it is better to use the term 'stored-program' to  
34 refer to the presence of programs inside a machine, whether or not those  
35 programs can change.

### 37 3.2 STORED-PROGRAM COMPUTERS

38  
39 A stored-program computer has an internal memory that can store  
40 strings of digits. Programs are stored in memory as lists of instructions  
41 placed in appropriate memory registers. A stored-program computer also  
42 contains at least one processor. The processor contains a tracking  
43 mechanism (program counter) that allows the processor to retrieve  
44 instructions from a program in the appropriate order. The processor can

1 extract strings of digits from memory and copy them into its internal  
2 registers (if they are data) or execute them on the data (if they are instruc-  
3 tions). The same string can act as data or as an instruction in different  
4 occasions. After an instruction is executed, the tracking mechanism  
5 allows the control unit to retrieve the next instruction until all the relevant  
6 instructions are executed and the computation is completed.

7 A stored-program machine need not be programmable. Some kinds of  
8 memory unit are read-only, namely, they can communicate their content  
9 to other components but their content cannot change. Other kinds of  
10 memory unit are such that digits can be inserted into them. This opens up  
11 the possibility that a program be inserted from the outside of the computer  
12 (or from the inside, i.e. from the processor) into a memory unit, allowing  
13 a computer to be (soft) programmed and the existing programs to be  
14 modified. Once the program is in the appropriate part of the memory, the  
15 computer will execute that program on the data. A programmable stored-  
16 program computer can even be set up to modify its own program, changing  
17 what it computes over time. Stored-program computers are usually  
18 designed to be soft-programmable and to execute any list of instructions,  
19 including branch instructions, up to their memory limitations; if so, they  
20 are computationally universal (see below). For this reason, the term 'stored-  
21 program computer' is often used as a synonym of 'universal computer'.

22 The idea to encode instructions as sequences of digits in the same form  
23 as the data and the idea of storing instructions inside the computer are  
24 the core of the notion of programmable, stored-program computer, which  
25 is perhaps the most fundamental aspect of modern mechanical computing.<sup>14</sup>  
26 At this point, it should be clear that not every computing mechanism is a  
27 computer, and not every computer is a programmable, stored-program  
28 computer. In order to have a programmable, stored-program computer,  
29 one needs to have a system of digits, a scheme for encoding instructions,  
30 a re-writable memory to store the instructions, and a processor that is  
31 appropriately organized to execute those instructions. Several technical  
32 problems need to be solved, such as how to retrieve instructions from  
33 memory in the right order and how to handle malformed instructions. I  
34 have briefly described these properties of computers, the problems that  
35 need to be solved to design them, and the solutions to those problems, in  
36 terms of the components of the mechanism and their functional organiza-  
37 tion. I will soon describe other properties of computers using the same  
38 strategy. This shows how the mechanistic account sheds light on the  
39 properties of computers.

### 40 41 3.3 SPECIAL-PURPOSE, GENERAL-PURPOSE, OR UNIVERSAL 42

43 Many old computers, which were not stored-program, were designed with  
44 specific applications in mind, such as solving certain classes of equations.

1 Some of these machines, like the ABC, were hardwired to follow a specific  
2 algorithm. They are called *special-purpose* computers to distinguish them  
3 from their general-purpose successors. Special-purpose computers are  
4 still used for a number of applications; for instance, computers in auto-  
5 mobiles are special-purpose.

6 In the 1940s, several computers were designed and built to be pro-  
7 grammable in the most flexible way, so as to solve as many problems as  
8 possible.<sup>15</sup> They were called *general-purpose* or, as von Neumann (1945)  
9 said, *all-purpose* computers. The extent to which computers are general-  
10 purpose is a matter of degree, which can be evaluated by looking at how  
11 much memory they have and how easy they are to program and use for  
12 certain applications.<sup>16</sup> General-purpose computers are programmable but  
13 need not be soft-programmable, let alone stored-program.

14 Assuming the Church-Turing thesis, namely the thesis that every  
15 effectively computable function is computable by a Turing machine, Alan  
16 Turing (1936–7) showed how to design universal computing mechanisms,  
17 i.e. computing mechanisms that would take any appropriately encoded  
18 program as input and respond to that program so as to compute *any*  
19 computable function.<sup>17</sup> Notice that Turing’s universal computing  
20 mechanisms, universal Turing machines, are not stored-program (see  
21 Section 4 below for a defense of this claim).

22 We have seen that soft programmable computers can respond to  
23 programs of unbounded length and manipulate their inputs (of finite but  
24 unbounded length) according to the instructions encoded in the program.  
25 This does not immediately turn them into universal computers, because,  
26 for example, they may not have the ability to handle branch-instructions.<sup>18</sup>  
27 But no one builds computers like that. Ordinary soft-programmable  
28 computers, designed to execute all the relevant kinds of instruction, are  
29 *universal* computing mechanisms. For example, ordinary computers like  
30 our desktop and laptop machines are universal in this sense. Even  
31 Charles Babbage’s legendary analytical engine, with minor modifications,  
32 would have been universal.<sup>19</sup> Universal computers can compute any Turing-  
33 computable function until they run out of time or memory. Because of  
34 this, the expressions ‘general-purpose computer’ and ‘all-purpose computer’  
35 are sometimes loosely used as synonyms of ‘universal computer’.

### 37 3.4 FUNCTIONAL HIERARCHIES

38  
39 Above, I briefly described the functional organization that allows soft-  
40 programmable computers to perform a finite number of primitive  
41 operations in response to a finite number of the corresponding kinds of  
42 instruction. Computers with this ability are computationally universal up  
43 to their memory and time limitations. In order to get them to execute any  
44 given program operating on any given notation, all that is needed is to

1 encode the given notation and program using the instructions of the  
2 computer's machine language.

3 In early stored-program computers, human programmers did this  
4 encoding manually. Encoding was slow, cumbersome, and prone to errors.  
5 To speed up the process of encoding and diminish the number of errors,  
6 early computer designers introduced ways to mechanize at least part of  
7 the encoding process, giving rise to a hierarchy of functional organizations  
8 within the stored-program computer. This hierarchy is made possible by  
9 automatic encoding mechanisms, such as assemblers, compilers, and  
10 operating systems. These mechanisms are nothing but programs executed  
11 by the computer's processor(s), whose instructions are often written in  
12 special, read-only memories. These mechanisms generate virtual memory,  
13 complex notations, and complex operations, which make the job of  
14 computer programmers and users easier and quicker. I will now briefly  
15 explain these three notions and their functions.

16 When a processor executes instructions, memory registers for instructions  
17 and data are functionally identified by the addresses of the memory  
18 registers that contain the data or instructions. Each memory register has  
19 a fixed storage capacity, which depends on the number of memory cells it  
20 contains. *Virtual memory* is a way to functionally identify data and  
21 instructions by virtual addresses, which are independent of the physical  
22 location of the data and instructions, so that a programmer or user need  
23 not keep track of the physical location of the data and instructions. During  
24 the execution phase, the physical addresses of the relevant memory  
25 registers are automatically generated by the compiler on the basis of the  
26 virtual addresses. Since virtual memory is identified independently of its  
27 physical location, in principle it has unlimited storage capacity, although  
28 in practice the total number of physical digits that a computer can store  
29 is limited by the size of its physical memory.<sup>20</sup>

30 In analyzing how a processor executes instructions, all data and instruc-  
31 tions must be described as binary strings (strings of bits) corresponding to  
32 the physical signals traveling through the processor, and the functional  
33 significance of these strings is determined by their location within an instruc-  
34 tion or data string. Moreover, all data and instruction strings have a fixed  
35 length, which corresponds to the length of the memory registers that contain  
36 them. *Complex notations*, instead, can contain any characters from any finite  
37 alphabet, such as the English alphabet. Programmers and users can use  
38 complex notations to form data structures that are natural and easy to  
39 interpret in a convenient way, similarly to natural languages. Thanks to  
40 virtual memory, these strings can be concatenated into strings of any length  
41 (up to the computer's memory limitations). During the execution phase, the  
42 encoding of data structures written in complex notations into data written  
43 in machine language is automatically taken care of by the functional hierarchy  
44 within the computer (operating system, compiler, and assembler).

1 The processor can only receive a finite number of instruction types  
2 corresponding to the primitive operations that the processor can execute.  
3 *Complex operations* are operations effectively defined in terms of primitive  
4 operations or in terms of already effectively defined complex operations.  
5 As long as the computer is universal, the Church-Turing thesis guarantees  
6 that any Turing-computable operation can be effectively defined in terms  
7 of the primitive operations of the computer. Complex operations can be  
8 expressed using a complex notation (e.g. an English word or phrase;  
9 for instance, a high-level programming language may include a control  
10 structure of the form UNTIL P TRUE DO \_\_\_ ENDUNTIL) that is more  
11 transparent to the programmers and users than a binary string would be,  
12 and placed in a virtual memory location. A *high-level programming*  
13 *language* is nothing but a complex notation that allows one to use a set  
14 of complex operations in writing programs. For their own convenience,  
15 programmers and users will typically assign a semantics to strings written  
16 in complex notations. This ascription of a semantics is often largely  
17 implicit, relying on the natural tendency of language users to interpret  
18 linguistic expressions of languages they understand. Thanks to virtual  
19 memory, complex instructions and programs containing them can be of  
20 any length (up to the memory limitations of the computers). During the  
21 execution of one of these programs, the encoding of the instructions  
22 into machine language instructions is automatically taken care of by the  
23 functional hierarchy within the computer.

24 Notice that the considerations made in this section apply only to pro-  
25 grammable, stored-program, universal computers. In order to obtain the  
26 wonderful flexibility of use that comes with the functional hierarchy of  
27 programming languages, one needs a very special kind of mechanism: a  
28 programmable, stored-program, universal computer.

29 The convenience of complex notations and the complex operations  
30 they represent, together with the natural tendency of programmers and  
31 users to assign them a semantics, makes it tempting to conclude that  
32 computers' inputs, outputs, and internal states are individuated by their  
33 content. For example, both computer scientists and philosophers some-  
34 times say that computers *understand* their instructions. The literal reading  
35 of this statement is discussed at length, and rejected, in Piccinini, 2004a  
36 and forthcoming b. Now we are ready to clarify in what sense computers  
37 do understand their instructions.

38 Computer instructions and data have functional significance, which  
39 depend on their role within the functional hierarchy of the computer.  
40 Before a program written in a complex notation is executed, its data and  
41 instructions are automatically encoded into machine language data and  
42 instructions. Then they can be executed. All the relevant elements of the  
43 process, including the programs written in complex notation, the pro-  
44 grams written in machine language, and the programs constituting the

1 functional hierarchy (operating system, compiler, and assembler), can be  
2 functionally characterized as strings of digits. All the operations performed  
3 by the processor in response to these instructions can be functionally  
4 characterized as operations on strings. The resulting kind of “computer  
5 understanding” is mechanistically explainable without ascribing any  
6 semantics to the inputs, internal states, or outputs of the computer.

7 At the same time, data and instructions are inserted into computers  
8 and retrieved from them by programmers and users who have their own  
9 purposes. As long as the functional hierarchy is working properly,  
10 programmers and users are free to assign a semantics to their data and  
11 instructions in any way that fits their purposes. This semantics applies  
12 naturally to the data and instructions written in the complex notation  
13 used by the programmers and users, although it will need to be replaced  
14 by a different semantics when the complex code is compiled and assembled  
15 into machine language data and instructions. The semantics of a computer’s  
16 inputs, outputs, and internal states is helpful in understanding how and  
17 why computers are used, but it is unnecessary to individuate computing  
18 mechanisms and the functions they compute.

### 20 3.5 DIGITAL VERSUS ANALOG

21  
22 Until now, I have restricted my account to devices that manipulate strings  
23 of digits. These devices are usually called *digital* computers (and calculators).  
24 There are also so-called *analog* computers, which were developed before  
25 digital computers but are no longer in widespread use. The distinction  
26 between analog and digital computers has generated considerable confusion.  
27 For example, it is easy to find claims to the effect that analog computers  
28 (or even analog systems in general) can be approximated to any desired  
29 degree of accuracy by digital computers, countered by arguments to the  
30 effect that some analog systems are computationally more powerful than  
31 Turing machines (Siegelmann, 1999). There is no room here for a detailed  
32 treatment of the digital-analog distinction in general, or even the more  
33 specific distinction between digital and analog computers. For present  
34 purposes, it will suffice to briefly draw some of the distinctions that are  
35 lumped together under the analog-digital banner, and then sketch an  
36 account of analog computers.

37 First, analog computation proper should be distinguished from analog  
38 modeling more generally. Sometimes, scale models and other kinds of  
39 “analog” model or modeling technologies (e.g. wind tunnels and certain  
40 electrical circuits) are called ‘analog computers’ (e.g. Hughes, 1999,  
41 p. 138). This use is presumably due to the fact that analog models, like  
42 analog computers properly so called, are used to draw inferences about  
43 other systems. An immediate concern with this use of the term ‘analog  
44 computer’ is that everything can be said to be analogous to something

1 else in some respect and used to draw inferences about it. This turns  
2 everything into an analog computer in this sense, which trivializes the  
3 notion of analog computer.

4 This concern can be alleviated by considering only mechanisms whose  
5 *function* is bearing certain analogies to certain other systems.<sup>21</sup> This  
6 functional restriction on analog models is similar to the functional  
7 restriction on computing mechanisms that grounds the present mechanistic  
8 account of computing mechanisms. Still, the resulting class of systems  
9 remains too broad for present purposes. Aside from the fact that both  
10 analog computers and other analog models are used for modeling  
11 purposes, this use of ‘analog computer’ has little to do with the standard  
12 notion of analog computer. Hence, it should be left out of discussions of  
13 computing mechanisms.

14 A second issue concerns whether a system is continuous or discrete.  
15 Analog systems are often said to be continuous, whereas digital systems  
16 are said to be discrete. When some computationalists claim that con-  
17 nectionist or neural systems are analog, their motivation seems to be  
18 that some of the variables representing connectionist systems can take a  
19 continuous range of values.<sup>22</sup> One problem with grounding the analog-digital  
20 distinction in the continuous-discrete distinction alone is that a system  
21 can only be said to be continuous or discrete under a given mathematical  
22 description, which applies to the system at a certain level of analysis.  
23 Thus, the continuous-discrete dichotomy, although relevant, seems insufficient  
24 to distinguish between analog and digital computers other than relative  
25 to a level.<sup>23</sup>

26 The only way to establish whether physical systems are ultimately  
27 continuous or discrete depends on fundamental physics. On one hand,  
28 some authors speculate that at the most fundamental level, everything  
29 will turn out to be discrete (e.g. Toffoli, 1984; Wolfram, 2002). If this were  
30 true, under this usage there would be no analog computers at the fundamental  
31 physical level. On the other hand, the physics and engineering of middle-  
32 sized objects are still overwhelmingly done using differential equations,  
33 which presuppose that physical systems as well as spacetime are continuous.  
34 This means that at the level of middle-sized objects, there should be no  
35 digital computers. But the notions of digital and analog computers have  
36 a well-established usage in computer science and engineering, which  
37 seems independent of the ultimate shape of physical theory. It is this usage  
38 that originally motivated computationalism. Therefore, the continuous-  
39 discrete distinction alone is not enough to draw the distinction between  
40 analog and digital computers that is of interest to computer scientists,  
41 engineers, and brain theorists.

42 Previous philosophical treatments of the digital-analog distinction have  
43 addressed a generic, intuitive distinction, with special emphasis on modes  
44 of representation, and did not take into account the functional properties

1 of different classes of *computers* (Goodman, 1968; Lewis, 1971; Haugeland,  
2 1981, and Blanchowicz, 1997). Those treatments do not serve our present  
3 purposes, for two reasons. First, our current goal is to understand computers  
4 *qua* computers and not *qua* representational systems. In other words, we  
5 should stay neutral on whether computers' inputs, outputs, or internal  
6 states represent, and if they do, on how they do so. Second, we are working  
7 within the mechanistic account of computing mechanisms, according to  
8 which computing mechanisms should be understood in terms of their  
9 (non-semantic) mechanistic properties.

10 Analog and digital computers are best distinguished by their different  
11 mechanistic properties.<sup>24</sup> Like digital computers, analog computers are  
12 made of (appropriately connected) input devices, output devices, and  
13 processing units (and in some cases, memory units). Like digital computers,  
14 analog computers have the function of generating outputs in accordance  
15 with a general rule whose application depends on their input. Aside from  
16 these broad similarities, however, analog computers are mechanistically  
17 very different from digital ones.

18 The most fundamental difference is in the inputs and outputs. Whereas  
19 the inputs and outputs of digital computers and their components are  
20 strings of digits, the inputs and outputs of analog computers and their  
21 components are what mathematicians call *real variables* (Pour-El, 1974).  
22 From a functional perspective, real variables are physical magnitudes that  
23 (i) *vary* over time, (ii) (are assumed to) take a *continuous range of values*  
24 within certain bounds, and (iii) (are assumed to) *vary continuously* over  
25 time. Examples of real variables include the rate of rotation of a mechanical  
26 shaft and the voltage level in an electrical wire.

27 The operations performed by computers are defined over their inputs  
28 and outputs. Whereas digital computers and their components perform  
29 operations defined over strings, analog computers and their components  
30 perform operations on real variables. Specifically, analog computers and  
31 their processing units have the function of transforming an input real variable  
32 into an output real variable that stands in a specified functional relation  
33 to the input. The discrete nature of strings makes it so that digital computers  
34 perform discrete operations on them (that is, they update their states only  
35 once every clock cycle), whereas the continuous change of a real variable  
36 over time makes it so that analog computers must operate continuously  
37 over time. By the same token, the rule that specifies the functional relation  
38 between the inputs and outputs of a digital computer is an effective  
39 procedure, i.e. a sequence of instructions, defined over strings from an  
40 alphabet, which applies uniformly to all relevant strings, whereas the "rule"  
41 that represents the functional relation between the inputs and outputs of  
42 an analog computer is a system of differential equations.

43 Due to the nature of their inputs, outputs, and corresponding operations,  
44 analog computers are intrinsically less precise than digital computers, for

1 two reasons. First, analog inputs and outputs can be distinguished from  
2 one another, by either a machine or an external observer, only up to a  
3 bounded degree of precision, which depends on the precision of the  
4 preparation and measuring processes, whereas by design, digital inputs  
5 and outputs can always be unambiguously distinguished. Second, analog  
6 operations are affected by the interference of an indefinite number of  
7 physical conditions within the mechanism, which are usually called  
8 “noise,” to the effect that their output is usually a worse approximation to  
9 the desired output than the input is to the desired input. These effects of  
10 noise may accumulate during an analog computation, making it difficult  
11 to maintain a high level of computational precision. Digital operations,  
12 by contrast, are unaffected by this kind of noise – either they are performed  
13 correctly, regardless of noise, or else they return incorrect results, in which  
14 case the system is said to malfunction.

15 Another limitation of analog computers, which does not affect their  
16 digital counterparts, is inherited from the limitations of any physical  
17 device. In principle, a real variable can take any real number as a value. In  
18 practice, a physical magnitude within a device can only take values within  
19 the bounds of the physical limits of the device. Physical components  
20 break down or malfunction if some of their relevant physical magnitudes,  
21 such as voltage, take values beyond certain bounds. Hence, the values of  
22 the inputs and outputs of analog computers and their components must  
23 fall within certain bounds; for example,  $\pm 100$  volts. Given this limitation,  
24 using analog computers requires that the problems being solved be  
25 appropriately scaled so that they do not require the real variables being  
26 manipulated by the computer to exceed the proper bounds of the computer’s  
27 components. This is an important reason why the solutions generated by  
28 analog computers need to be checked for possible errors by employing  
29 appropriate techniques (which often involve the use of digital computers).

30 Analog computers are designed and built primarily to solve systems of  
31 differential equations. The most effective general analog technique for  
32 this purpose involves successive integrations of real variables. Because of  
33 this, the crucial components of analog computers are *integrators*, whose  
34 function is to output a real variable that is the integral of their input real  
35 variable. The most general kinds of analog computer that have been built  
36 – *general-purpose analog computers* – contain a number of integrators  
37 combined with at least four other kinds of processing unit, which are  
38 defined by the operations they have the function to perform on their  
39 input. *Constant multipliers* have the function of generating an output real  
40 variable that is the product of an input real variable multiplied by a real  
41 constant. *Adders* have the function of generating an output real variable  
42 that is the sum of two input real variables. *Variable multipliers* have the  
43 function of generating an output real variable that is the product of two  
44 input real variables. Finally, *constant function generators* have the function

1 of generating an output whose value is constant. Many analog computers  
2 also include special components that generate real variables with special  
3 functional properties, such as sine waves. By connecting integrators and  
4 other components in appropriate ways, which may include feedback (i.e.  
5 recurrent) connections between the components, analog computers can  
6 be used to solve certain classes of differential equations.

7 Pure analog computers can be set up to perform different sequences of  
8 primitive operations, and in this sense, they are programmable. This  
9 notion is similar to that of hard programmability for digital computers,  
10 but unlike hard programmability, analog programmability is not a precursor  
11 to soft programmability, because it does not involve programs. Programs,  
12 which are the basis for soft programming digital computers, are  
13 sequences of instructions defined over strings. They are not defined over  
14 the real variables on which analog computers operate. Furthermore,  
15 programs cannot be effectively encoded as values of real variables. This is  
16 because for a program to be effectively encoded, the device that is  
17 responding to it must be able to unambiguously distinguish it from other  
18 programs. This can be done only if a program is encoded as a string.  
19 Effectively encoding programs as values of real variables would require  
20 unbounded precision in storing and measuring a real variable, which is  
21 beyond the limits of current (and foreseeable) analog technology.

22 Analog computers can be divided into special-purpose computers,  
23 whose function is to solve limited classes of differential equations, and  
24 general-purpose computers, whose function is to solve larger classes of  
25 differential equations. Insofar as the distinction between special- and general-  
26 purpose analog computers has to do with flexibility in their application,  
27 it is analogous to the distinction between special- and general-purpose  
28 digital computers. But there are important disanalogies: these two distinctions  
29 rely on different functional properties of the relevant classes of devices,  
30 and the notion of general-purpose analog computer, unlike its digital  
31 counterpart, is not an approximation of Turing's notion of computational  
32 universality (see Section 3.3 above). Computational universality is a  
33 notion defined in terms of computation over strings, so analog computers  
34 – which do not operate on strings – are not devices for which it makes  
35 sense to ask whether they are computationally universal. Moreover,  
36 computationally universal mechanisms are computing mechanisms that  
37 are capable of responding to any program (written in an appropriate  
38 language). We have already seen that pure analog computers are not in  
39 the business of executing programs; this is another reason why analog  
40 computers are not in the business of being computationally universal.

41 It should also be noted that general-purpose analog computers are not  
42 maximal kinds of computer in the sense in which standard general-  
43 purpose digital computers are. At most, a digital computer is capable of  
44 computing the class of Turing-computable functions.<sup>25</sup> By contrast, it

1 may be possible to extend the general-purpose analog computer by adding  
2 components that perform different operations on real variables, and the  
3 result may be a more powerful analog computer.<sup>26</sup>

4 Since analog computers do not operate on strings, we cannot apply  
5 Turing's notion of computable functions over strings directly to measure  
6 the power of analog computers. Instead, we can measure the power of  
7 analog computers by employing the notion of function of a real variable.  
8 Refining work by Shannon (1941), Pour-El identified precisely the class of  
9 functions of a real variable that can be generated by general-purpose analog  
10 computers. They are the differentially algebraic functions, namely, functions  
11 that arise as solutions to algebraic differential equations (Pour-El, 1974;  
12 see also Lipshitz and Rubel, 1987; and Rubel and Singer, 1985). Algebraic  
13 differential equations are equations of the form  $P(y, y', y'', \dots, y^{(n)}) = 0$ ,  
14 where  $P$  is a polynomial with integer coefficients and  $y$  is a function of  $x$ .  
15 It has also been shown that there are algebraic differential equations that  
16 are "universal" in the sense that any continuous function of a real variable  
17 can be approximated with arbitrary accuracy over the whole positive time  
18 axis  $0 \leq t < \infty$  by a solution of the equation. Corresponding to such universal  
19 equations, there are general-purpose analog computers with as little as  
20 four integrators whose outputs can, in principle, approximate any con-  
21 tinuous function of a real variable arbitrarily well (see Duffin, 1981 and  
22 Boshernitzan, 1986).

23 We have seen that analog computers do not do everything that digital  
24 computers do; in particular, they do not perform operations defined over  
25 strings of digits and do not execute programs. On the other hand, there is  
26 an important sense in which digital computers can do everything that  
27 general-purpose analog computers can. Rubel has shown that given any  
28 system of algebraic differential equations and initial conditions that  
29 describe a general-purpose analog computer  $A$ , it is possible to effectively  
30 derive an algorithm that will approximate  $A$ 's output to an arbitrary  
31 degree of accuracy (Rubel, 1989). From this, however, it doesn't follow that  
32 the behavior of every physical system can be approximated to any desired  
33 degree of precision by digital computers.

34 Some limitations of analog computers can be overcome by adding  
35 digital components to them and by employing a mixture of analog and  
36 digital processes. A detailed treatment of hybrid computing goes beyond  
37 the scope of this paper. Suffice it to say that the last generation of analog  
38 computers to be widely used were analog-digital hybrids, which contained  
39 digital memory units as well as digital processing units capable of being  
40 soft-programmable (Korn and Korn, 1972). In order to build a stored-program  
41 or soft-programmable analog computer, one needs digital components,  
42 and the result is a computer that owes the interesting computational  
43 properties that it shares with digital computers (such as being stored-  
44 program and soft-programmable) to its digital properties.

1 Given how little (pure) analog computers have in common with digital  
2 computers, calculators, and other computing mechanisms, and given that  
3 analog computers do not even perform computations in the sense defined  
4 by the mathematical theory of computation, one may wonder why both  
5 classes of devices are called ‘computers’. The answer lies in the history of  
6 these devices. As I mentioned above, the term ‘computer’ was apparently  
7 first used for a machine (as opposed to a computing human) by John  
8 Atanasoff in the early 1940s. At that time, what we now call analog  
9 computers were called *differential analyzers*. Digital machines operating  
10 on strings of digits, such as Atanasoff’s ABC, were often designed to  
11 solve problems similar to those solved by differential analyzers – namely,  
12 solving systems of differential equations – by the manipulation of strings.  
13 Since the new digital machines operated on digits and could replace  
14 computing humans at solving complicated problems by following algo-  
15 rithms, they were dubbed ‘computers’. The differential analyzers soon  
16 came to be re-named ‘analog computers’, perhaps because both classes of  
17 machines were initially designed for similar practical purposes, and for a  
18 few decades they competed with each other. These historical factors should  
19 not blind us to the fact that analog computers manipulate vehicles and  
20 perform operations that are radically different from those of digital  
21 computers. Atanasoff himself, for one, was clear about this: his earliest  
22 way to distinguish digital computing mechanisms from analog machines  
23 like differential analyzers was to call the former “computing machines  
24 proper” (quoted by Burks, 2002, p. 101).

### 26 3.6 SERIAL VERSUS PARALLEL

28 The distinction between parallel and serial computers has also generated  
29 some confusion. A common claim is that the brain cannot be a “serial  
30 computer” like our ordinary digital computers, because it is “parallel”  
31 (e.g. Churchland *et al.*, 1990, p. 47; Dennett, 1991, p. 214; Churchland  
32 and Sejnowski, 1992, p. 7). In evaluating this claim, we should keep in  
33 mind that digital computers can be parallel too, in several senses of the  
34 term. There are several distinctions to be made, and our understanding of  
35 computers can only improve if we make at least the main ones explicit.

36 First, there is the question of whether in a computing mechanism only  
37 one computationally relevant *event* – for instance, the transmission of a  
38 signal between components – can occur during any relevant time interval.  
39 In this limited sense, virtually all complex computing mechanisms are  
40 parallel. For example, in most computing mechanisms, the existence of  
41 many communication lines between different components allows data to  
42 be transmitted in parallel. Even Turing machines do at least two things  
43 for every instruction they follow: they act on their tape and update their  
44 internal state.

1 A separate question is whether a computing mechanism can perform  
2 only one or more than one *computational operation* during any relevant  
3 time interval. Analog computers are parallel in this sense. This appears to  
4 be the sense that connectionist computationalists appeal to when they say  
5 that the brain is “massively parallel.” But again, most complex computing  
6 mechanisms, such as Boolean circuits, are parallel in this sense. Since  
7 most (digital) computers are made out of Boolean circuits and other  
8 computing components that are parallel in the same sense, they are  
9 parallel in this sense too. In this respect, then, there is no principled difference  
10 between ordinary computers and connectionist computing systems.

11 A third question is whether a computing mechanism executes one or  
12 more than one *instruction* at a time. There have been attempts to design  
13 parallel processors, which perform many operations at once by employing  
14 many executive units (such as datapaths) in parallel. Another way to  
15 achieve the same goal is to connect many processors together within one  
16 supercomputer. There are now in operation supercomputers that include  
17 thousands of processors working in parallel. The difficulty in using these  
18 parallel computers consists of organizing computational tasks so that  
19 they can be modularized, i.e. divided into sub-problems that can be solved  
20 independently by different processors. Instructions must be organized so that  
21 executing one (set of) instruction(s) is not a prerequisite for executing  
22 other (sets of) instructions in parallel to it (them) and does not interfere  
23 with their execution. This is sometimes possible and sometimes not,  
24 depending on which part of which computational problem is being  
25 solved. Some parts of some problems can be solved in parallel, but others  
26 can't, and some problems must be solved serially. Another difficulty is  
27 that in order to obtain the benefits of parallelism, typically the size of the  
28 hardware that must be employed in a computation grows (linearly) with  
29 the size of the input. This makes for prohibitively large (and expensive)  
30 hardware as soon as the problem instances to be solved by parallel  
31 computation become nontrivial in size. This is the notion of parallelism  
32 that is most relevant to computability theory. Strictly speaking, it applies  
33 only to computing mechanisms that execute instructions. Hence, it is  
34 irrelevant to ordinary analog computers and connectionist computing  
35 systems, which do not execute instructions.

36 In the connectionist literature, there is a persistent tendency to call  
37 neurons ‘processors’ (e.g. Siegelmann, 2003). In many cases, this language  
38 implicitly or explicitly suggests an analogy between a brain and a parallel  
39 computer that contains many processors, so that every neuron corresponds  
40 to one processor. This is misleading, because there is no useful sense in  
41 which a neuron can be said to execute instructions in the way that a  
42 computer processor can. In terms of their functional role within a  
43 computing mechanism, neurons compare more with logic gates than with  
44 the processors of digital computers. In fact, modeling neurons as logic

1 gates was the basis for the first formulation of a computational theory of  
2 the brain (McCulloch and Pitts, 1943). Nevertheless, it may be worth  
3 noticing that if the comparison between neurons and processors is taken  
4 seriously, then in this sense – the most important for computability  
5 theory – neurons are *serial* processors, because they perform only one  
6 functionally relevant activity (i.e. they fire at a certain rate) during any  
7 relevant time interval. Even if we consider an entire connectionist  
8 computing system as a processor, we obtain the same result; namely, that  
9 the network is a serial processor (it turns one input string into one output  
10 string). However, neither neurons nor ordinary connectionist computing  
11 mechanisms are really comparable to computer processors in terms of  
12 their organization and function – they certainly don't execute instructions.  
13 So, to compare connectionist systems and computer processors in this  
14 respect is inappropriate.<sup>27</sup>

15 Finally, there is the question of whether a processor starts executing an  
16 instruction only after the end of the execution of the preceding instruction,  
17 or whether different instructions are executed in an overlapping way. The  
18 latter becomes possible when the processor is organized so that the different  
19 activities that are necessary to execute instructions (e.g. fetching an  
20 instruction from memory, performing the corresponding operation, and  
21 writing the result in memory) can all be performed at the same time on  
22 different instructions by the same processor. This kind of parallelism in  
23 the execution of instructions diminishes the global computation time for  
24 a given program, and it applies only to processors that execute instructions.  
25 It is a common feature of contemporary computers, where it is called  
26 *pipelining*.<sup>28</sup>

27 The above parallel-serial distinctions apply clearly to computing  
28 mechanisms, but the parallel-serial distinction is obviously broader than  
29 a distinction between modes of computing. Many things other than  
30 computations can be done in parallel. For example, instead of digging ten  
31 holes by yourself, you can get ten people to dig ten holes at the same time.  
32 Whether a process is serial or parallel is a different question from whether  
33 it is digital or analog (in various senses of the term), computational or  
34 non-computational.

#### 35 36 37 **4. Application 1: are Turing machines computers?** 38

39 Turing machines (TMs) can be seen and studied mathematically as lists  
40 of instructions or sequences of abstract strings, with no mechanistic  
41 counterpart. They can also be seen as a type of computing mechanism,  
42 made out of a tape divided into squares and a unit that moves along the  
43 tape, acts on it, and is in one out of a finite number of internal states  
44 (see Davis *et al.*, 1994 for more details). The tape and the active unit are

1 the components of TMs. Their functions are, respectively, storing digits  
2 and performing operations on digits. The active unit of TMs is typically  
3 treated as a black box, though in principle it may be analyzed as a finite  
4 state automaton, which in turn may be analyzed as a Boolean circuit plus  
5 a memory register. When they are seen as mechanisms, TMs fall naturally  
6 under the present account of computing mechanisms. They are uniquely  
7 defined by their mechanistic description, which includes their list of  
8 instructions.

9 There are two importantly different classes of TMs: ordinary TMs and  
10 universal TMs. Ordinary TMs are not programmable and *a fortiori* not  
11 universal. They are “hardwired” to compute one and only one function,  
12 as specified by the list of instructions that uniquely individuates each TM.  
13 Since TMs’ instructions constitute full-blown algorithms, by the light of  
14 the present account they are special-purpose computers.

15 By contrast, universal TMs are programmable (and of course universal),  
16 because they respond to a portion of the digits written on their tape by  
17 computing the function that would be computed by the TM encoded by  
18 those digits. Nevertheless, universal TMs are *not* stored-program computers,  
19 because their architecture has no memory component – separate from its  
20 input and output devices – for storing data, instructions, and results. The  
21 programs for universal TMs are stored on the same tape that contains the  
22 input and the output. In this respect, universal TMs are somewhat analogous  
23 to early punched cards computers. The tape can be considered an input  
24 and output device, and with some semantic stretch, a memory. But since  
25 there is no distinction between input device, output device, and memory, a  
26 universal TM should not be considered a stored-program computer properly  
27 so called, for the same reason that most punched cards machines aren’t.

28 This account contrasts with a common way of understanding TMs,  
29 according to which TM tapes are analogous to internal computer memor-  
30 ies, but it is in line with Turing’s original description of his machines:

31  
32 We may compare a man in the process of computing a real number to a machine  
33 which is only capable of a finite number of conditions  $q_1, q_2, \dots, q_R$  which will be called  
34 “*m*-configurations”. *The machine is supplied with a “tape”* (the analogue of paper) run-  
35 ning through it, and divided into sections (called “squares”) each capable of bearing  
36 a “symbol”. At any moment *there is just one square*, say the *r*-th, bearing the symbol  $G(r)$   
37 which is “*in the machine*”. We may call this square the “scanned square”. The symbol on  
38 the scanned may be called the “scanned symbol”. The “scanned symbol” is the only one of  
39 which the machine is, so to speak, “directly aware”. However, *by altering its m-configuration*  
40 *the machine can effectively remember* some of the symbols which it has “seen” (scanned)  
41 previously. (Turing, 1936–7, p. 117; emphasis added)

42  
43 Turing defines his machines by analogy with computing humans. As  
44 Eli Dresner (2003) points out, Turing does not even describe his machines

1 as including the tape as a component, let alone as a memory component  
2 – the tape is “the analogue of paper” on which a human writes. For Tur-  
3 ing, the analogue of human memory is not the machine’s tape, but the  
4 machine’s internal states (*m*-configurations). Only in 1945 and afterwards,  
5 when the design and construction of stored-program computers were  
6 under way, did Turing describe TM tapes as components of the machine  
7 and draw a functional analogy between TM tapes and computer memory  
8 (Turing, 1945). Still, this analogy remains only partial, because computer  
9 memories properly so called are distinct from input and output devices.  
10 The functional distinction between memory, input devices, and output  
11 devices renders stored-program computers much more flexible in their  
12 use than literal TMs would be, making it possible to install and remove  
13 programs from memory, store multiple programs, switch between them,  
14 create a functional hierarchy, etc. These, in turn, are some reasons why no  
15 one builds literal implementations of TMs for practical applications.<sup>29</sup>

16 Even after the invention of TMs, it was still an important conceptual  
17 advance to design machines that could store instructions in an internal  
18 memory component. So, it is anachronistic to attribute the idea of the stored-  
19 program computer to Turing (as done, e.g. by Aspray, 1990 and Copeland,  
20 2000). This conclusion exemplifies how a nuanced, mechanistic under-  
21 standing of computing mechanisms sheds light on the history of computing.  
22  
23

## 24 **5. *Application 2: a taxonomy of computationalist theses***

25  
26 Computing mechanisms have been employed in computational theories  
27 of mind and brain. These theories are sometimes said to be metaphors  
28 (Eliasmith, 2003), but a careful reading of the relevant literature shows  
29 that computationalism is a literal mechanistic hypothesis (Piccinini,  
30 2003b, 2004b). The mechanistic account of computing mechanisms can  
31 be used to add precision to computationalist theses about the brain by  
32 taxonomizing them in order of increasing strength. The theses range from  
33 the commonsensical and uncontroversial thesis that the brain processes  
34 information to the strong thesis that it is a programmable, stored-  
35 program, universal computer. Each thesis presupposes the truth of the  
36 preceding one and adds further assumptions to it:  
37

- 38 1) The brain is a collection of interconnected neurons that deal with  
39 information and control (in an intuitive, formally undefined sense).
- 40 2) Networks of neurons are computing mechanisms (generic connec-  
41 tionist computationalism).
- 42 3) Networks of neurons are Boolean circuits or finite state automata  
43 (McCulloch and Pitts, 1943; Nelson, 1982).
- 44 4) The brain is a programmable computer (Devitt and Sterelny, 1999).

- 1 5) The brain is a programmable, stored-program computer (Fodor,  
2 1975).<sup>30</sup>  
3 6) The brain is a universal computer (Newell and Simon, 1976).  
4

5 Strictly speaking, (6) does not presuppose (5). For instance, universal  
6 TMs are not stored-program. In practice, however, all supporters of (6)  
7 also endorse (5), for the good reason that there is no evidence of a storage  
8 system in the environment – analogous to the tape of TMs – that would  
9 store the putative programs executed by brains.

10 Thesis (1) is sufficiently weak and generic that it entails nothing con-  
11 troversial about neural mechanisms. Even a non-cognitivist (e.g. a  
12 behaviorist) should agree with it. I mention it here to distinguish it from  
13 nontrivial computationalist theses and leave it out of the discussion.

14 Historically, thesis (3) is the first formulation of computationalism.  
15 Its weakening leads to (2), which includes modern connectionist com-  
16 putationalism. Its strengthening leads to what is often called classical  
17 computationalism. Classical computationalism is usually identified with  
18 (5) or (6), but sometimes (4) and even (3) have been discussed as options.  
19 The above taxonomy allows us to see that different formulations of com-  
20 putationalism are not equivalent to one another, and that they vary in the  
21 strength of their assumptions about the mechanistic properties of neural  
22 mechanisms.

23 The above list includes only the theses that have figured prominently in  
24 the computationalist literature. It is by no means exhaustive of possible  
25 computationalist theses. First, notice that (3) can be divided into a relatively  
26 weak thesis, according to which the brain is a collection of Boolean cir-  
27 cuits, and a much stronger one, according to which the brain is a collec-  
28 tion of finite state automata. The mechanistic account of computing  
29 mechanisms shows how to construct theses that are intermediate in  
30 strength between these two. For instance, one could hypothesize that the  
31 brain is a collection of components that can execute complex pseudo-  
32 algorithms (i.e. effective procedures defined only on finitely many inputs  
33 of bounded length), like the multiplication and division components of  
34 modern computers.

35 Another possible version of computationalism is the hypothesis that  
36 the brain is a calculator. This possibility is intriguing because I know of  
37 no one who has ever proposed it, even though calculators are *bona fide*  
38 computing mechanisms. The mechanistic account sheds light on this fact.  
39 Among other limitations calculators have, their computational repertoire  
40 is fixed. There is no interesting sense in which they can learn to compute  
41 new things or acquire new computational capacities. One important  
42 factor that attracted people to computationalism is the flexibility and  
43 power of computers, flexibility and power that calculators lack. Because of  
44 this, it is not surprising that no one has proposed that brains are calculators.

1 The kinds of computer that are most strongly associated with com-  
2 putationalism are programmable, stored-program, universal computers.  
3 Again, the mechanistic account sheds light on this fact: those machines,  
4 unlike other computing mechanisms, have the exciting property that given  
5 appropriate programs, they can automatically compute any computable  
6 function and even modify their own programs.  
7  
8

### 9 **6. *Application 3: questions of hardware***

10  
11 It is often said that even if the brain is a computing mechanism, it need  
12 not have a von Neumann architecture (Pylyshyn, 1984; Churchland and  
13 Sejnowski, 1992). In these discussions, ‘von Neumann architecture’ is  
14 used as a generic term for the functional organization of ordinary digital  
15 computers. The claim that the brain need not have a von Neumann  
16 architecture is used to discount apparent dissimilarities between the  
17 functional organization of brains and that of ordinary digital computers  
18 as irrelevant to computationalism. The idea is that brains may compute  
19 by means other than those exploited by modern digital computers.

20 It is true that not all computing mechanisms need have a von Neumann  
21 architecture. For example, Turing machines don’t. But this does not  
22 eliminate the constraints that different versions of computationalism put  
23 on the functional organization of the brain, if the brain is to perform the  
24 relevant kinds of computation. In the current discussion, I intentionally  
25 avoided the term ‘von Neumann architecture’ because it is so generic that  
26 it obscures the many issues of functional organization that are relevant  
27 to the design and computing power of computing mechanisms. The present  
28 account allows us to increase the precision of our claims about computer and  
29 brain architectures, avoiding generic terms like ‘von Neumann architecture’  
30 and focusing on various mechanistic properties of computing mechanisms  
31 (and hence on what their computing power is).

32 If the brain is expected to be a programmable, stored-program, universal  
33 computer, as it is by some versions of computationalism, it must contain  
34 programs as well as components that store and execute programs. More  
35 generally, any kind of computation, even the most trivial transformation  
36 of one digit into another (as performed by a NOT gate) requires appropriate  
37 hardware. So any nontrivial computationalist thesis, depending on the  
38 computation power it ascribes to the brain, constrains the functional  
39 properties that brains must exhibit if they are to perform the relevant  
40 computations. The following are general questions about neural hardware  
41 that apply to some or all computationalist theses about the brain:  
42

- 43 1. What are the digits manipulated in the neural computation, and  
44 what are their types?

- 1     **2.** What are the elementary computational operations on neural digits,  
2     and what are the components that perform them?
- 3     **3.** How are the digits concatenated to one another, so that strings of  
4     them can be identified as inputs, internal states, and outputs of  
5     neural mechanisms and nontrivial computations from input strings  
6     to output strings can be ascribed to neural mechanisms?
- 7     **4.** What are the compositional rules between elementary operations,  
8     and the corresponding ways to connect the components, such that  
9     complex operations can be formed out of elementary ones and  
10    performed by the mechanism?
- 11    **5.** If the mechanism stores programs or even just data for the com-  
12    putations, what are the memory cells and registers and how do  
13    they work?
- 14    **6.** What are the control units that determine which operations are  
15    executed at any given time and how do they work? This question  
16    is particularly pressing if there has to be execution of programs,  
17    because the required kind of control unit is particularly sophisticated  
18    and needs to correctly coordinate its behavior with the components  
19    that store the programs.

21     When McCulloch and Pitts (1943) initially formulated computational-  
22     ism, they had answers to the relevant versions of the above questions. In  
23     answer to (1), they argued that the presence and the absence of a neural  
24     spike are the two types of digit on which neural computations are defined.  
25     In answer to (2), they appealed to Boolean operations and claimed that  
26     they are performed by neurons. In answer to (3) and (4), they relied on a  
27     formalism they in part created and in part drew from Carnap, which is  
28     computationally equivalent to finite state automata. In answer to (5), they  
29     hypothesized that there are closed loops of neural activity, which act as  
30     memory cells. In answer to (6), they largely appealed to the innate wiring  
31     of the brain.<sup>31</sup>

32     When von Neumann formulated his own version of computationalism  
33     (von Neumann, 1958), he also tried to answer at least the first two of the  
34     above questions. In answer to (1), he maintained that the firing rates of  
35     neurons are the digit types. In answer to (2), he maintained the elementary  
36     operations are arithmetical and logical operations on these firing rates.  
37     Although von Neumann's answers take into account the functional sig-  
38     nificance of neuronal spikes as it is understood by modern neurophysiologists,  
39     von Neumann did not have answers to questions 3 to 6, and he explicitly  
40     said that he did not know how the brain could possibly achieve the degree  
41     of computational precision that he thought it needed.<sup>32</sup>

42     Today's computationalists no longer believe McCulloch's or von  
43     Neumann's versions of computationalism. But if computationalism is to  
44     remain a substantive, empirical hypothesis about the brain, these questions

1 need to find convincing answers. If they don't, it may be time to abandon  
2 computationalism in favor of other mechanistic explanations of cognitive  
3 processes.  
4

## 6 7. *Against computational nihilism*

8 Before concluding, I wish to go back to computational nihilism – the  
9 view that whether something is a computer is in the eyes of the beholder.  
10 If everything may be described as a computer merely depending on how  
11 we look at it, the present account might appear futile. The present  
12 account distinguishes between computers and other things, as well as  
13 different kinds of computers, on the grounds of their mechanistic propert-  
14 ies. But if, in fact, whether something is a computer is in the eyes of the  
15 beholder, these might be distinctions without a difference.

16 My first reply to this worry is that computational nihilism makes no  
17 sense of our successful practices of designing, building, and using computers.  
18 We deal with computers because they have special capacities – capacities  
19 that other things lack. Different kinds of computers have different  
20 specific capacities. My account makes sense of these facts in terms of the  
21 mechanistic properties of different systems. By contrast, computational  
22 nihilism makes little if any sense of our successful practices of designing,  
23 building, and using computers. This reply shifts the burden on the nihilist but  
24 does not address the source of her worry. To do that, I will briefly address  
25 several different motivations for nihilism. I will find them all wanting.

26 Some authors have argued that everything performs *every* computation,  
27 or at least a large range of non-equivalent computations (Putnam, 1988;  
28 Searle, 1992; Dardis, 1993; Ludlow, 2003). According to this *panpancom-*  
29 *putationalism*, describing a system as performing a computation is akin  
30 to mapping the sequence of states defined by a computational description  
31 onto the sequence of states defined by a physical description of a system,  
32 without constraints on what constitutes an acceptable mapping. If this is  
33 an adequate account of computational description, it is not difficult to  
34 show that every dynamical system performs an indefinite number of non-  
35 equivalent computations, because all of those computations may be  
36 mapped onto the physically defined state transitions of the system.

37 But as many have pointed out, this construal of computational description  
38 is implausibly weak – it has little to do with the way computational  
39 descriptions are used in either science or common sense. For a computational  
40 description to be properly said to apply to a physical system, it must  
41 respect the system's causal structure. When this is the case, it is far from  
42 true that everything may be described as performing every computation  
43 (Searle, 1992, p. 209; Chalmers, 1994, 1996; Chrisley, 1995; Copeland,  
44 1996; Bontly, 1998; Scheutz, 2001).

1 A second version of pancomputationalism is that everything is a  
2 computer because computational properties are individuated semantically  
3 and everything may be interpreted as possessing semantic properties (e.g.  
4 Shagrir, 2006). This view is implicit in Churchland and Sejnowski's words,  
5 which I quoted at the beginning, to the effect that being a computer is the  
6 "interest-relative property that someone sees value in interpreting a  
7 system's states as representing states of some other system, and the prop-  
8 erties of the system support such an interpretation."

9 This semantic pancomputationalism may be rebutted in two ways.  
10 First, perhaps possessing semantic properties is *not* interest-relative and  
11 thus it's false that everything possesses semantic properties in the relevant  
12 sense. Many philosophers have defended realism about semantic properties  
13 and argued that at least some semantic properties, including those consti-  
14 tutive of computing mechanisms, are possessed only by special systems  
15 (e.g. Fodor, 1990). Although I am a realist about semantic properties, I  
16 endorse a more radical rebuttal to semantic pancomputationalism. Along  
17 with others, I have argued that computational properties need not be  
18 individuated semantically (Piccinini, 2004a, forthcoming b; cf. also Stich,  
19 1983; Egan, 1995). In fact, the account of computers articulated above  
20 made no reference to semantic properties. For present purposes, com-  
21 putational properties are not individuated semantically. Therefore, whether  
22 everything may be seen as possessing semantic properties is irrelevant to  
23 whether everything is a computer.

24 A third version of pancomputationalism is the view that everything is a  
25 computer because computational properties are causal properties of physical  
26 systems described at a high level of abstraction, and every physical system  
27 has causal properties. This view is well expressed by the following passage:

28  
29 [T]he mathematical theory based on recursion theory, Turing machines, complexity analyses,  
30 and the like – widely known as the "Theory of Computation" – is neither more nor less than  
31 a *mathematical theory of the flow of causality*. (Smith, 2002, p. 43; emphasis original)  
32

33 This causal pancomputationalism is consistent with the reply to  
34 *panpancomputationalism* mentioned above. Many proponents of that  
35 reply explicitly endorse a causal version of pancomputationalism.<sup>33</sup>

36 Being grounded in real causal properties, causal pancomputationalism  
37 is not as nihilistic as *panpancomputationalism* and semantic pancom-  
38 putationalism. But it's still at odds with the present account, according to  
39 which computers are a special subset of computing mechanisms, which in  
40 turn are a special subset of all systems. Drawing from Piccinini, 2007, we  
41 may respond to causal pancomputationalism by distinguishing two kinds  
42 of computational description: computational *models* and computational  
43 *explanations*. Computational models are descriptions that capture some  
44 aspects of the causal structure of a system for some modeling purposes.

1 The version of pancomputationalism under discussion has some plausibility  
2 in the weak sense that everything may be described by a computational  
3 model. Even this weak formulation, however, is true only if we are not  
4 concerned with how accurate our models are. If we are concerned with  
5 the accuracy of our models – as we typically are – this formulation turns  
6 into the thesis that everything may be modeled computationally to a certain  
7 degree of approximation. This thesis deserves careful scrutiny, but there is  
8 neither room nor reason to address it in depth here. For here I am not  
9 concerned with computational modeling. I am concerned with computational  
10 explanation.

11 When we literally describe something as a computer – or more generally,  
12 as a computing mechanism – we are not simply modeling its behavior.  
13 Rather, we are *explaining* its behavior in a special way. That is to say, we  
14 are appealing to a special kind of process – computation – to explain the  
15 capacities of the system. Computational explanation applies only to a  
16 relatively small class of systems; most behaviors of most systems are not  
17 explained computationally. For instance, both hurricane Katrina and  
18 your desktop machine may be described by a computational model. But  
19 only the latter's behavior is explained by its inner computations.<sup>34</sup>

20 In this paper, I relied on a mechanistic account of computational  
21 explanation that I have defended at length elsewhere (Piccinini, 2007,  
22 forthcoming a, b). The mechanistic account is inspired by recent work in  
23 the philosophy of science and by the way computing mechanisms are  
24 individuated and their capacities are explained in computer science and  
25 computability theory. According to the mechanistic account, a computing  
26 mechanism is a mechanism whose *function* is to perform computations.

27 Appealing to functions in an account of computing mechanisms may  
28 appear questionable or even question-begging. For an account couched in  
29 terms of functional properties supports an objective notion of computing  
30 mechanism only in so far as functional properties are objective. John  
31 Searle maintains just the opposite:

32  
33 [T]o say that something is *functioning as* a computational process is to say something  
34 more than that a pattern of physical events is occurring. It requires the assignment of a  
35 computational interpretation by some agent. Analogously, we might discover in nature  
36 objects that had the same sort of shape as chairs and that could therefore be used as chairs;  
37 but we could not discover objects in nature which were functioning as chairs, except  
38 relative to some agents who regarded them or used them as chairs. (Searle, 1992, p. 211;  
39 emphasis original)

40  
41 Searle argues that functional properties are not intrinsic to physics:  
42 intrinsic physical properties do not dictate one functional description over  
43 another; hence, functional descriptions are observer-relative. Since  
44 computational descriptions are functional, they are observer-relative

1 too. From this, Searle concludes that computational properties cannot be  
2 discovered and characterized by natural science (cf. Searle, 1992, p. 212).<sup>35</sup>  
3 This objection – that if computational descriptions are functional, then  
4 they are observer-relative in a sense that makes them unscientific – is  
5 the last source of computational nihilism that I will discuss. It is refuted  
6 by the following considerations.

7 First, it would be hasty to maintain that observer-relative descriptions  
8 are ipso facto scientifically illegitimate. Some bona fide physical descriptions,  
9 such as descriptions of position and velocity according to Relativity  
10 Theory, are observer-relative in the sense that observers within different  
11 frames of references will assign different positions and velocities to the  
12 same objects. Other bona fide physical descriptions, such as descriptions  
13 of position and momentum according to Quantum Mechanics, are observer-  
14 relative in the sense that a system's variable is assigned determinate values  
15 only after a measurement operation, and the specific value that is  
16 assigned to the variable depends on which measurement operation is  
17 performed. In short, it takes more than observer-relativity to conclude  
18 that a description is unscientific.

19 By 'observer-relative predicate', Searle means a predicate that refers to  
20 properties that require the existence of observers and their intentionality  
21 (ibid., p. 211). To avoid confusion with other notions of observer-relativity,  
22 I will refer to predicates of this kind as *intentionality-relative*. Still, not all  
23 intentionality-relative predicates are scientifically problematic to the same  
24 degree. On one hand, there are intentionality-relative predicates, such as  
25 'tasting better than chocolate,' which are of dubious scientific usefulness  
26 (at least outside the psychology of taste). On the other hand, there are  
27 other intentionality-relative properties, such as "being useful to kill,"  
28 which refer to objective properties. A typical goal of forensics scientists is  
29 to discover and characterize the means by which victims are murdered.  
30 Even if we were to grant, for the sake of the argument, that functions  
31 are intentionality-relative, Searle would still have to show that they are  
32 unscientific. Unfortunately, he does not.

33 Second, it remains to be seen whether functional properties are  
34 intentionality-relative. Insofar as they are relevant here, they are not.  
35 Searle writes as if properties must be either intrinsic or intentionality-  
36 relative. But this is a false dichotomy. Typical relational properties are neither  
37 intrinsic nor intentionality-relative (e.g. having less mass than Pluto).  
38 Functional properties are relational, and hence they are not intrinsic. But  
39 the functional properties my account relies on are not intentionality-relative.  
40 This applies to both functioning as and having a function.

41 Although *functioning as an X* is not an intrinsic property of a system,  
42 its instantiation does not depend on the intentionality of any observers.  
43 An object can function as a chair only if it has a flat surface standing  
44 above the ground, an appropriate size and rigidity, etc., all of which are

1 intentionality-independent properties. In addition, an object actually  
2 functions as a chair only if its users relate to it in an appropriate way.  
3 Whether the appropriate relation holds depends on what users actually  
4 do – it is not influenced by the intentionality of observers. Analogously,  
5 an object can function as a computer only if it possesses certain relevant  
6 (intentionality-independent) properties, and it actually functions as a com-  
7 puter only if it is used in the relevant way.

8 A similar point applies to *having function F*. True, most chairs wouldn't  
9 exist if they weren't (intentionally) built by human beings, and human  
10 beings are observers *par excellence*. Still, it doesn't follow that chairs having  
11 their function depends on the existence of *observers* with their intentionality.  
12 The dependence of chairs on observers is a contingent fact, due to the  
13 coincidence that most chair builders count as observers. Chairs retain  
14 their characteristic function – to be sat on – regardless of whether anyone  
15 observes them. The point is especially obvious in the case of functions  
16 acquired through use rather than design (like a stone used to sit on), func-  
17 tions of animal artifacts (like a spider web), or functions of biological  
18 traits (like the pumping function of the heart). Since the latter functions  
19 are acquired and possessed by objects without the intentional intervention  
20 of any observers, having a function is not intentionality-relative.

21 Sometimes, Searle blurs the distinction between being relative to a user  
22 and being relative to an observer: for instance, in one place he writes that  
23 observer-relative facts are those that “exist relative to the intentionality of  
24 observers, users, etc.” (Searle, 1995, p. 9). But the same facts may depend  
25 on users and observers in very different ways. A fact may depend on the  
26 intentionality of observers because it's a matter of opinion or taste (e.g.  
27 whether lemon tastes good); such a fact does not depend on there being  
28 any users of anything. Another fact may depend on the existence of users  
29 (which may be non-human animals, robots, or zombies) because it is  
30 about a property that relates objects to users; such a fact need not depend  
31 on the intentionality of observers (cf. de Ridder, 2006, p. 89). Whether an  
32 artifact has a function may well depend on the intentionality of users;  
33 once that is disentangled from the intentionality of observers, however,  
34 there remains no reason to deny that having such a function is an objective  
35 property of the artifact.

36 Third, Searle blurs the distinction between having a function and func-  
37 tioning as. They are not the same. For instance, both a screwdriver and a  
38 knife in working order may be used to either drive screws or make cuts.  
39 In the former case, they function as screwdrivers; in the second, as knives.  
40 From this, it doesn't follow that which is the screwdriver and which is the  
41 knife is an intentionality-relative affair, except in the trivial sense that  
42 without people to make them, there would be neither knives nor screw-  
43 drivers. But both a screwdriver and a knife may be damaged (e.g. by  
44 being bent) to the point of being unable to drive screws or make cuts. If

1 so, they cannot function as either screwdrivers or knives. Nevertheless,  
2 they retain their respective functions.<sup>36</sup> This distinction is relevant here. I  
3 do not claim that computing mechanisms always function as such; I claim  
4 that they have the function of computing.

5 Finally, I am relying on functions in the sense in which both artifacts  
6 and biological systems may possess functions. In this familiar sense,  
7 hearts are pumps, kidneys and livers are filters, and bones, joints, and  
8 muscles are arranged together to form lever systems. As a long tradition  
9 in philosophy of science acknowledges, this notion of function is at the  
10 heart of many special sciences, including neuroscience and computer  
11 science. Although there used to be philosophers who regarded functional  
12 descriptions with suspicion, there is now consensus that functional  
13 descriptions in biology and engineering are bona fide scientific descriptions.  
14 Even Searle, in spite of his insistence that functional descriptions are  
15 intentionality-relative, eventually admits that they are “objective” and  
16 discoverable by science (Searle, 1995, p. 15). This conclusion directly  
17 contradicts his earlier argument to the effect that computational descriptions  
18 cannot be discovered and characterized by science because they are  
19 functional (Searle, 1992, p. 212).

20 There is no consensus on how to account for functional descriptions  
21 within the special sciences. According to one family of proposals, systems  
22 have functions in virtue of their evolutionary (or perhaps more generally,  
23 reproductive) history (e.g. Millikan, 1993; Neander, 1991; Preston, 2003).  
24 According to another family of proposals, functions are capacities that  
25 contribute to the capacities of a system (e.g. Cummins, 2000). There are  
26 other, less prominent accounts of functions, but there is no room here to  
27 address the extensive literature on this topic.<sup>37</sup> I favor accounts of functions  
28 that see them as properties of mechanisms and accommodate them within  
29 mechanistic explanations (Craver, 2001; Wimsatt, 2002; de Ridder, 2006).  
30 But any account that provides an objective notion of function will do, at  
31 least to a first degree of approximation.<sup>38</sup>

## 32 33 34 **8. Conclusion**

35  
36 Contrary to what some authors maintain (e.g. Churchland and Sejnowski,  
37 1992; Searle, 1992), whether something is a computer, and what kind of  
38 computer it is, is an objective feature of a system. Computers are calculators  
39 of large capacity, whose function is to perform computations that involve  
40 long sequences of primitive operations on strings of digits, operations  
41 that can be performed automatically by the computers’ processors.  
42 Calculators are a special kind of computing mechanism. Computing  
43 mechanisms, in turn, are distinct from other systems in that they manipulate  
44 strings of digits according to appropriate rules.

1 Different classes of computers can be programmed in different ways or  
2 compute different classes of functions. These and other useful distinctions  
3 between classes of computers can be drawn by looking at computers' mecha-  
4 nistic properties, and they can be profitably used in historical and philosophical  
5 discussions pertaining to computers and other computing mechanisms.

6 This mechanistic account of computers has several advantages. First, it  
7 underwrites our intuitive distinctions between systems that compute and  
8 systems that don't as well as between computers and other computing  
9 mechanisms, such as calculators. Second, it explains the versatility of  
10 computers in terms of their functional organization. Third, it sheds light on  
11 why computers, not calculators or other computing mechanisms, inspired  
12 the computational theory of mind and brain. Fourth, it explicates the notion  
13 of explanation by program execution, i.e. an explanation of a system's  
14 capacity by postulating the execution of a program for that capacity.

15 Explanations by program execution are invoked in the philosophy of  
16 mind literature (cf. Piccinini, 2004c). Given the mechanistic account  
17 of computers, explanations by program execution are a special kind of  
18 mechanistic explanation that applies to soft-programmable computers.  
19 Soft-programmable computers are computers with processors that respond  
20 differentially to different strings of digits, to the effect that different  
21 operations are performed on data. Within a computer, program execution  
22 is a process by which (a stable state of) a certain part of the mechanism,  
23 the program, affects another part of the mechanism, the processor, so that  
24 the processor performs appropriate operations on (a stable state of yet)  
25 another part of the mechanism, the data. Only mechanisms with the relev-  
26 ant mechanistic properties are subject to explanation by program execu-  
27 tion. By identifying more precisely the class of computers that support  
28 explanation by program execution and how they do so, the mechanistic  
29 account of computers makes explicit the commitments of those who  
30 appeal to explanation by program execution in the philosophy of mind.

31 Finally, the present account of computers can be used to formulate a  
32 rigorous taxonomy of computationalist theses about the brain, which  
33 makes explicit their empirical commitments to specific functional prop-  
34 erties of brains, and to compare the strength of the different empirical  
35 commitments of different computationalist theses. This makes it ideal to  
36 ground discussions of computational theories of mind and brain.<sup>39</sup>

37  
38 Department of Philosophy  
39 University of Missouri – St. Louis  
40

41  
42 NOTES

43 <sup>1</sup> I borrow the phrase 'large capacity' from John V. Atanasoff, who was apparently the first  
44 person to use the term 'computer' for a kind of machine (Atanasoff, 1940; Atanasoff, 1984).

1     <sup>2</sup> This section draws from Piccinini forthcoming a, which contains a general formulation  
2 and defense of my account of computing mechanisms. This paper is a natural sequel to  
3 that. Recent work on mechanistic explanation in general includes Bechtel, 2007; Craver,  
4 2007; Darden, 2006; Glennan, 2002, and Wimsatt, 2007.

5     <sup>3</sup> The notion of string of digit in question is nothing but a concrete counterpart to the  
6 mathematical notion of string of letter (or “symbols”). For a technical treatment of the  
7 theory of strings, see Corcoran *et al.* 1974. The present notion of computation over strings  
8 is a generalization of the rigorous notion of computation of functions of natural numbers,  
9 which derives from work by Alan Turing (Turing 1936–7) and other logicians. The present  
10 notion is relevant to the explication of what are ordinarily called *digital* computers and  
11 computing mechanisms. In section 3.5 below, I will discuss so called *analog* computers and  
12 argue that they must be analyzed in terms of a different notion of computation.

13     <sup>4</sup> For more details on programmable calculators, and a statement that they are a kind  
14 of computer, see Engelsohn, 1978.

15     <sup>5</sup> A consequence of this is the often-remarked fact that calculators cannot perform  
16 branches; namely, they cannot choose among different operations depending on whether  
17 some condition obtains. (Branching is necessary to compute all computable functions.)

18     <sup>6</sup> Atanasoff, 1940. On the ABC, see also Burks and Burks, 1988; Burks, 2002, and  
19 Gustafson, 2000.

20     <sup>7</sup> A similar proposal is made by Burks and Burks, 1988, ch. 5.

21     <sup>8</sup> The present distinction between programmability insensitive to the input and pro-  
22 grammability sensitive to the input was partially inspired by a somewhat similar distinction,  
23 made by Brennecke, 2000, pp. 62–64, between executing a fixed sequence and using the  
24 output as input.

25     <sup>9</sup> On the ENIAC, see Van der Spiegel *et al.* 2000.

26     <sup>10</sup> This point is effectively made by Brennecke, 2000, pp. 64–65.

27     <sup>11</sup> On the Harvard Mark I, see Cohen, 1999.

28     <sup>12</sup> Cf. Brennecke, 2000, p. 66.

29     <sup>13</sup> Alan Turing is one of the few people who discuss this feature of internally soft-pro-  
30 grammable computers; he used it in his reply to the mathematical objection to the view  
31 that machines can think. For an extended discussion, see Piccinini, 2003a.

32     <sup>14</sup> In their authoritative textbook on computer organization, Patterson and Hennessy  
33 list these two ideas as the essence of modern computers (Patterson and Hennessy, 1998,  
34 p. 121).

35     <sup>15</sup> For an overview of early computers, see Rojas and Hashagen, 2000.

36     <sup>16</sup> For a valuable attempt at distinguishing between degrees to which a computer is  
37 general-purpose, see Bromley 1983.

38     <sup>17</sup> For an introduction to computability theory, see Davis *et al.*, 1994.

39     <sup>18</sup> Strictly speaking, branching is not necessary for computational universality, but the  
40 alternative is too impractical to be relevant (Rojas, 1998).

41     <sup>19</sup> Thanks to Doron Swade for this point.

42     <sup>20</sup> For more details on virtual memory, including its many advantages, see Patterson and  
43 Hennessy, 1998, pp. 579–602.

44     <sup>21</sup> Thanks to an anonymous referee for making this point.

<sup>22</sup> Cf.: “The input to a neuron is analog (continuous values between 0 and 1)” (Churchland  
and Sejnowski, 1992, p. 51).

<sup>23</sup> By contrast, when I characterized digits as discrete, I implicitly assumed that the  
appropriate level of analysis was determined by the system’s mechanistic explanation.

<sup>24</sup> Authoritative works on analog computers, which I have used as sources for the following  
remarks, include Jackson, 1960; Johnson, 1963; Korn and Korn, 1972, and Wilkins, 1970.

1       <sup>25</sup> I am here ignoring the possibility, for now quite speculative, of building hypercomputers  
2 (cf. Copeland, 2000).

3       <sup>26</sup> For a step in this direction, see Rubel, 1993.

4       <sup>27</sup> Connectionist systems and the way some of them compute deserve their own account  
5 (Piccinini, forthcoming c).

6       <sup>28</sup> For more details on pipelining, see Patterson and Hennessy 1998, chap. 6.

7       <sup>29</sup> For an extended argument that computers have important features lacked by Turing  
8 machines, see Sloman, 2002. Some of the features listed by Sloman are made possible, in  
9 part, by the functional distinction between memory, input devices, and output devices. For  
10 some friendly amendments to Sloman's argument, see Piccinini, 2005.

11       <sup>30</sup> This thesis should not be confused with the uncontroversial fact that human beings  
12 are programmable in the sense of being able to follow different algorithms.

13       <sup>31</sup> For a detailed analysis of McCulloch and Pitts's theory, see Piccinini, 2004b.

14       <sup>32</sup> For a more extended discussion of von Neumann's theory of the brain, see Piccinini  
15 2003b.

16       <sup>33</sup> A related view may be found in the work of a few physicists, who see every physical  
17 process as a computation (e.g. Toffoli, 1984; Wolfram, 2002). A variant of this version of  
18 pancomputationalism is the view that everything is either a computer or a hypercomputer,  
19 where a hypercomputer is a machine computationally more powerful than a universal  
20 Turing machine (Copeland, 2000).

21       <sup>34</sup> Craver (2006) offers a general account of the distinction between mere models and  
22 explanatory models that supports my approach. According to Craver, explanatory models,  
23 unlike mere models, describe mechanisms for the explananda. My approach aligns with  
24 Craver's account in the following way. A computational model in my sense may or may not  
25 be an explanatory model in Craver's sense. But in general, even when a computational  
26 model is explanatory, it need not provide a computational explanation. Computational  
27 explanation is the special type of mechanistic explanation that applies to computing  
28 mechanisms.

29       <sup>35</sup> Searle formulates his point about observer-relativity of computational properties in  
30 terms of syntactic properties and adds that computational properties are syntactic.  
31 Unfortunately, Searle doesn't say what he means by 'syntactic'. I think that thanks to Alan  
32 Turing and others, we have a better understanding of 'computational' than 'syntactic'. I  
33 also doubt that all computational properties are syntactic (in the usual senses of those  
34 terms). To avoid getting entangled in an irrelevant dispute about syntactic properties, I  
35 have reformulated Searle's point directly in terms of computational properties.

36       <sup>36</sup> Elsewhere, Searle seems to agree. Cf. Searle, 1995, p. 45. For discussion of this  
37 point and related criticisms of Searle's view of functional properties, see Kroes, 2003 and  
38 Miller, 2005.

39       <sup>37</sup> Two useful collections are Allen, Bekoff, and Lauder, 1998, and Ariew, Cummins, and  
40 Perlman, 2002. More recent contributions include Perlman, 2004; Hourkes and Vermaas,  
41 2004; Cameron, 2004; Johansson, 2004; Schroeder, 2004; Vermaas and Houkes, 2006;  
42 Scheele, 2006; Franssen, 2006; Vermaas, 2006; Houkes, 2006; Houkes and Meijers, 2006;  
43 Kroes, 2006.

44       <sup>38</sup> Anyone who remains skeptical about functions need not reject the present account.  
She may attempt to reformulate it in terms of activities, or capacities, or dispositions, or  
whatever suitable notion she finds metaphysically kosher.

<sup>39</sup> Thanks to Peter Machamer, Eric Thomson, and the anonymous referees for comments  
on previous drafts. Thanks also to those who discussed computers with me – especially Eli  
Dresner, Jonathan Mills, and Doron Swade. Material that led to this paper was presented  
at the Canadian Society for the History and Philosophy of Science, Toronto, May 2002,

1 Computing and Philosophy (CAP@CMU), Pittsburgh, PA, August 2002, and the University  
2 of Pittsburgh in January 2003. Thanks to the audiences for their feedback. The writing of  
3 this paper was supported in part by a Research Grant from the University of Missouri.  
4 Thanks to Jeff Dauer for editorial assistance.

## REFERENCES

- 7 Allen, C., Bekoff, M. and Lauder, G., eds (1998). *Nature's Purposes: Analysis of Function*  
8 *and Design in Biology*. Cambridge, MA: MIT Press.
- 9 Ariew, A., Cummins, R. and Perلمان, M., eds (2002). *Functions: New Essays in the*  
10 *Philosophy of Psychology and Biology*. Oxford: Oxford University Press.
- 11 Aspray, W. (1990). *John von Neumann and the Origins of Modern Computing*. Cambridge,  
12 MA: MIT Press.
- 13 Atanasoff, J. V. (1940). "Computing Machine for the Solution of Large Systems of Linear  
14 Algebraic Equations," Ames, IA: Iowa State College.
- 15 Atanasoff, J. V. (1984). "Advent of Electronic Digital Computing," *Annals of the History of*  
16 *Computing* 6, pp. 229–282.
- 17 Bechtel, W. (2007). *Mental Mechanisms: Philosophical Perspectives on the Sciences*  
18 *of Cognition and the Brain*. London: Routledge.
- 19 Blanchowicz, J. (1997). "Analog Representation beyond Mental Imagery," *The Journal of*  
20 *Philosophy* 94, pp. 55–84.
- 21 Bontly, T. (1998). "Individualism and the Nature of Syntactic States," *British Journal for*  
22 *the Philosophy of Science* 49, pp. 557–574.
- 23 Boshernitzan, M. (1986). "Universal Formulae and Universal Differential Equations," *The*  
24 *Annals of Mathematics, 2nd Series* 124(2), pp. 273–291.
- 25 Brennecke, A. (2000). "Hardware Components and Computer Design," in R. Rojas and U.  
26 Hashagen (eds) *The First Computers-History and Architectures*. Cambridge, MA: MIT  
27 Press, pp. 53–68.
- 28 Bromley, A. G. (1983). "What Defines a 'General-Purpose' Computer?," *Annals of the History*  
29 *of Computing* 5, pp. 303–305.
- 30 Burks, A. R. (2002). *Who Invented the Computer?* Amherst, MA: Prometheus.
- 31 Burks, A. R. and Burks, A. W. (1988). *The First Electronic Computer: The Atanasoff Story*.  
32 Ann Arbor: University of Michigan Press.
- 33 Cameron, R. (2004). "How to Be a Realist About Sui Generis Teleology Yet Feel at Home  
34 in the 21st Century," *The Monist* 87, pp. 72–95.
- 35 Chalmers, D. (1994). "On Implementing a Computation," *Minds and Machines* 4, pp. 391–402.
- 36 Chalmers, D. J. (1996). "Does a Rock Implement Every Finite-State Automaton?,"  
37 *Synthese* 108, pp. 310–333.
- 38 Chrisley, R. L. (1995). "Why Everything Doesn't Realize Every Computation," *Minds and*  
39 *Machines* 4, pp. 403–430.
- 40 Churchland, P. S., Koch, C. and Sejnowski, T. J. (1990). "What is Computational Neuro-  
41 science?," in E. L. Schwartz (ed.) *Computational Neuroscience*. Cambridge, MA: MIT Press,  
42 pp. 46–55.
- 43 Churchland, P. S. and Sejnowski, T. J. (1992). *The Computational Brain*. Cambridge, MA:  
44 MIT Press.
- Cohen, I. B. (1999). *Howard Aiken: Portrait of a Computer Pioneer*. Cambridge, MA: MIT  
Press.
- Copeland, B. J. (1996). "What is Computation?," *Synthese* 108, pp. 224–359.
- Copeland, B. J. (2000). "Narrow Versus Wide Mechanism: Including a Re-Examination of  
Turing's Views on the Mind-Machine Issue," *The Journal of Philosophy* XCVI, pp. 5–32.

- 1 Corcoran, J., Frank, W. and Maloney, M. (1974). "String Theory," *The Journal of Symbolic*  
 2 *Logic* 39, pp. 625–637.
- 3 Craver, C. F. (2001). "Role Functions, Mechanisms, and Hierarchy," *Philosophy of Science*  
 4 68, pp. 53–74.
- 5 Craver, C. F. (2006). "When Mechanistic Models Explain," *Synthese* 153, pp. 355–376.
- 6 Craver, C. F. (2007). *Explaining the Brain*. Oxford: Oxford University Press.
- 7 Cummins, R. (1989). *Meaning and Mental Representation*. Cambridge, MA: MIT Press.
- 8 Cummins, R. (2000). "How does it work?" vs. "What are the laws?": Two Conceptions of  
 9 Psychological Explanation," in F. C. Keil and R. A. Wilson (eds) *Explanation and*  
 10 *Cognition*. Cambridge, MA: MIT Press.
- 11 Darden, L. (2006). *Reasoning in Biological Discoveries*. New York: Cambridge University Press.
- 12 Dardis, A. (1993). "Comment on Searle: Philosophy and the Empirical Study of Conscious-
- 13 ness,"
- Consciousness and Cognition*
- 2, pp. 320–333.
- 14 Davis, M., Sigal, R. and Weyuker, E. J. (1994). *Computability, Complexity, and Languages*.  
 15 Boston: Academic.
- 16 Dennett, D. C. (1991). *Consciousness Explained*. Boston, MA: Little, Brown.
- 17 de Ridder, J. (2006). "Mechanistic Artefact Explanation," *Studies in History and Philosophy of*  
 18 *Science* 37, pp. 81–96.
- 19 Devitt, M. and Sterelny, K. (1999). *Language and Reality: An Introduction to the Philosophy*  
 20 *of Language*. Cambridge, MA: MIT Press.
- 21 Dresner, E. (2003). "Effective Memory' and Turing's Model of Mind," *Journal of*  
 22 *Experimental and Theoretical Artificial Intelligence* 15, pp. 113–123.
- 23 Duffin, R. J. (1981). "Rubel's Universal Differential Equation," *Proceedings of the National*  
 24 *Academy of Sciences USA* 78(8 [Part 1: Physical Sciences]), pp. 4661–4662.
- 25 Egan, F. (1995). "Computation and Content," *Philosophical Review* 104, pp. 181–203.
- 26 Eliasmith, C. (2003). "Moving Beyond Metaphors: Understanding the Mind for What It  
 27 Is," *Journal of Philosophy* C(10), pp. 493–520.
- 28 Engelsohn, H. S. (1978). *Programming Programmable Calculators*. Rochelle Park, NJ:  
 29 Hayden.
- 30 Fodor, J. A. (1975). *The Language of Thought*. Cambridge, MA: Harvard University Press.
- 31 Fodor, J. A. (1990). *A Theory of Content and Other Essays*. Cambridge, MA: MIT Press.
- 32 Franssen, M. (2006). "The Normativity of Artefacts," *Studies in History and Philosophy of*  
 33 *Science* 37, pp. 42–57.
- 34 Glennan, S. S. (2002). "Rethinking Mechanistic Explanation," *Philosophy of Science* 64,  
 35 pp. 605–206.
- 36 Goodman, N. (1968). *Languages of Art*. Indianapolis: Bobbs-Merrill.
- 37 Gustafson, J. (2000). "Reconstruction of the Atanasoff-Berry Computer," in R. Rojas and  
 38 U. Hashagen (eds) *The First Computers-History and Architectures*. Cambridge, MA:  
 39 MIT Press, pp. 91–106.
- 40 Haugeland, J. (1981). "Analog and Analog," *Philosophical Topics* 12, pp. 213–225.
- 41 Houkes, W. (2006). "Knowledge of Artefact Functions," *Studies in History and Philosophy*  
 42 *of Science* 37, pp. 102–113.
- 43 Houkes, W. and Meijers, A. (2006). "The Ontology of Artefacts: The Hard Problem,"  
 44 *Studies in History and Philosophy of Science* 37, pp. 118–131.
- Houkes, W. and Vermaas, P. (2004). "Actions versus Functions: A Plea for an Alternative  
 Metaphysics of Artifacts," *The Monist* 87, pp. 52–71.
- Hughes, R. I. G. (1999). "The Ising Model, Computer Simulation, and Universal Physics,"  
 in M. S. Morgan and M. Morrison (eds) *Models as Mediators*. Cambridge: Cambridge  
 University Press, pp. 97–145.
- Jackson, A. S. (1960). *Analog Computation*. New York: McGraw-Hill.

- 1 Johansson, I. (2004). "Functions, Function Concepts, and Scales," *The Monist* 87,  
2 pp. 96–114.
- 3 Johnson, C. L. (1963). *Analog Computer Techniques, Second Edition*. New York: McGraw-Hill.
- 4 Korn, G. A. and Korn, T. M. (1972). *Electronic Analog and Hybrid Computers; 2nd*  
5 *Completely Revised Edition*. New York: McGraw-Hill.
- 6 Kroes, P. (2003). "Screwdriver Philosophy: Searle's Analysis of Technical Functions,"  
7 *Techné* 6, pp. 21–34.
- 8 Kroes, P. (2006). "Coherence of Structural and Functional Descriptions of Technical  
9 Artefacts," *Studies in History and Philosophy of Science* 37, pp. 137–151.
- 10 Lewis, D. K. (1971). "Analog and Digital," *Noûs* 5, pp. 321–327.
- 11 Lipshitz, L. and Rubel, L. A. (1987). "A Differentially Algebraic Replacement Theorem,  
12 and Analog Computability," *Proceedings of the American Mathematical Society* 99(2),  
13 pp. 367–372.
- 14 Ludlow, P. (2003). "Externalism, Logical Form, and Linguistic Intentions," in A. Barber  
15 (ed.) *Epistemology of Language*. Oxford: Oxford University Press.
- 16 McCulloch, W. S. and Pitts, W. H. (1943). "A Logical Calculus of the Ideas Immanent in  
17 Nervous Activity," *Bulletin of Mathematical Biophysics* 7, pp. 115–133.
- 18 Miller, S. (2005). "Artefacts and Collective Intentionality," *Techné* 9: 52–67.
- 19 Millikan, R. G. (1993). *White Queen Psychology and Other Essays for Alice*. Cambridge,  
20 MA: MIT Press.
- 21 Neander, K. (1991). "Functions as Selected Effects: The Conceptual Analyst's Defence,"  
22 *Philosophy of Science* 58, pp. 168–184.
- 23 Nelson, R. J. (1982). *The Logic of Mind*. Dordrecht: Reidel.
- 24 Newell, A. and Simon, H. A. (1976). "Computer Science as an Empirical Enquiry: Symbols  
25 and Search," *Communications of the ACM* 19, pp. 113–126.
- 26 Patterson, D. A. and Hennessy, J. L. (1998). *Computer Organization and Design: The Hardware*  
27 *Software Interface*. San Francisco: Morgan Kauffman.
- 28 Perlman, M. (2004). "The Modern Philosophical Resurrection of Teleology," *The Monist*  
29 87, pp. 3–51.
- 30 Piccinini, G. (2003a). "Alan Turing and the Mathematical Objection," *Minds and Machines*  
31 13, pp. 23–48.
- 32 Piccinini, G. (2003b). "Review of John von Neumann's *The Computer and the Brain*,"  
33 *Minds and Machines* 13, pp. 327–332.
- 34 Piccinini, G. (2004a). "Functionalism, Computationalism, and Mental Contents,"  
35 *Canadian Journal of Philosophy* 34, pp. 375–410.
- 36 Piccinini, G. (2004b). "The First Computational Theory of Mind and Brain: A Close Look  
37 at McCulloch and Pitts's 'Logical Calculus of Ideas Immanent in Nervous Activity'," *Synthese*  
38 141, pp. 175–215.
- 39 Piccinini, G. (2004c). "Functionalism, Computationalism, and Mental States," *Studies in*  
40 *the History and Philosophy of Science* 35, pp. 811–833.
- 41 Piccinini, G. (2005). "Review of M. Scheutz's *Computationalism: New Directions*,"  
42 *Philosophical Psychology* 18, pp. 387–391.
- 43 Piccinini, G. (2007). "Computational Modeling vs. Computational Explanation: Is Everything  
44 a Turing Machine, and Does It Matter to the Philosophy of Mind?" *Australasian Journal*  
45 *of Philosophy* 85, pp. 93–115.
- 46 Piccinini, G. (forthcoming a). "Computing Mechanisms," *Philosophy of Science*.
- 47 Piccinini, G. (forthcoming b). "Computation without Representation," *Philosophical*  
48 *Studies*.
- 49 Piccinini, G. (forthcoming c). "Some Neural Networks Compute, Others Don't," *Neural*  
50 *Networks*.

- 1 Pour-El, M. B. (1974). "Abstract Computability and Its Relation to the General Purpose  
2 Analog Computer (Some Connections between Logic, Differential Equations and  
3 Analog Computers)," *Transactions of the American Mathematical Society* 199, pp. 1–  
4 28.
- 5 Preston, B. (2003). "Of Marigold Beer: A Reply to Vermaas and Houkes," *British Journal  
6 for the Philosophy of Science* 54, pp. 601–612.
- 7 Putnam, H. (1988). *Representation and Reality*. Cambridge, MA: MIT Press.
- 8 Pylyshyn, Z. W. (1984). *Computation and Cognition*. Cambridge, MA: MIT Press.
- 9 Rojas, R. (1998). "How to Make Zuse's Z3 a Universal Computer," *IEEE Annals of the  
10 History of Computing* 20(3), pp. 51–54.
- 11 Rojas, R. and Hashagen, U. eds (2000). *The First Computers-History and Architectures*.  
12 Cambridge, MA: MIT Press.
- 13 Roth, M. (2005). "Program Execution in Connectionist Networks," *Mind & Language* 20,  
14 pp. 448–467.
- 15 Rubel, L. A. (1989). "Digital Simulation of Analog Computation and Church's Thesis,"  
16 *Journal of Symbolic Logic* 54, pp. 1011–1017.
- 17 Rubel, L. A. (1993). "The Extended Analog Computer," *Advances in Applied Mathematics*  
18 14, pp. 39–50.
- 19 Rubel, L. A. and Singer, M. F. (1985). "A Differentially Algebraic Elimination Theorem  
20 with Application to Analog Computability in the Calculus of Variations," *Proceedings  
21 of the American Mathematical Society* 94(4), pp. 653–658.
- 22 Scheele, M. (2006). "Function and Use of Artefacts: Social Conditions of Function  
23 Ascription," *Studies in History and Philosophy of Science* 37, pp. 23–36.
- 24 Scheutz, M. (2001). "Causal versus Computational Complexity," *Minds and Machines* 11,  
25 pp. 534–566.
- 26 Schroeder, T. (2004). "Functions from Regulation," *The Monist* 87, pp. 115–135.
- 27 Searle, J. R. (1992). *The Rediscovery of the Mind*. Cambridge, MA: MIT Press.
- 28 Searle, J. R. (1995). *The Construction of Social Reality*. New York: The Free Press.
- 29 Shagrir, O. (2006). "Why We View the Brain as a Computer," *Synthese* 153, pp. 393–416.
- 30 Shannon, C. E. (1941). "Mathematical Theory of the Differential Analyzer," *Journal of  
31 Mathematics and Physics* XX, pp. 337–354.
- 32 Siegelmann, H. T. (1999). *Neural Networks and Analog Computation: Beyond the Turing  
33 Limit*. Boston, MA: Birkhäuser.
- 34 Siegelmann, H. T. (2003). "Neural and Super-Turing Computing," *Minds and Machines* 13,  
35 pp. 103–114.
- 36 Sloman, A. (2002). "The Irrelevance of Turing Machines to Artificial Intelligence," in  
37 M. Scheutz (ed.) *Computationalism: New Directions*. Cambridge, MA: MIT Press,  
38 pp. 87–127.
- 39 Smith, B. C. (2002). "The Foundations of Computing," in M. Scheutz (ed.) *Computationalism:  
40 New Directions*. Cambridge, MA: MIT Press, pp. 23–58.
- 41 Stich, S. (1983). *From Folk Psychology to Cognitive Science*. Cambridge, MA: MIT Press.
- 42 Toffoli, T. (1984). "Cellular Automata as an Alternative to (rather than an Approximation  
43 of) Differential Equations in Modeling Physics," *Physica* 10D, pp. 117–127.
- 44 Turing, A. M. (1936–7) [2004]. "On computable numbers, with an application to the  
Entscheidungsproblem," in M. Davis (ed.) *The Undecidable*. Dover: Mineola.
- Turing, A. M. (1945) [2005]. "Notes on Memory," in B. J. Copeland (ed.) *Alan Turing's  
Automatic Computing Engine*. Oxford: Oxford University Press, pp. 456–457.
- Van der Spiegel, J., Tau, J. F., Alailima, T and Ang, L. P. (2000). "The ENIAC: History,  
Operation and Reconstruction in VSLI," in R. Rojas and U. Hashagen (eds) *The First  
Computers-History and Architectures*. Cambridge, MA: MIT Press, pp. 121–178.

- 1 Vermaas, P. E. (2006). "The Physical Connection: Engineering Function Ascription to  
2 Technical Artefacts and their Components," *Studies in History and Philosophy of  
3 Science* 37, pp. 62–75.
- 4 Vermaas, P. E. and Houkes, W. (2006). "Technical Functions: A Drawbridge between the  
5 Intentional and Structural Natures of Technical Artefacts," *Studies in History and  
6 Philosophy of Science* 37, pp. 5–18.
- 7 von Neumann, J. (1945). First Draft of a Report on the EDVAC. Philadelphia, PA: Moore  
8 School of Electrical Engineering, University of Pennsylvania.
- 9 von Neumann, J. (1958). *The Computer and the Brain*. New Haven, CT: Yale University  
10 Press.
- 11 Wilkins, B. R. (1970). *Analogue and Iterative Methods in Computation, Simulation, and  
12 Control*. London: Chapman and Hall.
- 13 Wimsatt, W. C. (2002). "Functional Organization, Analogy, and Inference," in A. Ariew,  
14 R. Cummins and M. Perlman (eds) *Functions: New Essays in the Philosophy of Psychology  
15 and Biology*. Oxford: Oxford University Press, pp. 173–221.
- 16 Wimsatt, W. (2007). *Re-Engineering Philosophy for Limited Beings: Piecewise Approximations  
17 to Reality*. Cambridge, MA: Harvard University Press.
- 18 Wolfram, S. (2002). *A New Kind of Science*. Champaign, IL: Wolfram Media.
- 19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44

# MARKED PROOF

## Please correct and return this set

Please use the proof correction marks shown below for all alterations and corrections. If you wish to return your proof by fax you should ensure that all amendments are written clearly in dark ink and are made well within the page margins.

<i>Instruction to printer</i>	<i>Textual mark</i>	<i>Marginal mark</i>
Leave unchanged	... under matter to remain	Ⓟ
Insert in text the matter indicated in the margin	∧	New matter followed by ∧ or ∧ <sup>Ⓢ</sup>
Delete	/ through single character, rule or underline or ┌───┐ through all characters to be deleted	Ⓞ or Ⓞ <sup>Ⓢ</sup>
Substitute character or substitute part of one or more word(s)	/ through letter or ┌───┐ through characters	new character / or new characters /
Change to italics	— under matter to be changed	↙
Change to capitals	≡ under matter to be changed	≡
Change to small capitals	≡ under matter to be changed	≡
Change to bold type	~ under matter to be changed	~
Change to bold italic	≈ under matter to be changed	≈
Change to lower case	Encircle matter to be changed	≡
Change italic to upright type	(As above)	⊕
Change bold to non-bold type	(As above)	⊖
Insert 'superior' character	/ through character or ∧ where required	Υ or Υ under character e.g. Υ or Υ
Insert 'inferior' character	(As above)	∧ over character e.g. ∧
Insert full stop	(As above)	⊙
Insert comma	(As above)	,
Insert single quotation marks	(As above)	ʹ or ʸ and/or ʹ or ʸ
Insert double quotation marks	(As above)	“ or ” and/or ” or ”
Insert hyphen	(As above)	⊥
Start new paragraph	┌	┌
No new paragraph	┐	┐
Transpose	└┐	└┐
Close up	linking ○ characters	○
Insert or substitute space between characters or words	/ through character or ∧ where required	Υ
Reduce space between characters or words		↑