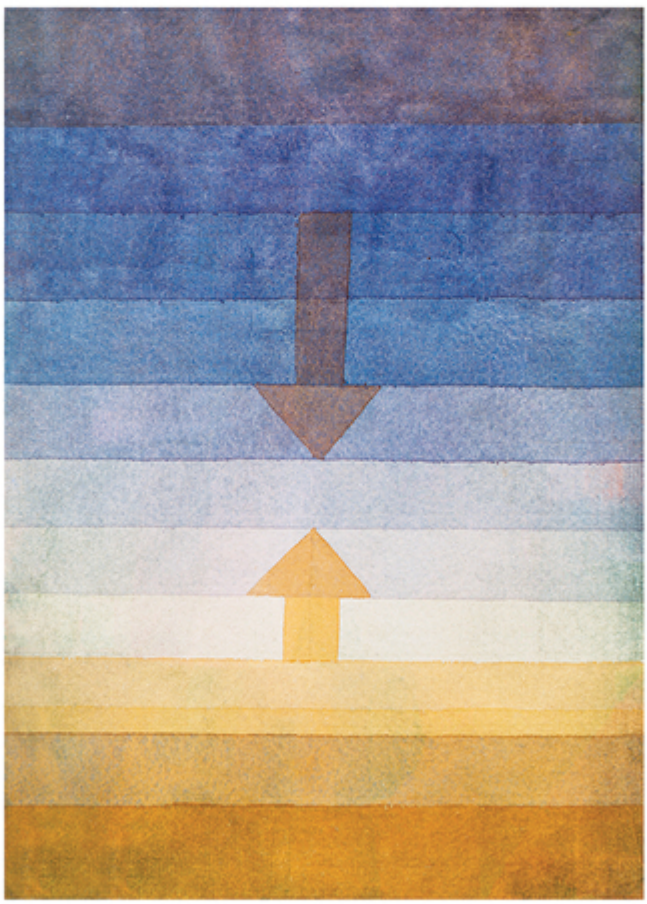
OXFORD

PHYSICAL COMPUTATION

A Mechanistic Account



GUALTIERO PICCININI

Physical Computation

Physical Computation

A Mechanistic Account

Gualtiero Piccinini

OXFORD
UNIVERSITY PRESS

OXFORD
UNIVERSITY PRESS

Great Clarendon Street, Oxford, OX2 6DP,
United Kingdom

Oxford University Press is a department of the University of Oxford.
It furthers the University's objective of excellence in research, scholarship,
and education by publishing worldwide. Oxford is a registered trade mark of
Oxford University Press in the UK and in certain other countries

© Gualtiero Piccinini 2015

The moral rights of the author have been asserted

First Edition published in 2015

Impression: 1

All rights reserved. No part of this publication may be reproduced, stored in
a retrieval system, or transmitted, in any form or by any means, without the
prior permission in writing of Oxford University Press, or as expressly permitted
by law, by licence or under terms agreed with the appropriate reprographics
rights organization. Enquiries concerning reproduction outside the scope of the
above should be sent to the Rights Department, Oxford University Press, at the
address above

You must not circulate this work in any other form
and you must impose this same condition on any acquirer

Published in the United States of America by Oxford University Press
198 Madison Avenue, New York, NY 10016, United States of America

British Library Cataloguing in Publication Data
Data available

Library of Congress Control Number: 2014957194

ISBN 978–0–19–965885–5

Printed and bound by
CPI Group (UK) Ltd, Croydon, CR0 4YY

Links to third party websites are provided by Oxford in good faith and
for information only. Oxford disclaims any responsibility for the materials
contained in any third party website referenced in this work.

Contents

Acknowledgements

I encountered the debate about the foundations of cognitive science when, as an undergraduate at the University of Turin, I took an introductory course in cognitive psychology taught by Bruno Bara. Those early wonderings eventually led to my Ph.D. dissertation at the University of Pittsburgh, and then to a stream of papers on computation and related topics. This book organizes, streamlines, revises, and systematizes most of what I've thought and published about concrete computation. (It leaves out my work on computational theories of cognition.)

I could not have gotten where I am without the help of my teachers, beginning with my parents and continuing all the way to my college and graduate school teachers. In college, the following were especially influential: in philosophy, Elisabetta Galeotti, Diego Marconi, Gianni Vattimo, and Carlo Augusto Viano; in psychology, Bruno Bara and Giuliano Geminiani. In graduate school, the following played important roles in my thinking about computation: my advisor Peter Machamer as well as Nuel Belnap, Robert Daley (Computability Theory), Bard Ermentrout (Computational Neuroscience), John Earman, Clark Glymour, Paul Griffiths, Rick Grush, Ken Manders, John Norton, and Merrilee Salmon. Other important teachers during graduate school were Robert Brandom, Joe Camp, Adolf Grünbaum, Susie Johnson (Theory of Mind), Pat Levitt (Systems Neuroscience), Jay McClelland (Cognitive Neuroscience), John McDowell, Ted McGuire, Robert Olby, Carl Olson (Cognitive Neuroscience), Fritz Ringer, Laura Ruetsche, and Richmond Thomason.

Many people's work shaped mine. On computation, the classic work of Alonzo Church, Kurt Gödel, Stephen Kleene, Emil Post, and Alan Turing was paramount. On neural networks and their connection to digital computers, I learned the most from Warren McCulloch, Walter Pitts, John von Neumann, and Norbert Wiener. On analog computers, the work of Claude Shannon, Marian Pour-El, and Lee Rubel was crucial. On artificial intelligence and computational cognitive science, Allen Newell and Herbert Simon had the largest impact. The canonical textbook on computer organization and design by David Patterson and John Hennessy confirmed my early hunch that computational explanation is mechanistic and helped me articulate it. On the philosophy of computation, I found most helpful the works of Jack Copeland, Robert Cummins, Daniel Dennett, Frances Egan, Jerry Fodor, Justin Garson, Gilbert Harman, John Haugeland, Zenon Pylyshyn, Oron Shagrir, Wilfried Sieg, and Stephen Stich. On information, Fred Drestke and Claude Shannon were most significant. On functions, Christopher Boorse and Ruth Millikan made the biggest difference. On mechanisms, I was most influenced by Carl Craver, Lindley Darden, Peter Machamer, and William Wimsatt. On general metaphysics, John Heil and the NEH Summer Seminar he directed helped me an enormous amount.

The work that goes into this book has benefited from so many interactions with so many people over such a long time—including audiences at many talks—that I cannot hope to remember everyone who played a role. I apologize in advance to those I forget. Here are those I remember as most helpful: Darren Abramson, Fred Adams, Neal Anderson, Ken Aizawa, Sonya Bahar, Mark Balaguer, Bill Bechtel, Kenny Boyce, Dave Chalmers, Mark Collier, Carl Craver, Jack Copeland, Martin Davis, Daniel Dennett, Michael Dickson, Eli Dresner, Frances Egan, Chris Eliasmith, Ilke Ercan, Carrie Figdor, Jerry Fodor, Florent Franchette, Nir Fresco, Justin Garson, Carl Gillett, Robert Gordon, Sabrina Haimovici, Gilbert Harman, John Heil, Alberto Hernández, David M. Kaplan, Saul Kripke, Fred Kroon, Bill Lycan, Corey Maley, Marcin Milkowski, Jonathan Mills, Alex Morgan, Toby Ord, Anya Plutynski, Tom Polger, Hilary Putnam, Michael Rabin, Michael Rescorla, Brendan Ritchie, Brad Rives, Martin Roth, Anna-Mari Rusanen, Dan Ryder, Andrea Scarantino, Matthias Scheutz, Susan Schneider, Sam Scott, Larry Shapiro, Rebecca Skloot, Mark Sprevak, Oron Shagrir, Stuart Shapiro, Doron Swade, Wilfried Sieg, Eric Thomson, Brandon Towl, Ray Turner, Dan Weiskopf, and Julie Yoo.

Special thanks to Mark Sprevak, the members of the reading group he ran on a previous version of the manuscript, and an anonymous referee for OUP for their extremely helpful feedback. Thanks to Peter Momtchiloff, my editor at OUP, for his help, patience, and support.

Chapters 1–4 are indebted to G. Piccinini, "Computation in Physical Systems," *The Stanford Encyclopedia of Philosophy* (Fall 2010 Edition), Edward N. Zalta (ed.), URL = http://plato.stanford.edu/archives/fall2010/entries/computation-physicalsystems/

Chapter 1 is indebted to G. Piccinini, "Computing Mechanisms," *Philosophy of Science*, 74.4 (2007), 501–26.

Chapter 3 is indebted to G. Piccinini, "Computation without Representation," *Philosophical Studies*, 137.2 (2008), 205–41 and G. Piccinini, "Functionalism, Computationalism, and Mental Contents," *Canadian Journal of Philosophy*, 34.3 (2004), 375–410.

Chapter 4 is indebted to G. Piccinini, "Computational Modeling vs. Computational Explanation: Is Everything a Turing Machine, and Does It Matter to the Philosophy of Mind?" *Australasian Journal of Philosophy*, 85.1 (2007), 93–115.

Chapter 5 is indebted to G. Piccinini, "Functionalism, Computationalism, and Mental States," *Studies in the History and Philosophy of Science*, 35.4 (2004), 811–33 and G. Piccinini and C. Craver, "Integrating Psychology and Neuroscience: Functional Analyses as Mechanism Sketches," *Synthese*, 183.3 (2011), 283–311.

Chapter 6 is indebted to C. Maley and G. Piccinini, "A Unified Mechanistic Account of Teleological Functions for Psychology and Neuroscience," forthcoming in David Kaplan, ed., *Integrating Psychology and Neuroscience: Prospects and Problems*. Oxford: Oxford University Press.

Chapter 7 is indebted to G. Piccinini, "Computing Mechanisms," *Philosophy of Science*, 74.4 (2007), 501–26 and G. Piccinini, "Computation without Representation," *Philosophical Studies*, 137.2 (2008), 205–41.

Chapters 10–13 are indebted to G. Piccinini, "Computers," *Pacific Philosophical Quarterly*, 89.1 (2008), 32–73.

Chapter 13 is indebted to G. Piccinini, "Some Neural Networks Compute, Others Don't," *Neural Networks*, 21.2–3 (2008), 311–21.

Chapter 14 is indebted to G. Piccinini and A. Scarantino, "Information Processing, Computation, and Cognition," *Journal of Biological Physics*, 37.1 (2011), 1–38.

Chapters 15 and 16 are indebted to "The Physical Church-Turing Thesis: Modest or Bold?" *The British Journal for the Philosophy of Science*, 62.4 (2011), 733–69.

My co-authors for some of the above articles—Carl Craver, Corey Maley, and Andrea Scarantino—helped me develop important aspects of the view I defend and I am deeply grateful to them.

The articles from which the book borrows were usually refereed anonymously. Thanks to the many anonymous referees for their helpful feedback.

Thanks to my research assistant, Frank Faries, especially for drawing the figures for Chapters 8 and 9.

This material is based on work supported by the College of Arts and Sciences at the University of Missouri—St. Louis, the Charles Babbage Institute, the Institute for Advanced Studies at the Hebrew University of Jerusalem, the National Endowment for the Humanities, the National Science Foundation under grants no. SES-0216981 and SES-0924527, the University of Missouri, the University of Missouri—St. Louis, the Mellon Foundation, and the Regione Sardegna. Any opinions, findings, conclusions, and recommendations expressed in this book are those of the author and do not necessarily reflect the views of these funding institutions.

Deep thanks to my friends, partners, and family—my parents, sisters, brothers-in-law, and nieces—for their love and support, especially during my most challenging times. Thanks to my daughters—Violet, Brie, and Martine—for bringing so much meaning and joy to my life.

Introduction

This book is about the nature of concrete computation—the physical systems that perform computations and the computations they perform. I argue that concrete computing systems are a kind of functional mechanism. A functional mechanism is a system of component parts with causal powers that are organized to perform a function. Computing mechanisms are different from *non*-computing mechanisms because they have a special function: to manipulate vehicles based solely on differences between different portions of the vehicles in accordance with a rule that is defined over the vehicles and, possibly, certain internal states of the mechanism. I call this the *mechanistic account* of computation.

When I began articulating and presenting the mechanistic account of computation to philosophical audiences over ten years ago, I often encountered one of two dismissive responses. *Response one*: your view is obvious, well known, and uncontroversial—utterly dull. *Response two*: your view is counterintuitive, implausible, and untenable—totally worthless. These were not the only responses. Plenty of people engaged the substance of the mechanistic account of computation and discussed its pros and cons. But these radical responses were sufficiently common that they deserve to be addressed upfront.

If the mechanistic account elicited either one of these responses but not the other, perhaps the mechanistic account would be at fault. But the presence of both responses is encouraging because they cancel each other out, as it were. Those who respond dismissively appear to be unaware that the opposite dismissive response is equally common. If they knew this, presumably they would tone it down. For although reasonable people may disagree about whether a view is true or false, it is unreasonable to disagree on whether something is *obviously* true or *obviously* false. If it's so obvious, how can there be equally informed people who think the *opposite* is obvious?

The first dismissive response—that the mechanistic account is so obvious that it's dull—seems to be motivated by something like the following reasoning. For the sake of the argument, let's assume along with many philosophers that computation is a kind of symbol manipulation. There is an important distinction between the syntax of symbols (and, more generally, their formal properties) and their semantics. To a first approximation, syntactic (more generally, formal) properties are those that determine whether a symbolic structure is well formed—they make the difference

between 'the puzzle is solvable' and 'puzzle the is solvable'; semantic properties are those that determine what symbols mean—they make the difference between 'i vitelli dei romani sono belli' in most languages, where it means nothing; in Latin, where it means *go, Vitellus, at the Roman god's war cry*; and in Italian, where it means *the calves of the Romans are good-looking*. Most people find it intuitively compelling that computations operate on symbols based on their formal or syntactic properties alone and not at all based on their semantic properties. Furthermore, many philosophers assimilate computational explanation and functional analysis: computational states are often said to be individuated by their functional relations to other computational states, inputs, and outputs. Therefore, computational states and processes are individuated functionally, i.e., formally or syntactically. Saying that computation is mechanistic, as my account does, is just a relabeling of this standard view. Therefore, the mechanistic account of computation is nothing new. Something like this reasoning is behind the first dismissive response. It is deceptively persuasive but, alas, it goes way too fast.

A first problem is that physical systems don't wear their syntactic (or formal) properties on their sleeves. If the mechanistic account were based on syntactic properties, it should begin with an account of syntactic properties that does not presuppose the notion of computation. I don't know of any such account, and fortunately I don't need one. For the mechanistic account of computation is painstakingly built by specifying which properties of which mechanisms are computational, without ever invoking the notion of syntax (or formal property). Thus, the mechanistic account may provide ingredients for an account of syntax—not vice versa (Chapter 3, Section 4).

A second problem is the implicit assimilation of functional analysis and computational explanation, which is pervasive in the literature. I reject such an assimilation and argue that functional analysis provides a partial sketch of a mechanism (Chapter 5), defend a teleological account of functional mechanisms (Chapter 6), and argue that computational explanation is a specific kind of mechanistic explanation (Chapter 7).

An additional issue is that computations are often individuated semantically—in terms of functions from what is denoted by their inputs to what is denoted by their outputs. And philosophers interested in computation are often interested in how computation can explain cognition, which is usually assumed to deal in representations. After all, cognitive states and processes are typically individuated at least in part by their semantic content. Thus, many philosophers interested in computation believe that computational states and processes are individuated by their content in such a way that at least part of their essence is semantic. I call this the *semantic account* of computation. Therein lies the motivation for the second dismissive response: since computation is essentially semantic and the mechanistic account of computation denies this, the mechanistic account is obviously and horribly wrong.

But the semantic account of computation has its own problems. For starters, the notion of semantic property is even more obscure and more in need of naturalistic explication than that of syntactic property. In addition, I argue that individuating computations semantically always presupposes their non-semantic individuation, and that some computations are individuated purely non-semantically. Therefore, contrary to the second dismissive response, computation does not presuppose representation (Chapter 3).

But if we reject the view that computation presupposes representation, we risk falling into the view that everything performs computations—pancomputationalism (Chapter 4). This is not only counterintuitive—it also risks undermining the foundations of computer science and cognitive science. It is also a surprisingly popular view. Yet, I argue that pancomputationalism is misguided and we can avoid it by a judicious use of mechanistic explanation (Chapter 4).

The mechanistic account begins by adapting a mechanistic framework from the philosophy of science. This gives us identity conditions for mechanisms in terms of their components, their functions, and their organization, without invoking the notion of computation. To this general framework, a mechanistic account of computation must add criteria for what counts as computationally relevant mechanistic properties. I do this by adapting the notion of a string of letters, taken from logic and computability theory, and generalizing it to the notion of a system of vehicles that are defined solely based on differences between different portions of the vehicles. Any system whose function is to manipulate such vehicles in accordance with a rule, where the rule is defined in terms of the vehicles themselves, is a computing system. I explain how a system of appropriate vehicles can be found in the natural (concrete) world, yielding a robust (nontrivial) notion of computation (Chapter 7).

After that, I develop this general mechanistic account by explicating specific computing systems and their properties in mechanistic terms. I explicate the notion of primitive computing components (Chapter 8), complex computing components (Chapter 9), digital calculators (Chapter 10), digital computers (Chapter 11), analog computers (Chapter 12), and neural networks (Chapter 13). After that, I return to semantic properties under the guise of information processing (in several senses of the term); I argue that processing information is a form of computation but computation need not be a form of information processing (Chapter 14). I conclude the book with the limits of physical computation. Once the relevant question is clarified (Chapter 15), the evidence suggests that any function that is physically computable is computable by Turing machines (Chapter 16).

1

Towards an Account of Physical Computation

1. Abstract Computation and Concrete Computation

Computation may be studied mathematically by formally defining computing systems, such as algorithms and Turing machines, and proving theorems about their properties. The mathematical theory of computation is a well-established branch of mathematics. It studies which functions defined over a denumerable domain, such as the natural numbers or strings of letters from a finite alphabet, are computable by algorithm or by some restricted class of computational systems. It also studies how much time or space (i.e., memory) it takes for a computational system to compute certain functions, without worrying much about the particular units of time involved or how the memory cells are physically implemented.

By contrast, most uses of computation in science and everyday life involve *concrete* computation: computation in concrete physical systems such as computers and brains. Concrete computation is closely related to mathematical computation: we speak of physical systems as running an algorithm or as implementing a Turing machine, for example. But the relationship between concrete computation and mathematical computation is not part of the mathematical theory of computation per se and requires further investigation. This book is about concrete computation. We will see in due course that questions about concrete computation are not neatly separable from mathematical results about computation. The following mathematical results are especially crucial to our subsequent investigation.

The most important notion of computation is that of digital computation, which Alan Turing, Kurt Gödel, Alonzo Church, Emil Post, and Stephen Kleene formalized in the 1930s (see the Appendix for a sketch of the most relevant background and results). Their work investigated the foundations of mathematics. One crucial question was whether first order logic is decidable—whether there is an algorithm that determines whether any given first order logical formula is a theorem.

Turing (1936–7) and Church (1936) proved that the answer is negative: there is no such algorithm. To show this, they offered precise characterizations of the informal notion of algorithmically computable function. Turing did so in terms of so-called Turing machines—devices that manipulate discrete symbols written on a tape in

accordance with finitely many instructions. Other logicians did the same thing—they formalized the notion of algorithmically computable function—in terms of other notions, such as λ-definable functions and general recursive functions.

To their surprise, all such notions turned out to be extensionally equivalent, that is, any function computable within any of these formalisms is computable within any of the others. They took this as evidence that their quest for a precise definition of 'algorithm,' or 'effective procedure,' or 'algorithmically computable function,' had been successful. They had found a precise, mathematically defined counterpart to the informal notion of computation by algorithm—a mathematical notion that could be used to study in a rigorous way which functions can and cannot be computed by algorithm, and therefore which functions can and cannot be computed by machines that follow algorithms. The resulting view—that Turing machines and other equivalent formalisms capture the informal notion of algorithm—is now known as the *Church-Turing thesis* (more on this in Chapter 15). It provides the foundation for the mathematical theory of computation as well as mainstream computer science.

The theoretical significance of Turing et al.'s formalization of computation can hardly be overstated. As Gödel pointed out (in a lecture following one by Tarski):

> Tarski has stressed in his lecture (and I think justly) the great importance of the concept of general recursiveness (or Turing's computability). It seems to me that this importance is largely due to the fact that with this concept one has for the first time succeeded in giving an absolute definition of an interesting epistemological notion, i.e., one not depending on the formalism chosen (Gödel 1946, 84).

A standard Turing machine computes only one function. Turing also showed that there are *universal* Turing machines—machines that can compute any function computable by any other Turing machine. Universal machines do this by executing instructions that encode the behavior of the machine they simulate. Assuming the Church-Turing thesis, universal Turing machines can compute any function computable by algorithm. This result is significant for computer science: you don't need to build different computers for different functions; one universal computer will suffice to compute any computable function. Modern digital computers approximate universal machines in Turing's sense: digital computers can compute any function computable by algorithm for as long as they have time and memory. (Strictly speaking, a universal machine has an unbounded memory, whereas digital computer memories can be extended but not indefinitely, so they are not quite unbounded in the same way.)

The above result should not be confused with the common claim that computers can compute *anything*. Nothing could be further from the truth: another important result of computability theory is that most functions are not computable by Turing machines (thus, by digital computers). Turing machines compute functions defined over denumerable domains, such as strings of letters from a finite alphabet. There are uncountably many such functions. But there are only countably many Turing

machines; you can enumerate Turing machines by enumerating all lists of instructions. Since an uncountable infinity is much larger than a countable one, it follows that Turing machines (and hence digital computers) can compute only a tiny portion of all functions (over denumerable domains, such as natural numbers or strings of letters).

Turing machines and most modern computers are known as (classical) *digital* computers, that is, computers that manipulate strings of discrete, unambiguously distinguishable states. Digital computers are sometimes contrasted with *analog* computers, that is, machines that manipulate continuous variables. Continuous variables are variables that can change their value continuously over time while taking any value within a certain interval. Analog computers are used primarily to solve certain systems of differential equations (Pour-El 1974; Rubel 1993).

Classical digital computers may also be contrasted with *quantum* computers (Nielsen and Chuang 2000). Quantum computers manipulate quantum states called *qudits* (most commonly *binary* qudits, or *qubits*). Unlike the computational states of digital computers, qudits are not unambiguously distinguishable from one another. This book will focus on classical (i.e., non-quantum) computation.

The same entities studied in the mathematical theory of computation—Turing machines, algorithms, and so on—are said to be implemented by concrete physical systems. This poses a problem: under what conditions does a concrete, physical system perform a computation when computation is defined by an abstract mathematical formalism? This may be called the *problem of computational implementation*.

The problem of computational implementation may be formulated in a couple of different ways, depending on our ontology of mathematics. Some people interpret the formalisms of computability theory, as well as other portions of mathematics, as defining and referring to abstract objects. According to this interpretation, Turing machines, algorithms, and the like are abstract objects.[1]

Abstract objects in this sense should not be confused with abstraction in the sense of focusing on one aspect of something at the expense of other aspects. For instance, we talk about the economy of a country and discuss whether it is growing or contracting; we do so by abstracting away from many other aspects of the objects and properties that constitute that country. I will discuss this notion of abstraction (partial consideration) later. Now let's deal with abstract objects.

Abstract objects are putative entities that are supposed to be non-spatial, non-temporal, and non-causal. In other words, abstract objects have no spatial location, do not exist through time, and are causally inert. The view that there are abstract mathematical objects and that our mathematical truths describe such objects truly is called *platonism* (Balaguer 2009; Linnebo 2011; Rodriguez-Pereyra 2011; Rosen 2012;

[1] E.g.: 'Computational models are *abstract entities*. They are not located in space and time, and they do not participate in causal interactions' (Rescorla 2014b, 1277, emphasis added).

Swoyer 2008). According to platonism, mathematical entities such as the number 2 are abstract objects: the number 2 is in no particular place, exists atemporally, and is causally inert. Yet, when we say that 2 is the square root of 4, we say something true about that abstract object.

If you believe that computability theory is about abstract objects and there are such things—that is, if you are a platonist—you may ask: what must be the case for a given concrete physical system to implement a given abstract computational object (as opposed to another abstract object, or none at all)? This is how platonists formulate the problem of computational implementation.[2]

Non-platonists treat the formalisms of computability theory simply as abstract computational *descriptions*, without positing any abstract objects as their referents. The main contemporary alternative to mathematical platonism is the view that mathematics is putatively about abstract objects but there are no such things. Platonists are right that mathematical objects such as numbers, sets, functions, and Turing machines would be abstract objects, if there were any. But there are no abstract objects. Strictly speaking, there are no numbers, no sets, no functions, no Turing machines, etc. Therefore, strictly speaking, existential mathematical statements are not true; universal mathematical statements are vacuously true (for lack of referents). This view goes by the name of *fictionalism* (Field 1980, 1989; Balaguer 2013; Paseau 2013; Bueno 2009; Leng 2010).

If you believe there are no abstract objects, you may ask: what does it take for a given concrete physical system to satisfy a given abstract computational description (as opposed to another, or none at all)? This is how a non-platonist formulates the problem of computational implementation. Regardless of how the problem of computational implementation is formulated, solving it requires an account of concrete computation—an account of what it takes for a given physical system to perform a given computation.

A closely related problem is that of distinguishing between physical systems such as digital computers, which appear to compute, and physical systems such as rocks, which appear not to compute. Unlike computers, ordinary rocks are not sold in computer stores and are usually not taken to perform computations. Why? What do computers have that rocks lack, such that computers compute and rocks don't? (If indeed they don't?)

Questions on the nature of concrete computation should not be confused with questions about computational modeling. The dynamical evolution of many physical systems may be described by computational models. Computational models describe the dynamics of a system that are written into, and run by, a computer. The behavior of rocks—as well as rivers, ecosystems, and planetary systems, among

[2] E.g.: "we need a theory of implementation: the relation that holds between an *abstract computational object*... and a physical system" (Chalmers 2011, 325, emphasis added).

many others—may well be modeled computationally. From this, it doesn't follow that the modeled systems are computing devices—that they themselves perform computations. Prima facie, only relatively few and quite special systems compute. Explaining what makes them special—or explaining away our feeling that they are special—is the job of an account of concrete computation.

This book offers an account of concrete computation. I have reservations about both platonism and fictionalism but this is not the place to air them. I will remain officially neutral about the ontology of mathematics; my account works under any view about mathematical ontology.

2. Abstract Descriptions

Before developing an account of concrete computation, we should introduce a notion of abstraction that has little to do with abstract objects. Concrete physical systems satisfy a number of different descriptions. For example, one and the same physical system may be a juicy pineapple, a collection of cells that are bound together and release a certain amount of liquid under a certain amount of pressure, an organized system of molecules that give out certain chemicals under certain conditions, a bunch of atoms some of whose bonds break under certain conditions, etc. By the same token, the same physical system may be a Dell Latitude computer, a processor connected to a memory inside a case, a group of well-functioning digital circuits, an organized collection of electronic components, a vast set of molecules arranged in specific ways, etc.

A first observation about the above examples is that some descriptions involve smaller entities (and their properties) than others. Atoms are generally smaller than molecules, which are smaller than cells and electronic components, which in turn are smaller than pineapples and processors.

A second observation is that, within a complex system, usually larger things are made out of smaller things. Atoms compose molecules, which compose cells and circuits, which compose pineapples and processors.

A third observation is that in order to describe the same property of, or activity by, the same system, descriptions in terms of larger things and their properties are often more economical than descriptions in terms of their components and their properties. For example, it only takes a couple of predicates to say that something is a juicy pineapple. If we had to provide the same information by describing the system's cells and their properties, we would need to describe the kinds of cell, how they are arranged, and how much liquid they release when they are squished (and the relevant amount of squishing). To do this in an adequate way would take a lot of work. To do this in terms of the system's molecules and their properties would require giving even more information, which would take even more work. Doing this in terms of the system's atoms and their properties would be even worse. The same point applies to computing systems. It only takes one predicate to say that something is a Dell

Latitude. It would take considerably more to say enough about the system's processor, memory, and case to distinguish them from non-Dell-Latitude processors, memories, and cases. It would take even more to say the same thing in terms of the system's digital circuits, even more in terms of the system's electronic components, and even more in terms of the system's atoms.

Descriptions of the same system may be ranked according to how much information they require in order to express the same property or properties of the system. Descriptions that are more economical than others achieve their economy by expressing select information about the system at the expense of other information. In other words, economical descriptions omit many details about a system. They capture some aspects of a system while ignoring others. Following tradition, I will call a description more or less *abstract* depending on how many details it omits in order to express that a system possesses a certain property. The more economically a description accomplishes this, the more abstract it is. In our example, 'being a Dell Latitude' is more abstract than 'being a certain type of processor connected to a certain type of memory inside a certain kind of case,' which in turn is more abstract than 'being such and such a system of organized digital circuits.'

Mathematical descriptions of concrete physical systems are abstract in this sense. They express certain properties (e.g., shape, size, quantity) while ignoring others (e.g., color, chemical composition). In this book, I will give an account of computing systems according to which computational descriptions of concrete physical systems are mathematical and thus abstract in the same sense. They express certain properties (e.g., which function a system computes) in an economical way while ignoring others.

A platonist may object as follows. What about Turing machines and other computational entities defined purely mathematically, whose properties are studied by computability theory? Surely those are not concrete physical systems. Surely mathematicians are free to define computational entities, study their properties, and discover truths about them, without their descriptions needing to be true of any physical system. For instance, Turing machines have unbounded tapes, whereas there is a limit to how much the memory of a concrete computer can be extended. Therefore, Turing machines are not concrete physical systems. Therefore, Turing machines must be abstract objects. Therefore, we still need abstract computational objects to act as truthmakers for our computational descriptions.

My reply is simply that I am after an account of computation in the physical world. If Turing machines and other mathematically defined computational entities are abstract objects—and their descriptions are not true of any physical system—they fall outside the scope of my account. Therefore, even if we concluded that, if computability theory refers, it refers to abstract objects, this would not change my account of concrete computation.

3. Towards an Account of Concrete Computation

Here is where we've gotten so far: the problem of computational implementation is what distinguishes physical processes that count as computations from physical processes that do not, and which computations they count as. Equivalently, it is the question of what distinguishes physical systems that count as computing systems from physical systems that do not, and which computations they perform.

The problem of computational implementation takes different forms depending on how we think about mathematically defined computation. In this book I will assume that mathematically defined computation is an abstract (mathematical) description of a hypothetical process, which may be idealized in various ways and may or may not be physically realizable. The mathematical study of computation employs abstract descriptions without much concern for physical implementation. Concrete computation is a kind of physical process. It may be described in very concrete terms or more abstract terms. If the description is abstract enough in ways that I will articulate, it is a computational description.

If, on the contrary, you take abstract computation to be the description of abstract objects, then the question becomes one of identifying a relationship between concrete objects and their abstract counterparts such that the former count as implementations of the latter. Everything I say in this book may be reinterpreted within this platonist framework so that it answers the platonist question. (*Mutatis mutandis* for the fictionalist.) For by the time I show how certain concrete objects satisfy the relevant computational descriptions, I also solve the platonistically formulated problem of implementation. All the platonist has left to do is to interpret the computational descriptions as referring to abstract objects, and posit an appropriate relation between the abstract objects thus referred to and the concrete objects that putatively implement them.

In the rest of the book, I will defend the following solution to the problem of computational implementation.

> **The Mechanistic Account of Computation**: A physical computing system is a mechanism whose teleological function is performing a physical computation. A physical computation is the manipulation (by a functional mechanism) of a medium-independent vehicle according to a rule. A medium-independent vehicle is a physical variable defined solely in terms of its degrees of freedom (e.g., whether its value is 1 or 0 during a given time interval), as opposed to its specific physical composition (e.g., whether it's a voltage and what voltage values correspond to 1 or 0 during a given time interval). A rule is a mapping from inputs and/or internal states to internal states and/or outputs.

All of this will be unpacked in more detail in Chapter 7, after laying appropriate groundwork. The overarching argument for this mechanistic account is that it satisfies a number of desiderata better than the competition.

4. Desiderata

An account of concrete computation that does justice to the sciences of computation should have the following features: 1) objectivity; 2) explanation; 3) the right things compute; 4) the wrong things don't compute; 5) miscomputation is explained; and 6) an adequate taxonomy of computing systems is provided.

1 Objectivity

An account with objectivity is such that whether a system performs a particular computation is a matter of fact. Contrary to objectivity, some authors have suggested that computational descriptions are vacuous—a matter of free interpretation rather than fact. The alleged reason is that any system may be described as performing just about *any* computation, and there is no further fact of the matter as to whether one computational description is more accurate than another (Putnam 1988; Searle 1992; cf. Chapter 4, Section 2). This conclusion may be informally derived as follows.

Assume a simple mapping account of computation (Chapter 2): a system performs computation C if and only if there is a mapping between a sequence of states individuated by C and a sequence of states individuated by a physical description of the system (Putnam 1960, 1967b, 1988). Assume, as some have, that there are no constraints on which mappings are acceptable, so that any sequence of computational states may map onto any sequence of physical states of the same cardinality. If the sequence of physical states has larger cardinality, the computational states may map onto either equivalent classes or a subset of the physical states. Since physical variables can generally take real numbers as values and there are uncountably many of those, physical descriptions generally give rise to uncountably many states and state transitions. But ordinary computational descriptions give rise only to countably many states and state transitions. Therefore, there is a mapping from any (countable) sequence of computational state transitions onto either equivalent classes or a subset of physical states belonging to any (uncountable) sequence of physical state transitions. Therefore, generally, any physical system performs any computation.

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

Computer scientists and engineers appeal to empirical facts about the systems they study to determine which computations are performed by which systems. They apply computational descriptions to concrete systems in a way entirely analogous to other bona fide scientific descriptions. In addition, many psychologists and neuroscientists are in the business of discovering which computations are performed by minds and

brains. When they disagree, they address their opponents by mustering empirical evidence about the systems they study. Unless the prima facie legitimacy of those scientific practices can be explained away, a good account of concrete computation should entail that there is a fact of the matter as to which computations are performed by which systems.

2 Explanation

Computations performed by a system may explain its capacities. Ordinary digital computers are said to execute programs, and their capacities are normally explained by appealing to the programs they execute. The literature on computational theories of cognition contains explanations that appeal to the computations performed by the brain. The same literature also contains claims that cognitive capacities ought to be explained computationally, and more specifically, by program execution (e.g., Fodor 1968b; Cummins 1977; cf. Chapter 5). A good account of computing mechanisms should say how appeals to program execution, and more generally to computation, explain the behavior of computing systems. It should also say how program execution relates to the general notion of computation: whether they are the same and, if not, how they are related.

3 The right things compute

A good account of computing mechanisms should entail that paradigmatic examples of computing mechanisms, such as digital computers, calculators, both universal and non-universal Turing machines, and finite state automata, compute.

4 The wrong things don't compute

A good account of computing mechanisms should entail that all paradigmatic examples of non-computing mechanisms and systems, such as planetary systems, hurricanes, and digestive systems, don't perform computations.

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

The accounts just mentioned fail desideratum 3 or 4. Why, what's wrong with that? There are debatable cases, such as look-up tables and analog computers. Whether those really compute may be open to debate, and in some cases it may be open to stipulation. But there are plenty of clear cases. Digital computers, calculators, Turing

machines, and finite state automata are paradigmatic computing systems. They constitute the subject matter of computer science and computability theory. Planetary systems, the weather, and digestive systems are paradigmatic non-computing systems[3]; at the very least, it's not obvious how to explain their behavior computationally. If we can find an account that works for the clear cases, the unclear ones may be left to fall wherever the account says they do—"spoils to the victor" (Lewis 1986, 203; cf. Collins, Hall, and Paul 2004, 32).

Insofar as the assumptions of computer scientists and computability theorists ground the success of their science as well as the appeal of their notion of computation to practitioners of other disciplines, they ought to be respected. By satisfying desiderata 3 and 4, a good account of physical computation draws a principled distinction between systems that compute and systems that don't, and it draws it in a place that fits the presuppositions of good science.

5 *Miscomputation is explained*

Computations can go wrong. To a first approximation, a system M miscomputes just in case M is computing function f on input i, $f(i) = o_1$, M outputs o_2, and $o_2 \neq o_1$. Here o_1 and o_2 represent any possible outcome of a computation, including the possibility that the function is undefined for a given input, which corresponds to a non-halting computation. Miscomputation is analogous to misrepresentation (Dretske 1986), but it's not the same. Something (e.g., a sorter) may compute correctly or incorrectly regardless of whether it represents or misrepresents anything. Something (e.g., a painting) may represent correctly or incorrectly regardless of whether it computes or miscomputes anything.

Fresco (2013, Chapter 2) points out that the above definition of miscomputation is too permissive. Consider a system M that is computing function f on input i, with $f(i) = o_1$, and outputs o_1. Even though the output is correct, M may have failed to compute $f(i)$ correctly. M outputting o_1 may be due not to a correct computation of $f(i)$ but to random noise in M's output device, or to M following a computational path that has nothing to do with $f(i)$ but still results in o_1, or to M switching from computing $f(i)$ to computing $g(i)$, where $g \neq f$, but $g(i) = o_1$. In practice, these are unlikely scenarios. Nevertheless, we may still wish to say that a miscomputation occurred.

To insure this, we may restrict our definition of miscomputation as follows. As before, let M be a system computing function $f(i) = o_1$. Let P be the procedure M is supposed to follow in computing $f(i)$, with P consisting of computational steps $s_1, s_2, \ldots s_n$. By definition, the outcome of s_n is o_1. Let s_i be the ith step in the sequence $s_1, s_2, \ldots s_{n-1}$.

[3] For evidence, see Fodor 1968b, 632; Fodor 1975, 74; Dreyfus 1979, 68, 101–2; Searle 1980, 37–8; Searle 1992, 208.

M miscomputes f(i) just in case, for any step s_i, after *M* takes step s_i, either *M* takes a computational step other than s_{i+1} or *M* fails to continue the computation.

In other words, *M* miscomputes just in case it fails to follow every step of the procedure it's supposed to follow all the way until producing the correct output.

Although miscomputation has been ignored by philosophers until very recently, a good account of computing mechanisms should explain how it's possible for a physical system to miscompute. This is desirable because miscomputation, or more informally, making computational mistakes, plays an important role in computer science and its applications. Those who design and use computing mechanisms devote a large portion of their efforts to avoiding miscomputations and devising techniques for preventing them. To the extent that an account of computing mechanisms makes no sense of that effort, it is unsatisfactory.

6 Taxonomy

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

To illustrate, consider Robert Cummins's account. According to Cummins (1983), for something to compute, it must execute a program. He also maintains that executing a program amounts to following the steps described by the program. This leads to paradoxical consequences. Consider that many paradigmatic computing mechanisms (such as non-universal Turing machines and finite state automata) are not characterized by computer scientists as executing programs, and they are considerably less powerful than the systems that are so characterized (i.e., universal Turing machines and idealized digital computers). Accordingly, we might conclude that non-universal Turing machines, finite state automata, etc., do not really compute. But this violates desideratum 3 (the right things compute). Alternatively, we might observe along with Cummins that non-universal Turing machines and finite states automata do follow the steps described by a program. Therefore, by Cummins's light, they execute a program, and hence they compute. But now we find it difficult to explain why they are less powerful than ordinary digital computers.[4] For under Cummins's account, we cannot say that, unlike digital computers, those other

[4] If we make the simplifying assumption that ordinary digital computers have a memory of fixed size, they are equivalent to (very special) finite state automata. Here I am contrasting ordinary digital computers with ordinary finite state automata.

systems lack the flexibility that comes with the capacity to execute programs. The difference between computing systems that execute programs and those that don't is important to computer science and computing applications, and it should make a difference to theories of cognition. We should prefer an account that honors that kind of difference to one that is blind to it.

With these desiderata as landmarks, I will proceed to formulate and evaluate different accounts of concrete computation. I will argue that the account that best satisfies these desiderata is the mechanistic one.

2

Mapping Accounts

1. The Simple Mapping Account

In this Chapter, I will introduce and assess one of two traditional types of account of physical computation: the mapping account. I will check which desiderata it satisfies. In the next chapter, I will repeat this exercise with respect to the other traditional type of account: the semantic account. In the following two chapters, I will examine in greater depth some deficiencies of the mapping and semantic accounts. This will pave the way for the mechanistic account.

In some fields, such as computer science and cognitive science, there are scientific theories that explain the capacities of certain physical systems—respectively, computers and brains—by appealing to the computations they perform. What does that mean? What does it take for a physical system to perform a certain computation? In answering these questions, I will mainly focus on computational states; the same conclusions apply to computational inputs and outputs.

One of the earliest and most influential types of account of concrete computation is the mapping account, which can be traced to work by Hilary Putnam on the metaphysics of mind (1960, 1967b). To a first approximation, the account says that anything that is accurately described by a computational description *C* is a computing system implementing *C*.

More precisely, Putnam sketches his earliest account in terms of Turing machines only, appealing to the "machine tables" that are a standard way of defining specific Turing machines. A machine table consists of one column for each of the (finitely many) internal states of the Turing machine and one row for each of the machine's (finitely many) symbol types. Each entry in the machine table specifies what the machine does given the pertinent symbol and internal state. Here is how Putnam explains what it takes for a physical system to be a Turing machine:

> A 'machine table' describes a machine if the machine has internal states corresponding to the columns of the table, and if it 'obeys' the instruction in the table in the following sense: when it is scanning a square on which a symbol s_1 appears and it is in, say, state *B*, that it carries out the 'instruction' in the appropriate row and column of the table (in this case, column *B* and row s_1). Any machine that is described by a machine table of the sort just exemplified is a Turing machine (Putnam 1960/1975a, 365; cf. Putnam 1967b/1975a, 433–4).

This account relies on several unexplained notions, such as square (of tape), symbol, scanning, and carrying out an instruction. Furthermore, the account is specified in terms of Turing machine tables, but there are other kinds of computational description. A general account of concrete computation should cover other computational descriptions besides Turing machine tables. Perhaps for these reasons, Putnam—soon followed by many others—abandoned reference to squares, symbols, scanning, and other notions specific to Turing machines; he substituted them with an appeal to a "physical" description of the system. The result of that substitution is what Godfrey-Smith (2009) dubs the "simple mapping account" of computation.

Putnam appeals to "physical" descriptions and contrasts them to "computational" descriptions, as if there were an unproblematic distinction between the two. This way of talking is common but problematic. As the literature on the thermodynamics of computation demonstrates (Leff and Rex 2003), computational descriptions are physical too. In any case, the present topic is physical computation. We cannot presuppose that computational descriptions are not physical.

The right contrast to draw, which is probably the contrast Putnam was intending to get at, is that between a computational description of a physical system and a more fine-grained physical description of the same system. For that reason, I will use the term 'microphysical' where Putnam and his followers write 'physical'. This use of 'microphysical' does not imply that a microphysical description is maximally fine-grained. It need not describe all the degrees of freedom of the physical system. What matters is that the microphysical description is more fine-grained (describes more degrees of freedom) than the computational description of the same system. For instance, even a description of the temperature of a system—a typical macroscopic description in standard thermodynamics—may count as microphysical (in the present sense) relative to a computational description of the same system. With this terminological caveat in place, I proceed to formulate Putnam's account of concrete computation.

According to the simple mapping account, a physical system *S* performs computation *C* just in case:

(i) There is a mapping from the states ascribed to *S* by a microphysical description to the states defined by computational description *C*, such that:
(ii) The state transitions between the microphysical states mirror the state transitions between the computational states.

Clause (ii) requires that for any computational state transition of the form $s_1 \rightarrow s_2$ (specified by the computational description *C*), if the system is in the microphysical state that maps onto s_1, it then goes into the microphysical state that maps onto s_2.

One difficulty with the formulation above is that ordinary microphysical descriptions, such as systems of differential equations, generally ascribe uncountably many states to physical systems, whereas ordinary computational descriptions, such as

Turing machine tables, ascribe at most countably many states. Thus, there aren't enough computational states for the microphysical states to map onto. One solution to this problem is to reverse the direction of the mapping, requiring a mapping of the computational states onto (a subset of) the microphysical states. Another, more common solution to this problem—often left implicit—is to select either a subset of the microphysical states or equivalence classes of the microphysical states and map those onto the computational states. When this is done, clause (i) is replaced by the following:

(i') There is a mapping from a subset of (or equivalence classes of) the states ascribed to *S* by a microphysical description to the states defined by computational description *C*.

The simple mapping account turns out to be very liberal: it attributes many computations to many systems. In the absence of restrictions on which mappings are acceptable, such mappings are easy to come by. To generate an easy mapping, just pick an arbitrary computational description of the ordinary digital kind, such as a Turing machine or a C++ program. Such a description gives rise to (countable) sequences of computational states. Pick an arbitrary sequence of computational states $s_1, \ldots s_n$. Now pick an ordinary microphysical description of a physical system, such as a system of differential equations. Finally, take one arbitrary trajectory of the system through the state space defined by the microphysical description, select an arbitrary countable subset of the states that form that trajectory, and map that countable subset onto states $s_1, \ldots s_n$. You're done: you have mapped an arbitrary sequence of microphysical states onto an arbitrary sequence of computational states. As a consequence, some have argued that every physical system implements every computation, or at least that most systems implement most (nonequivalent) computations (Putnam 1988; Searle 1992). I call this conclusion *unlimited pancomputationalism*.

Pancomputationalism, whether limited or unlimited, is a complex topic that I will address more systematically in Chapter 4. For now, the important point is that any account of concrete computation that entails *unlimited* pancomputationalism violates most of our desiderata, beginning with desideratum 1 (objectivity). The objectivity desideratum says that whether a system performs a particular computation is a matter of fact. But the procedure outlined above generates arbitrary mappings between arbitrary microphysical descriptions and arbitrary computational descriptions. Empirical facts about the physical system make no difference. The details of the computation being described make no difference. As a result, under the simple mapping account, claiming that a physical system performs a computation is a vacuous claim. There is almost nothing objective about it, at least in the sense that empirical facts make no difference to whether it obtains. In fact, proponents of unlimited pancomputationalism conclude that whether physical systems are

computational is vacuous and uninteresting; therefore, the computational theory of cognition is vacuous and uninteresting (Putnam 1988; Searle 1992).

On the contrary, the claim that a physical system is computational is interesting and has objective content. There are well-developed sciences devoted to figuring out which physical systems perform which computations. It would be amazing if they were so ill-founded that there is no fact of the matter. The problem is not with the claim that a physical system is computational. The problem is with the simple mapping account of concrete computation. The simple mapping account is inadequate precisely because it trivializes the claim that a physical system performs a computation. The desire to avoid this trivialization is one motivation behind other accounts of concrete computation.

2. Causal, Counterfactual, and Dispositional Accounts

One way to construct accounts of computation that are more restrictive than the simple mapping account is to impose constraints on acceptable mappings. Specifically, clause (ii) may be modified so as to require that the conditional that specifies the relevant microphysical state transitions be logically stronger than a material conditional.

As the simple mapping account has it, clause (ii) requires that for any computational state transition of the form $s_1 \rightarrow s_2$ (specified by a computational description), if a system is in a microphysical state that maps onto s_1, it then goes into a microphysical state that maps onto s_2. The second part of (ii) is a material conditional. It may be strengthened by turning it into a logically stronger conditional—specifically, a conditional expressing a relation that supports counterfactuals.

In a pure counterfactual account, clause (ii) is strengthened by requiring that the microphysical state transitions support certain counterfactuals (Maudlin 1989; Copeland 1996; Rescorla 2014). In other words, the pure *counterfactual account* requires the mapping between computational and microphysical descriptions to be such that the counterfactual relations between the microphysical states are isomorphic to the counterfactual relations between the computational states.

Different authors formulate the relevant counterfactuals in slightly different ways: (a) if the system had been in a microphysical state that maps onto an arbitrary computational state (specified by the relevant computational description), it would then have gone into a microphysical state that maps onto the relevant subsequent computational state (as specified by the computational description) (Maudlin 1989, 415); (b) if the system had been in a microphysical state that maps onto s_1, it would have gone into a microphysical state that maps onto s_2 (Copeland 1996, 341); (c) if the system were in a microphysical state that maps onto s_1, it would go into a microphysical state that maps onto s_2 (Chalmers 1996b, 312). Regardless of the exact formulation, none of these counterfactuals are satisfied by the material

conditional of clause (ii) as it appears in the simple mapping account of computation. Thus, counterfactual accounts are stronger than the simple mapping account.[1]

An account of concrete computation in which the microphysical state transitions support counterfactuals may also be generated by appealing to causal or dispositional relations, assuming—as most people do—that causal or dispositional relations support counterfactuals.

In a *causal account*, clause (ii) is strengthened by requiring a causal relation between the microphysical states: for any computational state transition of the form $s_1 \rightarrow s_2$ (specified by a computational description), if a system is in a microphysical state that maps onto s_1, its microphysical state *causes* it to go into a microphysical state that maps onto s_2 (Chrisley 1995; Chalmers 1994: 1996; Scheutz 1999, 2001).

To this causal constraint on acceptable mappings, David Chalmers (1994, 1996, 2011[2]) adds a further restriction: a genuine physical implementation of a computational system must divide into separate physical components, each of which maps onto the components specified by the computational formalism. Appealing to a decomposition of the system into its components is a step in the direction of a mechanistic account of computation, like the one I will present in Chapter 7. For now, the causal account *simpliciter* requires only that the mappings between computational and microphysical descriptions be such that the causal relations between the microphysical states are isomorphic to the relations between state transitions specified by the computational description. Thus, according to the causal account, computational descriptions specify the causal structure of a physical system at some level of abstraction (in the sense of Chapter 1, Section 2). Concrete computation is the causal structure of a physical process (at some level of abstraction).

In a *dispositional account*, clause (ii) is strengthened by requiring a dispositional relation between the microphysical states: for any computational state transition of the form $s_1 \rightarrow s_2$ (specified by a computational description), if the system is in the microphysical state that maps onto s_1, the system manifests a disposition whose manifestation is the transition from a microphysical state that maps onto s_1 to a microphysical state that maps onto s_2 (Klein 2008). In other words, the dispositional account requires the mapping between computational and microphysical descriptions to be such that the dispositional relations between the microphysical states are isomorphic to the relations between state transitions specified by the computational description. Thus, according to the dispositional account, concrete computation is the dispositional structure of a physical process (again, at some level of abstraction).

[1] Rescorla (2014b) adds the important point that the counterfactuals must be applied to the world within appropriate descriptive practices.

[2] Chalmers 2011 is followed by a series of 12 articles and a response by Chalmers in the same journal (issues 12.4, 13.1, 13.2, 13.4), which discuss causal accounts of computation in greater detail than I can do here.

There is a tight connection between counterfactual, causal, and dispositional accounts. As mentioned above, most people hold that causal and dispositional relations support counterfactuals. If so, then causal and dispositional accounts entail clause (ii) of the counterfactual account. But appealing to causation or dispositions may have advantages over appealing to counterfactuals alone, because supporting counterfactuals may be a weaker condition than being a cause or a disposition. Thus, causal and dispositional accounts may be able to block unwanted computational implementations that are allowed by the pure counterfactual account (Klein 2008, 145, makes the case for dispositional versus counterfactual accounts).

Another condition that satisfies counterfactuals is being a natural law. Accordingly, another closely related account says that clause (ii) is strengthened by requiring that a natural law hold between the computational states: for any computational state transition of the form $s_1 \rightarrow s_2$ (specified by a computational description), if the system is in a microphysical state that maps onto s_1, it is a natural law that the system goes into a microphysical state that maps onto s_2. This may be called the *nomological account* of computation. I am not aware of anyone defending this account, but it's another variation on the same theme.

According to a popular account of dispositions, dispositions are causal powers (and vice versa). If so, then the dispositional account entails the causal account (and vice versa). Similarly, many people hold that natural laws entail dispositions (or causal powers, which may be the same thing), or that dispositions (causal powers) entail natural laws, or both. In other words, a plausible view is that a physical system has causal powers C if and only if it has dispositions whose manifestations are C, and this is the case if and only if the system follows natural laws corresponding to C. To the extent that something along these lines is correct, the three accounts (causal, dispositional, and nomological) are equivalent to one another.

The difference between the simple mapping account on one hand and counterfactual, causal, and dispositional accounts on the other hand may be seen by examining a simple example.

Consider a rock under the sun, early in the morning. During any time interval, the rock's temperature rises. The rock goes from temperature T to temperature T+1, to T+2, to T+3. Now consider a NOT gate that feeds its output back to itself. (A NOT gate is a simple device that takes one binary input at a time and returns as output the opposite of its input.) At first, suppose the NOT gate receives '0' as an input; it then returns a '1'. After the '1' is fed back to the NOT gate, the gate returns a '0' again, and so on. The NOT gate goes back and forth between outputting a '0' and outputting a '1'. Now map physical states T and T+2 onto '0'; then map T+1 and T+3 onto '1'. *Et voilà*: according to the simple mapping account, the rock implements a NOT gate undergoing the computation represented by '0101'.

According to the counterfactual account, the rock's putative computational implementation is spurious, because the microphysical state transitions do not support the counterfactuals that should obtain between the computational states. If the rock were

put in state *T*, it may or may not transition into *T*+1 depending on whether it is morning or evening and other extraneous factors. Since the rock's microphysical state transitions that map onto the NOT gate's computational state transitions do not support the right counterfactuals, the rock does not implement the NOT gate according to the counterfactual account.

According to the causal and dispositional accounts too, this putative computational implementation is spurious, because the microphysical state transitions are not due to causal or dispositional properties of the rock and its states. *T* does not cause *T*+1, nor does the rock have a disposition to go into *T*+1 when it is in *T*. Rather, the rock changes its state due to the action of the sun. Since the rock's microphysical state transitions that map onto the NOT gate's computational state transitions are not grounded in either the causal or dispositional properties of the rock and its states, the rock does not implement the NOT gate according to the causal and dispositional accounts.

3. The Limits of Mapping Accounts

Mapping accounts fulfill some desiderata but fall short of others.

With respect to *objectivity* (desideratum 1), we already saw that the simple mapping account fails miserably, because it makes it so easy to map (subsets of) microphysical states onto computational states that the only objective aspects of such mappings are the cardinality of the states and the directionality of the state transitions. As a consequence, the claim that a system performs a computation is trivialized.

By restricting which mappings are acceptable, complex mapping accounts do better. Only microphysical state transitions that support counterfactuals (or are causally related, or dispositionally related, or nomologically related) can be mapped onto computational states. As a consequence, physical systems are said to implement only those computations that capture some aspects of their causal, dispositional, or counterfactual supporting structure at some level of abstraction. These are objective matters. Therefore, complex mapping accounts go at least some way towards making physical computation an objective matter of fact.

One major complication remains. The causal (dispositional, nomological) structure of a physical system can be described at different levels of abstraction using different formalisms. For instance, we may describe locomotion at the level of the whole organism, its locomotive systems, the organs that make up the locomotive system, their tissues, the cells that make up the tissues, and so forth. At each of these levels, we may describe the system's causal (dispositional, nomological) structure using a C++ program, a Lisp program, a finite state automaton, or any other computational formalism. The result is a huge number of distinct computations to which the microphysical states of the system can be mapped. This multiplicity of computations seems to render implementation somewhat observer-dependent in an

undesirable way, because which computation is attributed to which system depends on which level of abstraction is chosen and which formalism is employed.

One possible response is to attribute all of those computations to the system. But when we say that a system performs a computation, generally we are able to identify either a unique process at one level of abstraction, described using a privileged formalism, or a limited number of computations at a limited number of levels of abstraction, described using privileged formalisms. For example, we say that our computer is browsing the internet by executing, say, Google Chrome. That's the computation we wish to attribute to the system. (There are also other, lower level computations within the digital computer, but they are still far fewer than the many processes that mapping accounts count as computations and there are precise relations of computational equivalence between those and executing Google Chrome; cf. Chapters 8–11.)

Proponents of mapping accounts may respond in one of two ways. First, they may find a way to single out one or more computations and formalisms as privileged in some way that corresponds to our scientific practices in computer science and cognitive science. Second, they may argue that all of the computations attributed by their account to physical systems are equivalent in some important respect. As far as I know, neither of these responses has been worked out in detail.[3]

Our second desideratum is *explanation*. This is a request for an account of computational explanation—of what is distinctive about it. In this respect, mapping accounts are limited by their direct appeal to the notion of cause, disposition, or counterfactual-supporting state transition. While those notions are surely explanatory, there is nothing in these accounts that distinguishes computational explanation from ordinary causal or dispositional explanation. According to these accounts, there doesn't seem to be anything distinctive about computational explanation relative to causal or dispositional explanation.

Another way to put this point is that mapping accounts have no resources for singling out *computational explanations* properly so called from ordinary *computational models*. A computational model is just an ordinary computational description of a physical system, in which a computer simulates the physical system. But the computation by itself does not *explain* the behavior of the system—the explanation is provided by whatever produces the phenomenon being modeled. By contrast, a computational explanation attributes a genuine computation to the system itself—it maintains that the behavior of the system is itself the result of a computation. Since mapping accounts attribute computations whenever there is a mapping between a computational description and a microphysical description of a system, they attribute computations to a system whenever there is a computational model thereof. Thus, mapping accounts have little or nothing to say about what distinguishes

[3] For a discussion of some potential additional complications, see Brown 2012; Scheutz 2012; Sprevak 2012: and Chalmers 2012.

computational explanation from other kinds of explanation. In fact, mapping accounts deem all physical systems as computational (pancomputationalism).

Our third desideratum is that paradigmatic examples of computing systems should be classified as such. Since mapping accounts are liberal in attributing computations, they have no trouble with this desideratum. The microphysical states (or rather, some subsets thereof) of digital computers, calculators, and any physically implemented Turing machines and finite state automata can be mapped onto computational descriptions as prescribed by mapping accounts. They all turn out to be computing systems, as well they should.

Where mapping accounts have serious trouble is with our fourth desideratum: the wrong things don't compute. The microphysical states (more precisely, subsets thereof) of paradigmatic non-computational processes—such as galaxy formation, the weather, or respiration—can also be mapped onto computational descriptions. The widespread use of computational models in all corners of empirical science is witness to that. As a consequence, mapping accounts count all those processes (and the systems that perform them) as computations. In fact, advocates of mapping accounts often explicitly endorse limited pancomputationalism—every physical system performs some computation (e.g., Putnam 1967b; Chalmers 1996b, 331; Scheutz 1999, 191).

In Chapter 4, I will argue that this *limited* version of pancomputationalism, though weaker than the *unlimited* pancomputationalism mentioned earlier, is still overly inclusive. In computer science and cognitive science, there is an important distinction between systems that compute and systems that do not. To account for this distinction, we will need to restrict the notion of computation further. To do so, we must move beyond mapping accounts of computation.

Our fifth desideratum is that a good account of physical computation gives an account of miscomputation: it should explain what it means for a concrete computation to give the *wrong* output. As their name indicates, mapping accounts are just based on mappings between microphysical states and computational states—such mappings can be created regardless of whether a computation produces the right result. Consider a calculator that yields '0' as the square root of 4. Such an output (and the process that yields it) can be mapped onto a computational description just as well as if the calculator had given the correct output. Therefore, mapping accounts appear to lack the resources for explaining miscomputation.

Our sixth and final desideratum is respecting and illuminating our scientific taxonomy of computing systems, which distinguishes between systems that have different degrees of computational power. In this respect, mapping accounts may go at least part of the way by relying on mappings between physical systems and different kinds of computational descriptions. Some physical systems may have such simple dynamics that they may only support mappings between their microphysical descriptions and logic gates, or small circuits of logic gates. Other physical systems may support mappings between their microphysical descriptions and more

sophisticated computational descriptions, such as finite state automata. So mapping accounts appear to have resources to address our taxonomy desideratum, although I am not aware of any detailed proposals.

The upshot of this chapter is that mapping accounts of computation are a valuable though inadequate effort to solve the problem of implementation. Their solution is that a physical system performs computation *C* just in case there is an appropriately restricted mapping from a subset of the system's microphysical state transitions onto *C*. This is a useful starting point but it's too weak to satisfy our desiderata. Three limitations of mapping accounts stand out: they fail to account for the possibility of miscomputation, they fail to distinguish computational explanation from computational modeling, and they entail pancomputationalism. I will discuss the last two limitations in more detail in Chapter 4. Before that, the next chapter will introduce and evaluate the other traditional type of account: the semantic account.

3

Semantic Accounts

1. No Computation without Representation?

In the previous chapter, I discussed a first type of traditional accounts of physical computation, mapping accounts, and argued that we need something better. In this chapter, I examine the second type of traditional accounts: semantic accounts. In the next two chapters, I will look more closely at some limits of traditional accounts, which will help introduce the mechanistic account.

In our everyday life, we usually employ computations to process meaningful symbols, to extract useful information from them. Semantic accounts of computation turn this practice into a metaphysical doctrine: computation is the processing of representations—more specifically, the processing of appropriate representations in appropriate ways. What all semantic accounts have in common is that, according to them, computation is a semantic notion: there is "no computation without representation" (Fodor 1981, 180).

A common motivation for semantic accounts is that we often individuate, or taxonomize, computational systems and their states (as well as inputs and outputs; I will omit this qualification henceforth) in terms of their semantic content. For example, we may describe a calculator as performing addition over *numbers*, where numbers are usually understood to be the semantic content of the calculator's states.

Taxonomizing computational systems and their states by their semantic content is useful—first and foremost, it allows us to see similarities of function between otherwise radically different systems. For instance, a neural system and a program running on an artificial digital computer may be said to compute shape from shading, even though they may perform otherwise radically different computations.

If this common practice of individuating computational states in terms of their semantic content is turned into a metaphysical doctrine about the nature of computational states—into a criterion for what counts as implementing a computation—it becomes the view that computational states have their content essentially. If the calculator's states represented something *other than* those numbers, they would be *different* computational states. If the calculator's states did *not* represent *at all*, they *wouldn't* be computational states. When defenders of the semantic view talk about individuating computational states semantically, they don't mean this to be merely a useful practice but a way of capturing an important aspect of the nature of

computation—one of the necessary conditions for a physical system to count as performing a given computation.[1]

I call any view that computational states are representations that have their content essentially a *semantic account of computation*. It may be formulated in stronger or weaker forms. In its strongest version, all and only semantic properties of a state are relevant to its computational nature. Probably, no one subscribes to this version. In weaker versions, either *all* semantic properties or *only* semantic properties of a state are relevant to its computational nature. The weakest, and most plausible, versions maintain that a computational state is *partially* individuated (in an essential way) by *some* of its semantic properties (and partially by non-semantic properties).[2] Although supporters of semantic accounts have rarely distinguished between weak and strong versions of their view and have not specified in great detail which semantic properties and which non-semantic properties are relevant to the nature of computational states, this will not matter much here. I will argue against any semantic account.

Relative to the mapping accounts of computation (Chapter 2), semantic accounts may be seen as imposing a further restriction on what counts as a genuine physical computation. Recall that the causal account of computation (as well as counterfactual and dispositional accounts) requires that a physical system possess appropriate causal (counterfactual, dispositional) structure in order to implement a computation. In addition to the restrictions imposed by mapping accounts, semantic accounts

[1] Oron Shagrir has pointed out to me that someone might maintain that computational states are necessarily representations while denying that computational states are individuated by their semantic properties, perhaps on the grounds of an analogy with anomalous monism. According to anomalous monism (Davidson 1970), mental states are necessarily physical even though they are not individuated by their physical properties; by the same token, computational states might be necessarily representational even though they are not individuated by their semantic properties. Since I'm not aware of anyone who has taken this stance explicitly or any reason for taking it, I will not discuss it further.

[2] Here is a very explicit formulation:

> Suppose we start with the notion of a syntactic description of representations. I don't think that this begs any questions because I don't think syntactic individuation requires semantic individuation. Roughly (at least for the case of natural languages) it requires (i) an inventory of basic objects (morphemes, as it might be) and (ii) a recursive definition of WFF (I think all the recursions are on constituents; but I doubt that matters in the present context). Finally, I assume that every computation is a causal sequence of tokenings of such states.
>
> Given that, there are two questions: 1. What distinguishes those of such causal sequences that *constitute computations* from those that don't? Answer, the former preserve *semantic properties* of the strings (paradigmatically, they take one from true inputs to true outputs). This requires that the tokened states have semantic interpretations (since, of course, only what is semantically interpreted can be evaluated for truth). So, in that sense, *the representations in question are individuated by their semantic properties inter alia*. 2. What are the constraints on the causal processes defined on such states? Answer, the effects of being in state *S* must be fully determined by the syntactic properties of *S* (together, of course, with the rules of state transition). That's the sense in which computation is a syntactic process.
>
> So computations are syntactic processes defined over semantically interpreted arrays of representations. (Jerry Fodor, personal correspondence, emphasis added.)

impose a semantic restriction. Only physical states that qualify as representations (of the relevant kind) may be mapped onto computational descriptions, thereby qualifying as computational states. If a state is not representational, it is not computational either.

Semantic accounts are probably the most popular in the philosophy of mind, because they appear to fulfill specific needs. Since minds and digital computers are generally assumed to manipulate (the right kind of) representations, they turn out to compute. Since most other systems are generally assumed not to manipulate (the relevant kind of) representations, they do not compute. Thus, semantic accounts appear to accommodate our desiderata 3 and 4 (Chapter 1, Section 4) about what does and does not count as a computing system. They keep minds and computers in while leaving most everything else out, thereby vindicating the computational theory of cognition as interesting and nontrivial.

Semantic accounts raise three important questions: 1) how representations are to be individuated, 2) what counts as a representation of the relevant kind, and 3) what gives representations their semantic content. I will eventually argue that semantic accounts of computation are incorrect, which means that these semantic issues are orthogonal to the problem of computational implementation. For now, I will look at how they affect computation individuation if we assume a semantic account.

On the individuation of computational states, the main debate divides internalists from externalists. According to externalists, computational vehicles are representations individuated by their *wide cognitive contents* (Burge 1986; Shapiro 1997; Shagrir 2001). Cognitive contents are a type of content ascribed to a system by a cognitive psychological theory; they involve what cognizers are responding to in their environment, or at least ways of representing stimuli. For instance, the cognitive contents of the visual system are visual contents, whereas the cognitive contents of the auditory system are auditory contents. Wide contents are, paradigmatically, the things that the representations refer to, which are typically outside cognitive systems.[3] By contrast, most internalists maintain that computational vehicles are representations individuated by *narrow cognitive contents* (Segal 1991). Narrow contents are, roughly speaking, semantic contents defined in terms of intrinsic properties of the system.

To illustrate the dispute, consider two physically identical cognitive systems *A* and *B*. Among the representations processed by *A* is representation *S*. *A* produces instances of *S* whenever *A* is in front of bodies of water, when *A* is thinking of water, and when *A* is forming plans to interact with water. In short, representation *S* appears to refer to water. Every time *A* processes *S*, system *B* processes representation *S'*, which is physically identical to *S*. But system *B* lives in an environment different from *A*'s environment. Whenever *A* is surrounded by water, *B* is surrounded by

[3] Alternatively, wide contents might be Fregean modes of presentation that determine referents but are individuated more finely than by what they refer to.

twater. Twater is a substance superficially indistinguishable from water but in fact physically different from it. Thus, representation *S'* appears to refer to twater (cf. Putnam 1975b). So, we are assuming that *A* and *B* live in relevantly different environments, such that *S* appears to refer to water while *S'* appears to refer to twater. We are also assuming that *A* is processing *S* in the same way that *B* is processing *S'*. There is no intrinsic physical difference between *A* and *B*.

According to externalists, when *A* is processing *S* and *B* is processing *S'* they are in computational states of *different* types. According to internalists, *A* and *B* are in computational states of the *same* type. In other words, externalists maintain that computational states are individuated in part by their reference, which is determined at least in part independently of the intrinsic physical properties of computing systems. By contrast, internalists maintain that computational states are individuated in a way that supervenes solely on the intrinsic physical properties of computational systems.

So far, externalists and internalists agree on one thing: computational states are individuated by *cognitive* contents. This assumption can be resisted without abandoning semantic accounts of computation. According to Egan (1999), computational states are not individuated by cognitive contents of any kind, either wide or narrow. Rather, they are individuated by their *mathematical* contents—that is, mathematical functions and objects ascribed as semantic contents to the computational states by a computational theory of the system. Since mathematical contents are the same across physical duplicates, Egan maintains that her mathematical contents are a kind of *narrow* content—she is a kind of internalist. I will discuss this point in Chapter 7, Section 2.8.

Let us now turn to what counts as a representation. This debate is less clearly delineated. According to some authors, only structures that have a language-like combinatorial syntax, which supports a compositional semantics, count as computational vehicles, and only manipulations that respect the semantic properties of such structures count as computations (Fodor 1975; Pylyshyn 1984). This suggestion flies in the face of computability theory, which imposes no such requirement on what counts as a computational vehicle. Other authors are more inclusive on what representational manipulations count as computations, but they have not been especially successful in drawing the line between computational and non-computational processes. Few people would include all manipulations of representations—including, say, painting a picture and recording a speech—as computations, but there is no consensus on where to draw the boundary between representational manipulations that count as computations and representational manipulations that do not.

A third question is what gives representations their semantic content. There are three families of views. Instrumentalists believe that ascribing semantic content to things is just heuristically useful for prediction or explanation; semantic properties are not real properties of computational states (e.g., Dennett 1987; Egan 2010, 2014). Realists who are not naturalists believe semantic properties are real properties of

computational states but are irreducible to non-semantic properties. Finally, realists who are also naturalists believe semantic properties are both real and reducible to non-semantic properties, though they disagree on how to reduce them (e.g., Fodor 2008; Harman 1987).

Semantic accounts of computation are closely related to the common view that computation is information processing. This idea is less clear than it may seem, because there are several notions of information. The connection between information processing and computation is different depending on which notion of information is at stake. For now I will briefly disambiguate the view that computation is information processing based on four important notions of information (I will address the relation between computation and information processing in a more systematic way in Chapter 14).

1. Information in the sense of thermodynamics is closely related to thermodynamic entropy. Entropy is a property of every physical system. Thermodynamic entropy is, roughly, a measure of an observer's uncertainty about the microscopic state of a system after she considers the observable macroscopic properties of the system. The study of the thermodynamics of computation is a lively field with many implications in the foundations of physics (Leff and Rex 2003). In this thermodynamic sense of 'information', any difference between two distinguishable states of a system may be said to carry information. Computation may well be said to be information processing in this sense (cf. Milkowski 2013), but this has little to do with semantics properly so called. Nevertheless, the connections between thermodynamics, computation, and information theory are one possible inspiration for the view that every physical system is a computing system (Chapter 4).
2. Information in the sense of communication theory is a measure of the average likelihood that a given message is transmitted between a source and a receiver (Shannon and Weaver 1949). But information in the sense of communication theory is insufficient for semantics in the sense that interests us.
3. Information in one semantic sense is approximately the same as "natural meaning" (Grice 1957). A signal carries information in this sense just in case it reliably correlates with a source (Dretske 1981). The view that computation is information processing in this sense is prima facie implausible, because many computations—such as arithmetical calculations carried out on digital computers—do not seem to carry any natural meaning. Nevertheless, this notion of semantic information is relevant here because it has been used by some theorists to ground an account of representation (Dretske 1981; Fodor 2008).
4. Information in another semantic sense is just ordinary semantic content or "non-natural meaning" (Grice 1957). This is the kind of semantic content that most philosophers discuss. The view that computation is information processing in this sense is similar to a generic semantic account of computation.

Although semantic accounts of computation appear to fulfill the needs of philosophers of mind, they are less suited to make sense of some scientific practices pertaining to computation. Most pertinently, representation is not presupposed by the notion of computation employed in at least some areas of cognitive science as well as computability theory and computer science—the very sciences that gave rise to the notion of computation at the origin of the computational theory of cognition. I will defend this thesis later in this and later chapters. If this is correct, semantic accounts are not even adequate to the needs of philosophers of mind—at least those philosophers of mind who wish to make sense of the analogy between minds and the systems designed and studied by computer scientists and computability theorists.

For now, however, I will raise an objection to semantic accounts of computation that arises from the inside, so to speak. The objection is that individuating computations semantically presupposes a way to individuate computations non-semantically, and non-semantic individuation is necessary to the mechanistic explanation of the system's behavior. Therefore, we need a non-semantic way of individuating computations and a non-semantic account of concrete computation. This will motivate the search for an adequate non-semantic account of computation, to wit, the mechanistic account.

Before proceeding, two caveats are in order.

First caveat: whether computation has a semantic nature should not be confused with the issue of which properties of computational states are causally efficacious within a computation.

Here, the received view is that computational states are causally efficacious by virtue of properties that are *not* semantic. According to this view, which may be called the *non-semantic view of computational causation*, computational processes are "insensitive" or "indifferent" to the content of computational states; rather, they are sensitive only to (some) non-semantic properties of computational states. The properties to which computational processes are supposedly sensitive are often labeled as "formal" or "syntactic."[4]

The two issues—whether computation requires representation and whether computation is sensitive to semantic properties—are orthogonal to one another. Computation may be insensitive to semantic properties while having a (partially) semantic nature, as, e.g., Fodor (1980) argues. Or computation may be insensitive to semantic properties while having a non-semantic nature, as, e.g., Stich (1983) and Piccinini (2008) argue. Or computation may be sensitive to semantic properties while having a (partially) semantic nature, as Dietrich (1989), Peacocke (1994a, 1999), and Shagrir (1999) argue. Finally, computation may be sensitive to semantic properties, at

[4] The *locus classicus* is Fodor 1980: "computational processes are both symbolic and formal . . . What makes syntactic operations a species of formal operations is that being syntactic is a way of not being semantic" (Fodor 1980, 64). See also Newell 1980.

least in some cases, while having a wholly non-semantic nature, as Rescorla (2012, 2014a) argues.

Contrary to my former self (Piccinini 2008), I now side with Rescorla for the following reason. Whether computation is sensitive to semantic properties depends on what "sensitive" and "semantic property" means and what kind of semantic property is involved. Surely differences in wide meaning alone (like the difference between the meaning of "water" on earth and the meaning of "water" on twin earth) cannot make a difference to computation (contrary to what Rescorla implies) for lack of a causal mechanism connecting such differences to the computation. Insofar as supporters of the non-semantic view of computational causation are after this point, I agree with them.

But differences in certain narrow meanings, like the difference between an instruction that means *add* and an instruction that means *subtract* (where such meanings are cashed out in terms of machine operations on strings; cf. Chapter 9), make a big difference to a computation. In that sense, computations are sensitive to semantic properties. There are also semantic properties that relate many computing systems to their environment—they relate internal representations to their referents—and are encoded in the vehicles in such a way that they make a difference to the system's computation, as when an embedded system computes over vehicles that represent its environment. In that sense, too, computation is sensitive to semantics.

The issue of semantic properties and their role in computational causation is too complex a topic to address it here. A more adequate treatment will have to wait for another occasion. But it's important to realize that at least some computations are sensitive to certain kinds of semantic properties even though they can be individuated non-semantically.

Second caveat: whether computation has a semantic nature should not be confused with whether computation (partially or wholly) explains content or intentionality.

Some critics of computational theories of mind maintain that being a computational state contributes nothing to original, or non-derived, intentionality (e.g., Searle 1980; Horst 1996). They assume that computational states have a non-semantic nature and argue that (non-semantically individuated) computational states cannot have original intentionality. Notice that these authors do not offer a fully worked out alternative to semantic accounts of computation; Searle (1992) even argues that there is no observer-independent way to individuate computational states. In response, many supporters of the computational theory of cognition have retained the view that computational states are, at least in part, semantic in nature.

Although I reject semantic accounts of computation, I will remain neutral on whether being computational contributes to explain original intentionality. That depends on what original intentionality amounts to—something on which there is little consensus. The mechanistic account of computation that I defend does not entail that computational states have no semantic content—they may or may not

have content. Nor does it entail that computational states cannot have original content or intentionality—perhaps some aspects of original content supervene on computational properties. Nor does it entail that computational states cannot be individuated, in our informal practices, by reference to their semantic content. As we just saw in my previous caveat, it doesn't even entail that computation is always insensitive to the semantic properties of computational states. What it does entail is that a computation can be fully individuated without any appeal to semantic content: there is computation without representation.

2. Semantic Individuation Presupposes Non-Semantic Individuation

Semantic accounts of computation hold that computations are individuated in an essential way by their semantic properties. This requires an account of semantic content, which in turn can be used to individuate computations. There are three main (families of) accounts of content. I will go over each of them and argue that individuating computations in terms of each of them presupposes a non-semantic individuation of computations.

The first account is Functional (or conceptual, or inferential) Role Semantics (FRS). FRS sees semantic content as (partially) reducible to the functional relations between a system's internal representations, inputs, and outputs (Harman 1973; Field 1978; Churchland 1979, 1989; Loar 1981; Block 1986). In other words, FRS assigns semantic content to some internal states of a system—the internal representations—based on the way they functionally (or conceptually, or inferentially) interact with other internal states, inputs, and outputs. For example, the content of my belief that my daughter is cute has to do with my ability to produce that thought in the presence of my daughter's cuteness, to infer from it that someone is cute or that my daughter has a property, and so on. Historically, FRS was the first naturalistic theory of content proposed within contemporary philosophy of mind (Sellars 1954, 1956, 1961, 1967, 1974).

Theorists who apply FRS to computational systems typically maintain that the functional (or conceptual, or inferential) relations between the internal representations, inputs, and outputs can be cashed out as computational relations (Harman 1973; Block 1986; Churchland 1989). In other words, they maintain that the functional (or conceptual, or inferential) relations that give rise to semantic content are (at least in part) computational relations. Therefore, since our topic is the nature of computation, I will assume that the functional (conceptual, inferential) relations that give rise to semantic content according to FRS are computational. But then, FRS cannot individuate computations semantically on pain of circularity. If semantic content reduces to functional relations and functional relations are computational, then computational relations must be individuated non-semantically or else the

account of computation becomes circular. Thus, FRS applied to computational systems presupposes a non-semantic way of individuating computational states.[5]

The second account of content is interpretational semantics, which sees semantic content as a property attributed to a system by an observer for the purpose of predicting behavior (Dennett 1969, 1978, 1987; Cummins 1983, 1989).[6] According to interpretational semantics, semantic content is not an objective property of a system but a way of describing a system that is convenient from a certain point of view, or stance. Presumably, different semantic contents may be ascribed to the same system by different observers for different purposes in somewhat arbitrary ways, which makes semantic properties non-objective.[7] Given that semantic content is non-objective in this way, interpretational semantics is a form of anti-realism about semantic content.

When interpretational semantics is conjoined with a semantic account of computation, its anti-realism about semantic content carries over into anti-realism about computation. For according to semantic accounts of computation, something is computational just in case it manipulates (certain kinds of) representations (in appropriate ways), but according to interpretational semantics something manipulates representations just in case observers choose to see the system as doing so. Thus, according to the conjunction of the two views, a system is computational just in case observers choose to see the system as computational.

The result of such a conjunction does not do justice to computer science and engineering. That my laptop is performing a computation seems to be an objective fact (as opposed to a fact that depends on how an observer chooses to interpret my laptop). What's more, my laptop appears to have perfectly objective computational properties: it's computationally universal (up to memory limitations), it stores programs that control its processor, it's running Microsoft Word, etc. Since—according to interpretational semantics—semantic properties are non-objective, anyone who endorses interpretational semantics and wishes to do justice to the objective facts of computer science needs a non-semantic way of individuating computations.

The third account of content is Informational or Teleological Semantics (ITS), which sees content as reducible to a combination of causal or informational relations between individual mental states and what they represent (Dretske 1981, 1986; Millikan 1984, 1993; Fodor 1987, 1990, 1998). What the specific causal or informational relations

[5] Another way to avoid circularity would be to give a non-computational account of the functional (conceptual, inferential) relations that give rise to semantic properties according to FRS. Since I'm not aware of any naturalistic proposal along these lines, I will set this possibility aside.

[6] Dennett and Cummins may or may not recognize themselves in this short sketch, which doesn't do justice to the sophistication of their work and the differences between them. For present purposes their views are close enough to the position I sketch in the text.

[7] If this is correct, then interpretational semantics may be too weak to underwrite semantic accounts of computation as I formulated them, though it may support a weakened formulation of semantic accounts. Since my purpose is to reject semantic accounts of computation, I will ignore this worry.

are, and even whether they are synchronic or span long stretches of history (as per evolution-based versions of ITS), does not matter for present purposes. What matters is that, roughly speaking, according to all versions of ITS, a state represents *X* just in case it stands in appropriate causal or informational relations to *X*.

While ITS may work at least to some extent for embedded systems such as brains and some artificial computers, which stand in the appropriate causal or informational relations to what they represent, ITS does not work for many artificial computers, i.e., the non-embedded ones. Ordinary computing artifacts need not stand in any causal or informational relations to anything that they represent. For that matter, ordinary computing artifacts need not represent anything. We may take a brand new computer, which has not interacted with much beyond the factory that produced it, and program it to alphabetize meaningless strings of letters. The result is a computation that stands in no causal or informational relations to anything that would assign semantic content to its internal states. Yet the computer performs computations all the same.

Someone may attempt to get around this by positing that non-embedded computing artifacts inherit their semantic content from their designers or users. Notice that this suggestion goes beyond ITS—it is a form of hybrid semantic theory that combines ITS for embedded systems with an intention-based theory for non-embedded computing artifacts. There is still a fatal problem. We can design and use computing artifacts while intending them to have *no* semantics whatsoever. For instance, we can program a computer to manipulate strings of symbols written in an arbitrary alphabet of our invention in an arbitrary way, without assigning any meanings to the strings. Therefore, again, anyone who endorses ITS (or a hybrid ITS plus intention-based-semantics view) and wishes to include ordinary (non-embedded) computers among computing systems needs a non-semantic way of individuating computations.

These options seem to exhaust the possibilities that are open to the supporter of a semantic account of computation: either semantic content comes from the computations themselves (FRS), or it comes from some non-computational natural properties of the content-bearing states (ITS), or it is in the eye of the beholder (interpretational semantics). Under any of these options, at least some computational states must be individuated non-semantically.

Finally, someone might endorse a kind of primitivism about content, according to which we can simply posit contents without attempting anything like a reductive analysis of what it is for states to have content (cf. Burge 2010). Based on this account, we can attribute contents to computational states without having to say anything more about how those states acquire their contents or what the contents are in nonsemantic terms. This is not an account of content, of course; it's an insistence on the legitimacy of positing contents in the absence of an account.

There are at least two problems with primitivism about content. First, contents are not the kind of thing that can be posited without a reductive account; they are not basic enough; they are not similar enough to fermions and bosons (cf., e.g., Fodor

1987). Second, positing contents without explaining how they are acquired and individuating computational states solely in terms of such primitive contents flies in the face of our scientific practices. This is especially obvious in computer science but also in the methodologically rigorous areas of cognitive science. Rigorous cognitive scientists do not limit themselves to positing semantically individuated computational states and processes. Doing so would hardly be an advance over traditional, non-mechanistic psychology—the kind of psychology that was rightly criticized by behaviorists.

In addition to positing computations, rigorous cognitive scientists construct computational models that go with their computational posits and demonstrate the feasibility of the posited computations. Computational models are what makes cognitive science mechanistically legitimate in response to behaviorist complaints that representations are illegitimate. Building computational models requires more than positing semantic contents; it requires nonsemantically individuated vehicles that carry those contents and are mechanistically manipulated within the computational models. So primitivism about semantic content is insufficient for individuating computations within a methodologically rigorous science; therefore, I will set it aside.

3. Arguments for Semantic Accounts of Computation

Before abandoning semantic accounts of computation, I will address the three main arguments that have been offered to support them. I will reply to each of them in turn.

The first argument pertains directly to computing systems and their states. It goes as follows:

Argument from the identity of computed functions

Computing systems and their states are individuated by the functions they compute.

Functions are individuated semantically, by the ordered couples <domain element, range element> denoted by the inputs and outputs of the computation.

Therefore, computing systems and their states are individuated semantically.

Variants of this argument may be found in the writing of several authors (Dietrich 1989; Smith 1996; Shagrir 1997, 1999; Peacocke 1999; Rescorla 2013 offers an argument roughly along these lines for *some* [not all] computing systems).[8]

[8] Cf. Dietrich and Peacocke:

> [A] correct account of computation requires us to attribute content to computational processes in order to explain which functions are being computed (Dietrich 1989, 119).
>
> There is no such thing as a purely formal determination of a mathematical function (Peacocke 1999, 199).

The argument from the identity of functions ignores that functions may be individuated in *two* ways. The argument from the identity of functions appeals to the set of ordered pairs <domain element, range element> denoted by the inputs and outputs of the computation (for example {<1, 10>, <10, 11>, . . . }, where '1', '10', '11', . . . denote the numbers 1, 2, 3, . . .). But functions can also be individuated by the set of ordered pairs <input type, output type>, where input and output types are the vehicles that enter and exit the computing system (for example {<'1', '10'>, <'10', '11'>, . . . }, where "1", "10", "11", . . . denote inscriptions of types '1', '10', '11', . . .). In other words, functions can be defined either over entities such as numbers, which may be the content of computational vehicles, or over entities such as strings of (suitably typed) letters from an alphabet, which are the computational vehicles themselves described at an appropriate level of abstraction. Both ways of individuating functions are important and useful for many purposes. Both can be used to describe what is computed by a system. By ignoring that functions can be individuated non-semantically, the argument from the identity of computed functions fails to establish that computation presupposes representation.

A closely related question is which of the two ways of individuating functions is most relevant to the nature of physical computation, which is to say, to solving the problem of computational implementation. The most relevant function individuation is the one based on vehicles. The other, semantic, way of individuating functions may be useful for many other purposes, including explaining why people build computers the way they do and why they use them, but it is insufficient and sometimes inappropriate for individuating a physical computation and explaining how it is performed. There are three reasons for this. First, per Section 2 above, the semantic individuation of functions presupposes their non-semantic individuation. The only way we can pick out the entities denoted by the inputs and outputs of a computation is by using a notation, which is individuated non-semantically. Second, some physical computations do not represent anything—ergo, they must be individuated purely non-semantically. Third, the physical entities physically manipulated by a physical computation are the vehicles, and the physical computation is defined in terms of the entities it manipulates and how it manipulates them. Therefore, we cannot fully individuate a physical computation solely by indicating what it represents (if it represents anything)—we must also indicate what vehicles it manipulates and how it manipulates them.

The last point deserves expansion. Even when a computation does manipulate representations, it is easy to see that a semantically individuated function is *insufficient* for fully individuating a physical computation and the system that performs it. This is because individuating a computation by the function it computes does not individuate physical computations as finely as we need when explaining the capacities of computing systems. Any domain of objects (e.g., numbers) may be represented in indefinitely many ways (i.e., notations). Any computable function may be computed by indefinitely many algorithms. Any algorithm may be implemented by

indefinitely many programs written in indefinitely many programming languages. Finally, any program may be executed by indefinitely many computer architectures. Even within the same programming language or computer architecture, typically there are different ways of implementing the same algorithm. So a semantically individuated function does not individuate computing systems and their states as finely as we need. If we were to individuate a physical computation solely by the function it denotes, we'd get the paradoxical consequence that systems that have different architectures, use different programming languages, and execute different programs that implement different algorithms (perhaps of different computational complexity) and manipulate different notations, are ascribed the same computation only because they compute the same semantically individuated function. To avoid this, individuating physical computations by the functions they compute is not enough. We should also allow other functional and structural (non-semantic) aspects of the computation, such as the program and the architecture, to contribute to the individuation of a physical computation and the system that performs it. So even when a semantically individuated function is pertinent to individuating a physical computation, it is *insufficient* for its complete individuation. Since we already established that semantic individuation can be dispensed with in favor of non-semantic individuation, we have now established that semantic individuation is neither necessary nor sufficient for individuating physical computations.

Given a description of a computing system that individuates the function being computed in terms of input and output vehicles, one may ask how it is that that system also computes the function <domain element, range element>, defined over whatever the vehicles represent (assuming they represent something). In order to explain this, what is needed is a further fact: that the inputs and outputs of the computation *denote* the elements of the domain and range of the function. This is a semantic fact, which relates non-semantically individuated inputs and outputs to their semantic content. Stating this semantic fact requires that we individuate the inputs and outputs of the computation independently of their denotations. Individuating inputs and outputs independently of their content is a necessary condition for stating this semantic fact. So, again, a non-semantic individuation of inputs and outputs is a prerequisite for talking about their content.

Question (Michael Rescorla, personal communication): Why can't we just say something like, the computational system enters into an internal state that refers to, say, the number 2. Or: the computational system stores in register A a symbol that refers to the number 2. These descriptions individuate computational states through their denotations, but they don't give any substantive non-semantic individuation of the computational state. I can see why one might want a substantive non-semantic individuation, but it doesn't seem that one needs it simply to state the content of the relevant states. By analogy, in folk psychology we talk about the content of mental states all the time without bringing in any level of non-semantic individuation.

Answer: There is a difference between the two descriptions above. The first simply refers to an otherwise unidentified internal state; the second refers to a specific

register, which is an architectural description. So the second is a more complete mechanistic description than the first. Both are very sketchy. We can give such sketchy descriptions but they don't go very far towards explaining how the system works. They aren't enough to build an artificial system that performs the same computation. An adequate explanation includes the notation being used, the algorithm followed, the architecture that executes the algorithm, etc. An adequate explanation is fully mechanistic (Chapter 5).

The second argument for semantic accounts of computation appeals to computational explanations of mental processes:

Argument from the identity of mental states

Computational states and processes are posited in explanations of mental states and processes (e.g., inference).

Mental states and processes have their content essentially.

Therefore, computational states and processes are individuated by the semantic properties of the mental states and processes they explain and have such contents essentially.

Variants of the argument from the identity of mental states may be found in many places (the most explicit include Fodor 1975; Pylyshyn 1984; Burge 1986; Peacocke 1994a, 1999; Wilson 2004).[9]

Premise 1 is uncontroversial; it simply takes notice that some scientific theories explain mental states and processes computationally. Premise 2 has been challenged (e.g., by Stich 1983), but for the sake of the argument I will ignore any concerns about whether mental states have content and whether they have such content essentially.

As appealing as the argument from the identity of mental states may sound, it is a non sequitur. As Egan (1995) notes, the only way the conclusion can be derived from the premises is by assuming that *explanantia* must be individuated by the same properties that individuate their *explananda*. This assumption is at odds with our explanatory practices. The relevant type of explanation is constitutive explanation,

[9] Cf. Burge and Peacocke:

> There is no other way to treat the visual system as solving the problem that the [computational] theory sees it as solving than by attributing intentional states (Burge 1986, 28–9).
>
> One of the tasks of a subpersonal computational psychology is to explain how individuals come to have beliefs, desires, perceptions, and other personal-level content-involving properties. If the content of personal-level states is externally individuated, then the contents mentioned in a subpersonal psychology that is explanatory of those personal states must also be externally individuated. One cannot fully explain the presence of an externally individuated state by citing only states that are internally individuated. On an externalist conception of subpersonal psychology, a content-involving computation commonly consists in the explanation of some externally individuated states by other externally individuated states (Peacocke 1994b, 224; Peacocke assumes that external individuation is a form of semantic individuation, which in principle need not be the case; cf. Chapter 7, Section 2.8).

whereby a property or capacity of a system is explained in terms of the functions and organization of its constituents. For example, consider the explanation of digestion. The *explanandum*, a certain type of state change of some organic substances, is individuated by the chemical properties of substances before, during, and after they enter the stomach. Its *explanans*, which involves secretions from certain glands in combination with the stomach's movements, is individuated by the activities of the stomach, its glands, and their secretions. This example shows that the individuation of *explanantia* independently of their *explananda* is an aspect of our explanatory practices. There is no reason to believe that this should fail to obtain in the case of explanations of mental states and processes. And without the assumption that *explanantia* must be individuated by the same properties that individuate their explananda, the argument from the identity of mental states doesn't go through.[10]

A third argument for a weakened semantic account of computation is due to Oron Shagrir (2001; cf. also Rescorla 2013). Here is a formulation using my terminology:

Argument from the multiplicity of computations

(1) The same computing system M implements multiple (non-semantically individuated) computations $C_1, \ldots C_n$ at the same time.

(2) For any task that M may perform, there is a unique $C_i \in \{C_1, \ldots C_n\}$, such that C_i alone explains M's performance of the task, and C_i is determined by the task performed by M in any given context.

(3) Tasks are individuated semantically.

---

(4) Therefore, in any given context, C_i is individuated semantically (in part).

(5) Therefore, in so far as computations explain the performance of a task by a system in any given context, they are individuated semantically (in part).

Premise (1) appeals to the fact that the physical inputs, outputs, and internal states of a system can be grouped together in different ways, so that different computational descriptions apply to them. For instance, imagine a simple device that takes two input digits, yields one output digit, and whose inputs and outputs may take three possible values (which may be called 0, ½, and 1). And suppose that the outputs are related to the inputs as follows:

Inputs → Output
0, 0 → 0
0, ½ → ½

[10] For a similar reply to the argument from the identity of mental states, see Egan 1995, 57ff. At this junction, Michael Rescorla (personal communication) reminds us that the computations attributed by certain scientific theories are characterized in representational terms. Therefore, at least in some cases, science posits computational states and processes that are individuated in content-involving terms. I agree but remind you, in turn, that the dispute is *not* about whether some scientific theories posit computations that can be individuated by their content (to which the answer is yes); it's whether such computations have a semantic nature, that is, whether computational states have their content essentially (they don't; cf. Section 2).

½, 0 → ½
0, 1 → ½
1, 0 → ½
½, ½ → ½
½, 1 → ½
1, ½ → ½
1, 1 → 1

The above is a bona fide computational description of our device. Under this description, the device performs an averaging task of sorts. Since this averaging task exploits all of the functionally distinct inputs and outputs of the device, I will refer to it as the *maximal* task of the device, and to the corresponding computation as the maximal computation.

If we group together and relabel our inputs and outputs, we may find other computational descriptions. For instance, we may group '0' and '½' together and call both of them 0, or we may group '½' and '1' together and call both of them 1. The first grouping turns our device into what is ordinarily called an AND gate; the second grouping turns it into an OR gate. As a consequence of these groupings and relabelings, our device implements several computations at once: our original averaging, the AND operation, the OR operation, etc. These operations form our set of computations $C_1, \ldots C_n$ mentioned in premise (1), all of which are implemented by our device at the same time.[11]

In principle, our device could be used to perform different tasks, each of which corresponds to one of the computations implemented by the device. It could be used to perform its maximal task (averaging) as well as a number of non-maximal tasks (conjunction, disjunction, etc.). But in any given context, our device may be used to perform only one specific task. For example, our device might be part of a larger device, which uses it to perform conjunctions. Premise (2) points out that in order to explain how our device performs a given task—say, conjunction—we must appeal to the relevant computational description, namely AND. So, the task performed by a computing system in a given context determines which computational description is explanatory in that context.

Although premises (1) and (2) are true and suggestive, they probably make little difference in most scientific contexts. Usually, computing systems like our simple averaging device are employed to perform their maximal task. In engineering applications, it would be unnecessarily costly and cumbersome to build a device

[11] Shagrir suggests another way in which a system might implement multiple computations, namely, by letting different sets of properties (e.g., voltage and temperature) implement different computations (Shagrir 2001, 375). But then either the different sets of properties correlate, in which case the two computations are the same, or they don't, in which case we simply have two physical processes implementing two different computations within the same system. (A system may perform many activities at the same time thanks to different internal processes, which may or may not have some parts in common; in the case of this example, both activities are computations and both processes are computational.)

with inputs and outputs of three kinds but use it to perform tasks that require inputs and outputs of only two kinds. In nature, it is unlikely that natural selection would generate a process that can differentiate between more possible inputs and outputs than it needs to in order to carry out its task. Although it is common for the same naturally occurring mechanism to perform different tasks, usually each task is subserved by a different process within the mechanism. And although some natural computational processes may have evolved from ancestors that required to differentiate more inputs and outputs than the current process, this seems unlikely to be the most common occurrence. So the possibilities mentioned in premises (1) and (2) may have little practical significance. Nevertheless, theoretically it is good to know what they entail about the individuation of computation, so let us examine the rest of the argument.

Premise (3) says that tasks are semantically individuated. For instance, one of our device's tasks, averaging, is defined over quantities, which are the implicit referents of the inputs and outputs of the device. Since, by (2), tasks determine which computational description is explanatory in a given context, (4) concludes that the computational identity of a device in a given context is partially determined by semantic properties. In other words, the computation that is explanatory in any given context is partially individuated semantically. Given that the argument does not depend on the specific device or computational description, (5) is a universal generalization of (4).

Before discussing the merits of the argument from the identity of computational tasks, notice that its conclusion is weaker than a full-blown semantic account of computation. For the argument begins by conceding that the (multiple) computations implemented by a device are individuated non-semantically. Semantic constraints play a role only in determining which of those computations is explanatory in a given context. As I pointed out above, it is likely that in most contexts of scientific interest, computing systems perform their maximal task, so that semantic constraints are unnecessary to determine which computation is explanatory. If this is correct, and if the argument from the multiplicity of computations is sound, then semantic constraints will play a role in few, if any, practically significant contexts. It remains to be seen whether the argument is sound.

The problem is with premise (3), and it is analogous to the problem with premise (2) in the argument from the identity of functions. The task of a computing system is to compute a certain function. As I pointed out above, functions may be individuated semantically, and therefore so may tasks. As I also pointed out above, functions may be individuated non-semantically too, and therefore so may tasks. For the same reasons given in the case of functions, the task description that is most basic in individuating computing systems and their processes is non-semantic.

Shagrir's reason for (3) seems to be that he assumes that non-semantic individuation is based on functional properties, and functional properties are narrow properties. If functional properties are narrow—that is, they supervene on what is

contained solely within the boundaries of the system, without reaching into its environment—then they are insufficient to determine which task a mechanism is performing within a context, and thus which computation is explanatory in that context. It goes to Shagrir's credit that he showed this to us. But the solution need not be an individuation of computations based on content, for there is also the possibility of a wide construal of functional properties.

In later chapters, I will follow Harman (1987, 1988) in construing functional properties as wide. Briefly, a functional property is wide just in case it supervenes on both what is inside the boundaries of the system and some portion of the system's environment (which may be a larger system that contains it). On this view, a functional property of a system depends not only on what happens within the system but also on how its environment is and how the system and its environment interact. Because of this, we cannot discover functional properties by studying the system in isolation—we must observe some portion of its environment and how the system interacts with its environment. For example, the heart has the function of circulating blood through the body by pumping it from the veins into the arteries. This function requires that blood vessels form a closed system—something that the heart has nothing to do with; the function cannot even be stated without making reference to entities and processes that occur outside the heart (the body, the circulation of the blood, the veins, the arteries). Circulating the blood is a wide function, and fulfilling it is a wide functional property.

Shagrir gives no reason to prefer a semantic individuation of computations to a wide functional individuation. Provided that the interaction between a mechanism and its context plays a role in individuating its functional (including computational) properties, a (non-semantic) functional individuation of computational states is sufficient to determine which task is being performed by a system, and therefore which computation is explanatory in a context.

In order to know which of the computations that are implemented by a computing mechanism is explanatory in a context, we need to know the relevant relations between computations and contexts. Therefore, we cannot determine which computation is explanatory within a context without looking outside the mechanism. I agree with Shagrir about this, and also about the fact that interpreting computations—describing computations semantically—is one way to relate computations to context. But it's not the only way: computations have effects on, and are affected by, their context. By looking at which effects of which computations are functionally significant within a context, we can identify the computation that is explanatory within that context.

Going back to our example, suppose our device is a component of a larger mechanism. By looking at whether the containing mechanism responds differentially to a '0', '½', and '1' or responds identically to two of them, we can determine which

computational description is explanatory without needing to invoke any semantic properties of the computations.[12]

This completes my assessment of arguments for semantic accounts of computation. The upshot is that computation does not require representation. Before ending this chapter by summarizing the pros and cons of semantic accounts, I will briefly discuss an attempt to account for computation syntactically rather than semantically.

4. The Syntactic Account

Some philosophers have denied that there are any semantic properties (Quine 1960; Stich 1983). If there are no semantic properties, semantic accounts of computation are a nonstarter. This, combined with the limitations of mapping accounts (Chapter 2), motivated some authors to search for a non-semantic account of computation that is more restrictive than mapping accounts. This may be called *syntactic account of computation*, because it appeals to the notion of syntax derived from logic and linguistics.

As we saw, semantic accounts need to specify which representations are relevant to computation. One view is that the relevant representations are language-like, that is, they have the kind of syntactic structure exhibited by sentences in a language. Computation, then, is the manipulation of language-like representations in a way that is sensitive to their syntactic structure and preserves their semantic properties (Fodor 1975).

As we also saw, using the notion of representation in an account of computation ends up presupposing a non-semantic way of individuating computations. If computation could be accounted for without appealing to representation, all would be well. One way to do so is to maintain that computation simply is the manipulation of language-like structures in accordance with their syntactic properties, leaving semantics by the wayside. The structures being manipulated are assumed to be language-like only in that they have syntactic properties—they need not have any semantics. In this syntactic account of computation, the notion of representation is not used at all.

The syntactic account may be seen as adding a restriction on acceptable mappings between computational descriptions and physical systems that replaces the semantic restriction proposed by semantic accounts. Instead of a semantic restriction, the syntactic account imposes a syntactic restriction: only physical states that qualify as

[12] Rescorla (2013) describes an example slightly different from Shagrir's but the moral is the same. Rescorla considers two machines that have the same local intrinsic properties but are embedded within two distinct communities. The two communities interpret their machines differently by employing base 10 and base 13 notations, respectively. Therefore, under their communities' distinct interpretations, the two machines perform different semantically individuated computations. Rescorla concludes against non-semantic accounts of computation, which supposedly cannot distinguish between the two computations. On the contrary, as I point out in the main text, the non-individualistic, non-semantic account of computation that I defend has enough resources to distinguish between Rescorla's two computations in terms of the different responses by the two communities to processes that are intrinsically exactly similar.

syntactic may be mapped onto computational descriptions, thereby qualifying as computational states. If a state lacks syntactic structure, it is not computational.

What remains to be seen is what counts as a syntactic state. An account of syntax in the physical world is due to Stephen Stich (1983, 150–7). Although Stich does not use the term 'computation', his account of syntax is aimed at grounding a syntactic account of mental states and processes. Stich's syntactic theory of mind is, in turn, his interpretation of the computational theories proposed by cognitive scientists—in competition with the semantic interpretation proposed by Fodor et al. Since Stich's account of syntax is ultimately aimed at grounding computational theories of cognition, Stich's account of syntax also provides an (implicit) syntactic account of computation.

According to Stich, roughly speaking, a physical system contains syntactically structured objects when two conditions are satisfied. First, there is a mapping between the behaviorally relevant physical states of the system and a class of syntactic types, which are specified by a grammar that defines how complex syntactic types can be formed out of (finitely many) primitive syntactic types. Second, the behavior of the system is explained by a theory whose generalizations are formulated in terms of formal relations between the syntactic types that map onto the physical states of the system.

The syntactic account of computation is not very popular. A common objection is that it seems difficult to give an account of primitive syntactic types that does not presuppose a prior semantic individuation of the types (Crane 1990; Jacquette 1991; Bontly 1998). According to this line of thought, something can be a token of a syntactic type only relative to a language in which that token has content. In fact, it is common to make sense of syntax by construing it as a way to combine symbols, that is, semantically interpreted constituents. If syntax is construed in this way, it presupposes semantics. If so, the syntactic account of computation either is incoherent or collapses into a semantic account.

But there is a way to maintain the syntactic account without relying on semantic properties. This may be seen by reflecting on the (non-semantic) properties of program-controlled computers (for a more detailed account, see Chapters 8–11). Some special mechanisms, namely program-controlled computers, have the ability to respond to (non-semantically individuated) strings of discrete states stored in their memory by executing sequences of primitive operations, which in turn generate new strings of states that get stored in memory. Different bits and pieces of these strings of states have different effects on the machine. Because of this, the strings can be analyzed into sub-strings. We can give an accurate description of how states can be compounded into sub-strings, and sub-strings can be compounded into strings, without presupposing that the strings of discrete states have any semantic content. This structure of the system of strings manipulated by the computer may be called its *syntax*. Some strings, called instructions, have the function of determining, at any given time, which operations are to be performed by the computer on the input

strings. Because of how computers are designed, the global effect of an instruction on the machine can be reduced to the effects of its sub-strings on the machine. Then, the effect of sub-strings on the computer can be assigned to them as their semantic content, and the way in which the content of the whole string depends on the content of its sub-strings can be specified by recursive clauses, with the result that the global effect of a string on the computer is assigned to it as its content. This assignment constitutes an *internal* semantics of a computer. An internal semantics assigns as contents to a system its own internal components and activities, whereas an ordinary (external) semantics assigns as contents to a system objects and properties in the system's environment.[13] Given that the strings manipulated by a computer may have a syntax (which determines how they are manipulated), and some of them have an internal semantics, they may be called a language, and indeed that is what computer scientists call them. None of this entails that computer languages have any external semantics, i.e., any semantic content in the sense used by Stich's critics, although it is compatible with their having one. Stich would probably have no difficulty in accepting that if a computer is capable of storing and executing its own instructions, then some of the computational states also have an internal semantics.

The above account of syntax is mechanistic, specified in terms of the components of a stored-program computer, their states, and their organization. From the vantage point of this mechanistic account of syntax, not only do we see the coherence of Stich's proposal, but we can also give a mechanistic account of his notion of syntax without presupposing any external semantics.

The real problem with the present version of the syntactic account of computation is that language-like syntactic structure is not necessary for computation as it is understood in computer science and computability theory. Although computing systems surely *can* manipulate linguistic structures, they don't have to. There are digital computing systems that manipulate simple sequences of letters, without losing their identity as computers. (Computability theorists call any set of words from a finite alphabet a language, but that broad notion of language should not be confused with the narrower notion—inspired by grammars in logic and linguistics—that Stich employs in his syntactic account of computation.) And then there is non-digital computation. This will motivate the full-blown mechanistic account of computation, a fragment of which was sketched above to ground the notion of language-like syntax in non-semantic terms. Within such a mechanistic framework, we could even define a broader notion of syntax that applies to any language in the sense of computability theory. If we do that, we end up with a mechanistically grounded syntactic account of (digital) computation.

The mechanistic account, to be presented in subsequent chapters, preserves a correct insight that lies behind the syntactic account of computation: the nature of

[13] For more on internal vs. external semantics, see Chapter 9; Fodor 1978; Dennett 1987.

computation is not semantic but mechanistic. And since the mechanistic account provides the resources for an account of syntax, it is incorrect to say that the mechanistic account is a syntactic account. If anything, it's the other way around: a properly syntactic account is a mechanistic account of (digital) computation. As I will argue in Chapter 7, however, there is more to computation than digital computation.

5. The Limits of Semantic Accounts

Semantic accounts of computation come in many versions, depending on which manipulations of which representations they count as computations. What they all have in common is that, according to them, computation requires representation. I have argued that, contrary to semantic accounts, representation is not required for computation. And even when computation occurs in the presence of representation—even when the vehicles of computation are representations—computations can be individuated non-semantically. That being said, let's summarize the extent to which semantic accounts satisfy the desiderata for an account of concrete computation that I listed in Chapter 1.

By restricting computation to (some kinds of) manipulation of (some kinds of) representations, semantic accounts may be able to make computation objective, which is our first desideratum. Whether they do depends on how objective representation is. If representing is an objective relation, then computation comes out objective. If representing is not objective, then computation is not objective either. Whether representing is an objective relation is a disputed matter.

Our second desideratum is an account of computational explanation. Appealing to semantically individuated computations is explanatory, especially if, again, representation is an objective property. For example, we may explain animal navigation in part by pointing at animals' ability to see, which we explain by pointing at their computing three-dimensional representations of their environment. This is computational explanation by semantically-individuated computation. It is an important aspect of our explanatory practices in both computer science and cognitive science.

But computational explanation does not stop at semantically individuated computations. Computational explanation also identifies the (non-semantically individuated) concrete vehicles that are being manipulated, the operations performed on such vehicles, the sequences of operations, and the concrete components that perform the operations. (All of this is described at a suitable level of abstraction.) In other words, as subsequent chapters will make clear, computational explanation is a form of mechanistic explanation, which may or may not include reference to the semantic properties of the computations.

Our third and fourth desiderata are that the right things compute and the wrong things don't compute. Here, whether a semantic account respects the boundaries of the computational concepts we use in our scientific practices depends on which manipulations of which representations count as computations under a semantic

account. As I mentioned in Chapter 1, some semantic accounts are too restrictive, because they rule out of the class of computing systems any system that fails to represent its own computational rules, including paradigmatic examples of computing systems such as non-universal Turing machines and standard finite state automata (Fodor 1975; Pylyshyn 1984). Other semantic accounts are too inclusive, because they attribute representational properties to all systems and they count any representational manipulation as computation (cf. Shagrir 2006b). Such overly inclusive accounts entail pancomputationalism, which will be discussed in more detail in the next chapter. It may be possible to formulate a semantic account that draws the boundary between computing and non-computing system closer to the right place. But no semantic account will ever draw the line in exactly the right place, because—as I argued throughout this chapter—some computations do not manipulate representations at all, and semantic accounts have no resources for counting them as computations.

Our fifth desideratum is an account of miscomputation. As far as I know, proponents of semantic accounts have paid a lot of attention to misrepresentation but have not addressed miscomputation. The two are logically independent notions. A system that misrepresents may or may not miscompute—it may be performing its computations correctly while using representations that are incorrect. Computer scientists say, "Garbage in, garbage out." In the other direction, a system that miscomputes may or may not misrepresent—it may not even represent anything at all; if it does represent, a miscomputation may accidentally yield a correct representation. Thus, misrepresentation does not immediately help understand miscomputation.

Finally, our sixth desideratum is an account of computational taxonomy. Here, representational notions do help somewhat. Specifically, we can distinguish between systems that manipulate representations without representing the rules they follow and systems that manipulate representations while also representing the rules they follow and using the representations of the rules to drive the computation. The latter are program-controlled computing systems (more on this in Chapter 11). But there are many more important distinctions to draw between systems with different degrees of computing power, and semantic accounts lack the resources to draw such distinctions. One of the most ambitious undertakings of this book is to show how much illuminating work the mechanistic account can do in formulating a clear taxonomy of computing systems, and illustrating the kind of theoretical payoff that can be gained from this enterprise. Once I introduce the mechanistic account in Chapter 7, I will take up this theoretical/taxonomic task in Chapters 8–13.

In conclusion, existing arguments for semantic accounts of computation fail to show that computation requires representation. Computational states need not be individuated semantically. Instead, computational states can be individuated by their structural and functional properties alone. This is good news, as a semantic individuation of computation presupposes its non-semantic individuation.

The point is not that content has no role to play in formulating and evaluating computational theories. It has many important roles to play, at least under the most common methodologies and assumptions. The point, rather, is that computing systems and their states have non-semantic identity conditions, and those identity conditions must be considered in a full explanation of their capacities. Once computational states are individuated non-semantically, semantic interpretations may (or may not) be assigned to them.

In both computer science and cognitive science, the most perspicuous way of individuating tasks is often semantic. We speak of computers doing arithmetic and of visual systems inferring properties of the world from retinal images—these are semantic characterizations of their tasks. But to those semantic characterizations, there correspond an indefinite number of possible non-semantic characterizations, which individuate different computational architectures, running different programs, written in different programming languages, executing different algorithms. Before a semantic characterization of a task can be mapped onto a particular concrete system, the semantic characterization needs to be paired with a non-semantic description of the system that performs the task.

A first corollary is that being a computational state does not entail having semantic properties. This applies to artifacts and natural systems alike. A computer can be truly described computationally without ascribing content to it, and so can a cognitive system. This corollary is important in light of the tendency among many theorists to construe the computational states postulated by computational theories of cognition as representational. This is a mistake, which begs the question of whether the computational states postulated by a theory of mind have content.[14] Perhaps they do, but perhaps—as Stephen Stich pointed out some time ago (Stich 1983)—they don't. Whether cognitive states have content should not be determined by the metaphysics of computation; it should be an independent substantive question. A good account of computation should not entail—as semantic accounts of computation do—that one cannot be a computationalist about cognitive states while also being an eliminativist about their content.

If mental states have content, there is a separate question of whether the contents of states posited by computational theories match the contents ascribed by folk psychology. Perhaps some or all the internal computational states have contents that match the folk psychological contents, as many computationalists believe (e.g., Fodor 1987; Pylyshyn 1984). Or perhaps they don't, as other computationalists maintain (e.g., Dennett 1987, esp. chap. 5). These are substantive questions that

[14] This is true only under the assumption, almost universally shared among supporters of semantic accounts of computation, that computational states are individuated by the same contents that individuate the cognitive states realized, in whole or in part, by those computational states. If one rejects that assumption, then the semantic account of computation is compatible with intentional eliminativism. But if one rejects that assumption, the semantic view of computational individuation ceases to have any significant positive motivation.

depend on the relationship between computational explanations of mental states and capacities and theories of mental content, and are at least in part empirical; they should not be settled by philosophizing on the metaphysics of computation. In light of these considerations, the mechanistic account of computation has the appealing feature that it leaves the questions of whether mental states have content and what content they have independent of the question of whether mental states are computational.

A second corollary relies on the premise that the possession of semantic properties does not entail the possession of computational properties. Since I'm not aware of any claim to the contrary, I will not argue for this premise. The two corollaries together entail that being computational is logically independent of having content, in the sense that it is possible to be a computational state without having content and vice versa. The computational theory of cognition and the representational theory of cognition address independent (orthogonal) problems. The computational theory of cognition may be formulated and discussed without presupposing that cognitive systems have content, so as to avoid getting entangled with the difficult issue of mental content. And the representational theory of cognition may be formulated without presupposing that cognitive states are computational.

These conclusions have no consequences on whether cognitive systems or computers have content, whether cognitive and computational content are the same, and whether cognitive content is reducible to computational content. Those questions must be answered by a theory of content, which is not the topic of this book.

The topic of this book is concrete computation, which, as it turns out, may or may not involve representation. In both this chapter and the previous one, we've encountered accounts that entail pancomputationalism—everything computes. Addressing pancomputationalism in detail is the task of the next chapter. Addressing pancomputationalism will help sharpen the distinction between computational modeling and computational explanation, which will bring us closer to an adequate account of concrete computation.

4

Pancomputationalism

1. Varieties of Pancomputationalism

As we saw in Chapters 2 and 3, mapping accounts and some semantic accounts of computation entail pancomputationalism—every physical system performs computations. I have encountered two gut reactions to pancomputationalism: some philosophers find it obviously false, too silly to be worth refuting; others find it obviously true, too trivial to require a defense. Neither camp sees the need for this chapter. But neither camp seems aware of the other camp. The existence of both camps, together with continuing appeals to pancomputationalism in the literature, compel me to analyze the matter more closely. In this chapter, I distinguish between different varieties of pancomputationalism and argue that both gut reactions get something right and something wrong. I find that although some varieties of pancomputationalism are more plausible than others, only the most trivial and uninteresting varieties are true. This speaks against any account of computation that entails pancomputationalism. In arguing against pancomputationalism I also sharpen the distinction between computational modeling and computational explanation, and that helps prepare the ground for the mechanistic account of computation.

Which physical systems perform computations? According to pancomputationalism, they all do. As Putnam put it, "everything is a Probabilistic Automaton under some Description" (Putnam 1967b: 31; 'probabilistic automaton' is Putnam's term for probabilistic Turing Machine).[1] Even rocks, hurricanes, and planetary systems—contrary to appearances—are computing systems. Pancomputationalism is quite popular among some philosophers and physicists.

Formulations of pancomputationalism vary with respect to how many computations—all, some, or just one—they attribute to each system.

The strongest versions of pancomputationalism maintain that every physical system performs every computation—or at least, every sufficiently complex system

[1] Cf. also: "A [physical symbol] system always contains the potential for being any other system if so instructed" (Newell 1980, 161); "For any object there is some description of that object such that under that description the object is a digital computer" (Searle 1992, 208); "everything can be conceived as a computer" (Shagrir 2006b, 393). Similar views are expressed by Block and Fodor (1972, 250); Churchland and Sejnowski (1992); Chalmers (1996, 331); Scheutz (1999, 191); and Smith (2002, 53), among others.

implements a large number of non-equivalent computations (Putnam 1988; Searle 1992). This may be called *unlimited* pancomputationalism.

Weaker versions of pancomputationalism maintain that every physical system performs one (as opposed to every) computation. Or perhaps everything performs a few computations, some of which encode the others in some relatively unproblematic way (Scheutz 2001). This may be called *limited* pancomputationalism.

Varieties of pancomputationalism also vary with respect to why everything performs computations—the *source* of pancomputationalism.

One alleged source of pancomputationalism is that which computation a system performs is a matter of relatively free interpretation. If whether a system performs a given computation depends solely or primarily on which perspective we take towards it, as opposed to objective fact, then it seems that everything computes because everything may be seen as computing (Searle 1992). This may be called *interpretivist* pancomputationalism.

Another alleged source of pancomputationalism is that everything has causal structure. According to the causal account, computation is the causal structure of physical processes at some level of abstraction (Chapter 2). Assuming that everything has causal structure, it follows that everything performs the computation constituted by its causal structure at a given level of abstraction (if there is a privileged level of abstraction), or perhaps that every physical system performs the computations constituted by its causal structures at every level of abstraction. This may be called *causal* pancomputationalism.

Not everyone will agree that everything has causal structure. Some processes may be non-causal, or causation may be just a *façon de parler* that does not capture anything fundamental about the world (e.g., Norton 2003). But those who have qualms about causation can recover a view similar to causal pancomputationalism by reformulating the causal account of computation and consequent version of pancomputationalism in terms they like—for example, in terms of laws, dispositions, or counterfactuals (cf. counterfactual and dispositional accounts of computation, Chapter 2).

A third alleged source of pancomputationalism is that every physical state carries information, in combination with an information-based semantics plus a liberal semantic account of computation (Chapter 3). According to semantic accounts of computation, computation is the manipulation of representations. According to information-based semantics, a representation is something that carries information. If every physical state carries information, then every physical system performs the computations constituted by the manipulation of its information-carrying states (cf. Shagrir 2006b). Both information-based semantics and the assumption that every physical state carries information (in the relevant sense) remain controversial.

Yet another alleged source of pancomputationalism is that computation is the nature of the physical universe. According to some physicists, the physical world is

computational at its most fundamental level. I will discuss this view, which is a special version of limited pancomputationalism, in Section 4.

2. Unlimited Pancomputationalism

Arguments for unlimited pancomputationalism differ in their details but they fall into two basic types based on especially liberal versions of, respectively, the mapping account and the semantic account.

The first argument relies on the simple mapping account. It begins by constructing a mapping between a more or less arbitrary computation—usually a digital one—and a more or less arbitrary microphysical process. The argument then assumes the simple mapping account of computation (or something close), according to which a mapping between a computational description and a microphysical description of a system, such that the microphysical process mirrors the computation, is sufficient for a computation to be physically implemented (Chapter 2, Section 1). The argument concludes that the given microphysical process implements the given computation. To the degree that the computation and the microphysical process were chosen arbitrarily, the conclusion is then generalized: any computation (of the relevant type) is implemented by any microphysical process (of the relevant type) (cf. Searle 1992; Putnam 1988).

An especially easy way of deriving this kind of mapping between an arbitrary computation and an arbitrary physical process is to point out that a (digital) computation can be described as a countable sequence of state transitions, whereas a microphysical process can usually be described as an uncountable sequence of state transitions. Since an uncountable sequence is larger than a countable sequence, there will always be a mapping from the countable sequence to a subset of the uncountable sequence such that the uncountable sequence mirrors the countable one.

The second argument appeals to unconstrained semantic interpretation. It claims that any more or less arbitrary microphysical state is subject to more or less arbitrary semantic interpretations, and is therefore a representation. Because of this, any sequence of microphysical state transitions is a sequence of representations. The argument then assumes an especially liberal version of the semantic account of computation (Chapter 3), according to which manipulating representations is sufficient for implementing a computation. The argument concludes that any microphysical process implements any more or less arbitrary computation. To the degree that the interpretation and the microphysical process were chosen arbitrarily, the conclusion is then generalized: any computation (of the relevant type) is implemented by any microphysical process (of the relevant type).

The first thing to notice about unlimited pancomputationalism is that, if it holds, the claim that a physical process implements a computation is trivialized. If every physical process implements every computation or if, more weakly, most physical processes implement lots of non-equivalent computations, the claim that a particular

physical process implements a particular computation becomes nearly vacuous. But when computer scientists and engineers say that your computer is running a particular software package, they don't seem to mean something that vacuous. Therefore, either the foundations of computer science are seriously in trouble or unlimited pancomputationalism is seriously wrong.

The second thing to notice is that arguments for pancomputationalism become more complicated if we want to consider inputs separately from state transitions and even more complicated if we try to switch from less constrained versions of the mapping and semantic accounts to more constrained versions. In fact, the more sophisticated mapping accounts discussed in Chapter 2, which restrict acceptable mappings to those involving counterfactual supporting state transitions, or causal state transitions, or state transitions grounded in dispositions, are largely motivated by avoiding unlimited pancomputationalism. I will not discuss all these complications because my goal is not to fix either the mapping account or the semantic account but to replace them with the mechanistic account. The interim conclusion is that unlimited pancomputationalism is uncontroversially very bad and that it should be possible to avoid it by fixing the mapping or semantic accounts of computation.[2]

I will now turn to criticizing *limited* pancomputationalism, which is actually endorsed by a surprisingly wide range of philosophers and scientists.

3. Limited Pancomputationalism

Limited pancomputationalism is a much more modest and plausible claim than its unlimited cousin. It holds that every physical system performs a computation, or perhaps a limited number of equivalent computations. Which computations are performed by which system depends on objective properties of the system such as its causal or dispositional structure at an appropriate level of abstraction, or what the system's states objectively represents. In fact, several authors who have mounted detailed responses to unlimited pancomputationalism and developed accounts of computation to avoid it explicitly endorse limited pancomputationalism (Chalmers 1996b, 331; Scheutz 1999, 191).

Given this reliance on objective properties of the system for attributing computations to it, limited pancomputationalism does not trivialize the claim that a given system performs a given computation (at a given level of abstraction). Which specific computation is performed by which specific system (at a given level of abstraction) depends on the specific objective properties (causal structure, dispositional structure, representational properties, etc., depending on which account of computation is employed) at the relevant level of abstraction.

[2] For discussion, see Brown 2012; Chalmers 1994, 1996, 2011, 2012; Copeland 1996; Chrisley 1995; Godfrey-Smith 2009, Scheutz 1999, 2001, 2012; Sprevak 2012.

But limited pancomputationalism still erases the distinction between systems that compute and systems that don't. Notice that, usually, limited pancomputationalists adopt a digital-only conception of computation, so they end up attributing digital computations to every physical system. According to them, rocks, hurricanes, and planetary systems perform (digital) computations in the same sense in which standard digital computers do. This result clashes not only with our untutored intuitions but also with our scientific practices. We have entire scientific disciplines devoted to studying specific classes of physical systems (digital computers, analog computers, quantum computers), figuring out which computations they can perform, and figuring out how efficiently and reliably they can perform them. It takes a lot of difficult technical work to design and build systems that perform digital computations reliably. To say that rocks do the same thing without any help from computer engineers makes a mockery of our computer science and technology. Or consider the vast efforts devoted to figuring out which computations are involved in performing cognitive tasks. The computational theory of cognition was introduced to shed new and explanatory light on cognition (Piccinini 2004b; Piccinini and Bahar 2013). But if every physical process is a computation, the computational theory of cognition loses much of its explanatory force.

In the face of these objections, limited pancomputationalists are likely to maintain that the explanatory force of computational explanations does not come from the claim that a system is computational *simpliciter*. Rather, explanatory force comes from the specific computations that a system performs (e.g., Chalmers 2011). According to this response, yes, a rock and a digital computer do perform computations in the same sense. But they perform radically different computations, and it is the difference between their computations that explains the difference between their behaviors. In the rest of this chapter, I will argue that this reply fails to consider the specific way in which computation explains the behavior of certain systems and not others.

Another objection to limited pancomputationalism begins with the observation that any moderately complex system satisfies indefinitely many objective computational descriptions. This may be seen by considering computational modeling. A computational model of a system may be pitched at different levels of granularity. For example, consider cellular automata models of the dynamics of a galaxy or a brain. The dynamics of a galaxy or a brain may be described using an indefinite number of cellular automata—using different state transition rules, different time steps, or cells that represent spatial regions of different sizes. Furthermore, an indefinite number of formalisms different from cellular automata can be used to compute the same functions computed by cellular automata. It appears that limited pancomputationalists are committed to the galaxy or the brain performing all these computations at once. This upshot does not trivialize the claim that a specific system performs a specific computation to the same degree as unlimited pancomputationalism does, because all of these attributions of computations to physical systems are

tied to objective properties of the system. But the claim that a given system performs a given computation is still trivialized to a large extent, because it still depends on matters that have nothing to do with the objective properties of the system, such as how fine-grained our model is or which programming language we use in building our model.

That does not appear to be the sense in which computers (or brains) perform computations. In the sciences of computation and cognition, there is an important distinction between the computations performed by a computational model of a system and the computations performed by the system being modeled. The computations performed by the system are specific to the system and do not depend on properties of the model such as the programming language being used in building the model or the fineness of grain at which the model is built. In other words, within our computational sciences there is an important distinction between mere computational *models* and computational *explanations*—or so I will soon argue. Limited pancomputationalism ignores this important distinction and the scientific practices that rely on it. Once again, limited pancomputationalism fails to do justice to our scientific practices involving computation. Either there is something deeply wrong with those scientific practices or, at the very least, limited pancomputationalism fails to capture the notion of concrete computation that is employed within those practices.

In the face of this objection, limited pancomputationalists may attempt to single out, among the many computational descriptions satisfied by each system, the one that is ontologically privileged—the one that captures the computation performed by the system. Such a find would restore full objectivity to the claim that a given system performs a given computation: at the privileged level of abstraction, there would be one and only one computation performed by each physical system. The main way to identify this privileged computation is to postulate a fundamental physical level whose most accurate computational description identifies the (most fundamental) computation performed by the system. This response is built into the view that the physical world is fundamentally computational.

4. The Universe as a Computing System

To claim that the physical universe is fundamentally computational is becoming increasingly popular: the universe itself is a computing system, and everything in it is a computing system too (or part thereof). Unlike the previous versions of pancomputationalism, which originate in philosophy, this *ontic* pancomputationalism originates in physics. It includes both an empirical claim and a metaphysical one. Although the two claims are logically independent, supporters of ontic pancomputationalism tend to make them both.

The empirical claim is that all fundamental physical magnitudes and their state transitions are such as to be exactly described by an appropriate computational

formalism—without resorting to the approximations that are a staple of standard computational modeling. This claim takes different forms depending on which computational formalism is taken to describe the universe exactly. The two main options are cellular automata, which are a classical computational formalism, and quantum computing, which is non-classical.

The earliest and best known version of ontic pancomputationalism is due to Konrad Zuse (1970, 1982) and Edward Fredkin, whose unpublished ideas on the subject influenced a number of American physicists (e.g., Feynman 1982; Toffoli 1982; Wolfram 2002; see also Wheeler 1982; Fredkin 1990). According to some of these physicists, the universe is a giant cellular automaton. A cellular automaton is a lattice of cells; each cell can take one out of finitely many states and updates its state in discrete steps depending on the state of its neighboring cells. For the universe to be a cellular automaton, all fundamental physical magnitudes must be discrete, i.e., they must take at most finitely many values. In addition, time and space must be fundamentally discrete or must emerge from the discrete processing of the cellular automaton. At a fundamental level, continuity is not a real feature of the world—there are no truly real-valued physical quantities. This flies in the face of most mainstream physics, but it is not an obviously false hypothesis. The hypothesis is that at a sufficiently small scale, which is currently beyond our observational and experimental reach, (apparent) continuity gives way to discreteness. Thus, all values of all fundamental variables, and all state transitions, can be fully and exactly captured by the states and state transitions of a cellular automaton.

Although cellular automata have been shown to describe many aspects of fundamental physics, it is difficult to see how to simulate the quantum mechanical features of the universe using a classical formalism such as cellular automata (Feynman 1982). This concern motivated the development of quantum computing formalisms (Deutsch 1985; Nielsen and Chuang 2000). Instead of relying on digits (most commonly, binary digits or bits) quantum computation relies on qudits (most commonly, binary qudits or qubits). The main difference between a digit and a qudit is that whereas a digit can take only one out of finitely many states, such as 0 and 1 (in the case of a bit), a qudit can also take an uncountable number of states that are superpositions of the basis states in varying degrees, such as superpositions of 0 and 1 (in the case of a qubit). Furthermore, unlike a collection of digits, a collection of qudits can exhibit quantum entanglement. According to the quantum version of ontic pancomputationalism, the universe is not a classical computer but a quantum computer, that is, not a computer that manipulates digits but a computer that manipulates qubits (Lloyd 2006)—or, more generally, qudits.

The quantum version of ontic pancomputationalism is less radical than the classical version. The classical version eliminates continuity from the universe, primarily on the grounds that eliminating continuity allows classical digital computers to describe the universe exactly rather than approximately. Thus, the classical version appears to be motivated not by empirical evidence but by epistemological

concerns. Although there is no direct evidence for classical ontic pancomputationalism, in principle it is a testable hypothesis (Fredkin 1990). By contrast, quantum ontic pancomputationalism may be seen as simply replacing some traditional quantum mechanical terminology with the terminology of quantum computation and quantum information theory (qudits) (e.g., Fuchs 2004; Bub 2005). Such a terminological change leaves the empirical content of quantum mechanics intact—insofar as the quantum version of ontic pancomputationalism makes empirical claims, it makes the same empirical claims as traditional quantum mechanics.

But ontic pancomputationalists do not limit themselves to making empirical claims. They often make an additional metaphysical claim. They claim that computation (or information, in the physical sense closely related to thermodynamic entropy) is what makes up the physical universe. This point is sometimes made by saying that, at the most fundamental physical level, there are brute differences between states—nothing more needs to or can be said about the nature of the states. This view reverses the traditional conception of the relation between computation and the physical world.

According to the traditional conception, which is presupposed by all accounts of computation I discussed up to here, physical computation requires a physical medium that implements it. Computation is an aspect of the organization and behavior of a physical system—there is no software without hardware. Thus, according to the traditional conception, if the universe is a cellular automaton, the ultimate constituents of the universe are the physical cells of the cellular automaton. It is legitimate to ask what kind of physical entity such cells are and how they interact with one another so as to satisfy their cellular automata rules.

By contrast, according to the metaphysical claim of ontic pancomputationalism, a physical system is just a system of computational states. Computation is ontologically prior to physical processes, as it were. "'Hardware' [is] made of 'software'" (Kantor 1982, 526, 534). According to this heterodox conception, if the universe is a cellular automaton, the cells of the automaton are not concrete, physical structures that causally interact with one another. Rather, they are software—purely "computational" entities.

Such a metaphysical claim requires an account of what computation, or software, or physical information, is. If computations are not configurations of physical entities, the most obvious alternative is that computations are abstract, mathematical entities, as numbers and sets are according to platonism (cf. Chapter 1). According to platonism, mathematical entities have no spatial location, no temporal duration, and no causal properties. They are not concrete, causal, spatiotemporal entities. The view that physical entities somehow reduce to abstract mathematical ones is called Pythagoreanism. The metaphysical claim of ontic pancomputationalism—the claim that physical systems reduce to computational ones—is a computational incarnation of Pythagoreanism. All is computation in the same sense in which more traditional

versions of Pythagoreanism maintain that all is number or that all is sets (Quine 1976).

There is some textual evidence that some ontic pancomputationalists do subscribe to this form of Pythagoreanism. As Wheeler (1982, 570) puts it, "the building element [of the universe] is the elementary 'yes, no' quantum phenomenon. It is an abstract entity. It is not localized in space and time." In addition, ontic pancomputationalists sometimes attempt to explain how space and time themselves emerge from the properties of the fundamental computing system they postulate. In any case, I'm not aware of any other way of cashing out the metaphysical claim of ontic pancomputationalism besides Pythagoreanism. As we've seen, ontic pancomputationalists explicitly reject the traditional view that physical computations are aspects of physical systems (for them it's the other way around: physical systems are aspects of computations). If they wish to also reject the claim that physical computations are abstract objects, they should explain what their ontology amounts to.

While it is beyond the scope of this book to do justice to ontic pancomputationalism, I reject it on both the empirical and the ontological fronts.

On the empirical front, insofar as ontic pancomputationalism departs from mainstream physics, there is hardly any positive evidence to support it. Proponents appear to be motivated by the desire for exact computational models of the world rather than empirical evidence that the models are correct. Even someone who shares this desire may well wonder why we should expect nature to fulfill it.

On the metaphysical front, let's grant for a moment, for the sake of the argument, that there are abstract entities such as numbers, sets, or software. These are the entities that Pythagoreanism puts at the fundamental physical level—they constitute everything physical. Specifically, according to ontic pancomputationalism, everything physical is made out of software. But physical things have causal powers that they exert through spacetime—they *do* things (push and pull, attract and repel, etc.). By contrast, abstract entities are supposed to have no spatiotemporal location and no causal powers. To put it mildly, it is not clear how abstract entities that have no causal powers and no spatiotemporal location can give rise to concrete things that exert causal powers through spacetime. In addition, physical things have concrete qualities that differentiate them from one another—categorical properties such as shape, mass, charge, etc. But again, abstract entities do not appear to have any concrete qualities. Again, it is not clear how abstract entities that have no concrete qualities can give rise to physical things that have concrete qualities. Once it is thought through, ontic pancomputationalism—like the Pythagoreanism of which it is a species—turns out to be at best a big metaphysical mystery, at worst a category mistake (cf. Martin 1997). It is too mysterious for my stomach.

Finally, the above objections to ontic pancomputationalism were based on the assumption that there are abstract entities. But what are abstract objects? We know what they are not—they are not in spacetime, they have no causal properties. Well, what *are* they? I have yet to find a convincing positive account of abstract objects.

And in the absence of a positive account, I find abstract objects unintelligible. This, of course, does not even begin to do justice to the subtle ontological issues surrounding abstract objects and the ontology of mathematics more generally—but that is a topic for another occasion. Meanwhile, for anyone like me who rejects abstract objects, ontic pancomputationalism—or at least its metaphysical claim—is a nonstarter.

So far, the moral of this chapter is that unlimited pancomputationalism is the product of insufficiently constrained accounts of physical computation, whereas limited pancomputationalism flies in the face of our scientific practices, and ontic pancomputationalism (a specific version of limited pancomputationalism) is implausible in its own right. In the rest of this chapter, I articulate the distinction between computational modeling and computational explanation. This distinction grounds a conclusive objection to limited pancomputationalism and shows us the way towards a more adequate account of concrete computation. From now on, by 'pancomputationalism' I will mean, primarily, limited pancomputationalism.

5. Computational Modeling vs. Computational Explanation

As I've pointed out, pancomputationalism clashes with both our scientific practices and common intuitions pertaining to computation. Normally we distinguish between systems that perform computations and systems that don't: "the solar system is not a computational system, but you and I, for all we now know, may be" (Fodor 1975, 74, n. 15; see also Fodor 1968a, 632; Dreyfus 1979, 68, 101–2; Searle 1980, 37–8; Searle 1992, 208). Besides planetary systems, stomachs and the weather are some of the most often cited paradigmatic examples of systems that do *not* perform computations. The view that some things are not computing systems flatly contradicts pancomputationalism. To resolve this contradiction, something needs to be done.

The best way forward is to distinguish different kinds of computational descriptions. Some computational descriptions *explain* the behavior of things by appeal to their computations, others do not. The former are relevant to computer science and computational neuroscience, the latter are not. Some authors have suggested something along these lines. For example, Block and Fodor write that "there are many ways in which it could turn out that organisms are automata [i.e., probabilistic Turing machines] in some sense more interesting than the sense in which everything is an automaton under some description" (Block and Fodor 1972, 250).

But until recently no one bothered to articulate in a satisfactory way the difference between explanatory and non-explanatory computational descriptions and their implications for pancomputationalism and the philosophy of computation. This chapter begins doing so, and the following chapters will complete the task. I will argue that pancomputationalism stems from lack of clarity on the distinction between computational modeling and computational explanation. Once that distinction is clarified, the only versions of pancomputationalism that survive are trivial and

uninteresting. And when pancomputationalism goes, the accounts of computation that entail it—mapping accounts and some versions of the semantic account—have to go with it. I've already argued that those accounts are inadequate in Chapters 2 and 3 because, among other reasons, they do not provide an adequate account of computational explanation. The rest of this Chapter completes that objection.

To a first approximation, the distinction we need is that between using a computational description to *model* the behavior of a system—such as when meteorologists predict the weather using computers—and using it to *explain* the behavior of a system—such as when computer scientists explain what computers do by appealing to the programs they execute. The two kinds of computational descriptions have different ontological implications about whether the behavior being described is a computation.

In *computational modeling* (as I'm using the term), the outputs of a computing system *C* are used to describe some behavior of another system *S* under some conditions. The explanation for *S*'s behavior has to do with *S*'s properties, not with the computation performed by the model. *C* performs computations in order to generate subsequent descriptions of *S*. The situation is analogous to other cases of modeling: just as a system may be modeled by a diagram or equation without being a diagram or equation in any interesting sense, a system may be modeled by a computing system without being a computing system in any interesting sense.

In *computational explanation*, by contrast, some behavior of a system *S* is explained by a particular kind of process internal to *S*—a computation—and by the properties of that computation. For instance, suppose we have a calculator in working order (i.e., it has power and is functioning properly). Shortly after we press certain buttons on the calculator in a certain sequence—say, the buttons marked '5,' '√,' and '='—a certain string of symbols, i.e. '2.236...,' appears on the calculator's display. We explain the calculator's output by pointing to the inputs we inserted into the calculator, the fact that the string '2.236...' represents the number 2.236..., the fact that 2.236... is the square root of 5, and—most crucially for present purposes—the specific activity performed by the calculator; namely, the computation of square roots. Whether we use another computing system to describe our calculator's behavior is independent of whether the explanation for that behavior appeals to a computation performed by the calculator. If we do use a computing system *C* distinct from our calculator to describe the calculator's behavior, then there will be two different computations: the calculator's and *C*'s. Nonetheless, the behavior of the calculator is explained by the fact that, *ceteris paribus*, it performs a square root computation.

In the next three sections, I provide a more explicit and precise taxonomy of legitimate senses in which something may be described computationally. First I discuss ordinary computational models based on differential equations, then computational models based on discrete formalisms, and finally computational explanation. In each case, I formulate a precise version of pancomputationalism,

evaluate it, and draw the relevant consequences for computer science and cognitive science.

6. Computational Modeling: Differential Equations

In one type of computational description, the states of a system *S* are represented by the outputs of a computing system *C*, and *C* computes representations of *S*'s state at different times. In order for *C* to compute representations of *S*, *C* must be given two sorts of inputs: (i) an input specifying *S*'s state at some initial time t_0, and (ii) an input specifying *S*'s dynamical evolution (i.e., how *S*'s state evolves over time).

Trivially, *S*'s dynamical evolution may be specified by representations of *S*'s states at subsequent times, which may be obtained by measuring the relevant variables of *S* at subsequent times. Such a trivial specification would then constitute a look-up table of *S*'s dynamical evolution. In the presence of such a table, *C*'s job reduces to retrieving the appropriate item from the look-up table. Less trivially, what is normally done is that *S*'s dynamical evolution is given by a mathematical description *A*—typically, a system of differential equations—which specifies how *S*'s variables vary as a function of *S*'s state.

If *A* is solvable analytically (and if the solution is known), then *C* may be given a representation of *A*'s solutions as inputs, and *C* may use that input (together with an input specifying *S*'s initial state) to compute a representation of *S*'s state at any given time. As is well known, however, most systems of differential equations are not solvable analytically, and this is where the present type of computational modeling proves most helpful. Mathematicians have devised numerical methods for approximating the values of a system *S*'s variables directly from *S*'s dynamical description *A*, without needing to rely on *A*'s analytic solutions. In such cases, *C* may be given a representation of *A* as input, and *C* may apply numerical methods to those inputs (together with an input specifying *S*'s initial state) to compute a representation of *S*'s state at any given time. This is the most common type of computational modeling, which has become ubiquitous in many sciences (Rohrlich 1990; Humphreys 2004; Winsberg 2010; Weisberg 2013).

The versatility of computational models and their popularity in many quarters of science may be part of the original motivation behind pancomputationalism. Given how many systems are routinely given computational descriptions by scientists in the most disparate disciplines, ranging from physics to biology to the social sciences, it is tempting to conclude that everything can be described as a computing system in the present sense. In fact, sometimes pancomputationalism is formulated as the claim that everything can be "simulated" by a computing system.[3] This simulation-based

[3] For early claims to this effect, see von Neumann 1951 and Putnam 1964. Some recent examples: "a standard digital computer . . . can display any pattern of responses to the environment whatsoever" (Churchland and Churchland 1990); "the laws of physics, at least as currently understood, are computable,

formulation of pancomputationalism implies that, if something can be simulated computationally, it can also be described as a computation. This implication can be questioned, of course, and my ultimate goal is to reject it. But even if we accept the implication for the sake of the argument, careful examination of computational modeling undermines pancomputationalism.

Most scientific descriptions are not exact but approximate. At the very least, measurements can be performed only within a margin of error, and the values of a system's variables can be specified only with finite precision. The kind of computational descriptions under discussion are not only approximate in these standard manners, but also in more significant ways. First, the mathematical description *A* that specifies the dynamical evolution of a system *S* only represents what is known about the dynamical evolution of *S*. Some factors that influence *S*'s dynamical evolution may be unknown, and since *A* may not capture them, the dynamical evolution specified by *A* may differ from *S*'s actual dynamical evolution. Second, to include everything that is known about *S* in *A* may make the mathematics analytically or computationally intractable. Typical dynamical descriptions within the sciences embody idealizations and simplifications relative to what is known about a system, and these idealizations and simplifications may generate a difference between what the descriptions say and what the system does. Third, the numerical methods for computing the state of a system from its dynamical equations are only approximate, introducing a further discrepancy between the outputs of the computational model and the behavior of the modeled system. Fourth, computational accuracy requires computational resources, such as memory and time. Typically, the more accuracy is required, the more computational resources need to be invested; but computational resources are always finite. Fifth, most deterministic dynamical systems are nonlinear, and most nonlinear deterministic dynamical systems have dynamics that are very sensitive to the system's initial conditions. As a consequence, many systems' dynamical evolution diverges exponentially from any representation of their dynamical evolution based on a finite specification of their initial condition. (A finite specification, of course, is all that scientists can generate in practice). Finally, many systems are non-deterministic, so that their model can predict one of their possible behaviors, but not their actual one. Because of these factors, computational models generate descriptions that only approximate the behavior of a system to some degree.

If we don't care how good our approximations are, that is, if we allow the approximations generated by our computational descriptions to be arbitrarily distant from the dynamical evolution of the system being approximated, then the thesis that everything can be described as a computing system in the present sense becomes trivially true. But one virtue of scientific descriptions is accuracy, and one goal of scientists when building computational descriptions is to generate relatively accurate

and . . . human behavior is a consequence of physical laws. If so, then it follows that a computational system can simulate human behavior" (Chalmers 1996a, 329).

representations of a system's dynamical evolution. If we do care how good our approximations are, then the thesis that everything can be described as a computing system becomes too fuzzy to be significant. For whether something can be described as a computing system becomes a matter of degree, which depends on whether it can be computationally approximated to the degree of accuracy that is desired in any given case. The answer varies from case to case, and it depends at least on the dynamical properties of the system, how much is known about them, what idealizations and simplifications are adopted in the model, what numerical methods are used in the computation, and what computational resources are available. Building computational models that are relatively accurate and knowing in what ways and to what degree they are accurate takes a lot of hard work.[4]

The statement that something can be described as a computing system in the present sense applies equally well to paradigmatic computing systems (e.g., digital computers can be approximated by other computers) and to paradigmatic noncomputing systems (e.g., the weather can be approximated by meteorological computer programs). What explains the system's behavior has to do with the properties of the system, which may or may not be computational, not with the computation performed by the model.

In the present sense, '*S* is a computing system' means that *S* can be described as a computing system to some degree of approximation for some modeling purpose. This can be done in an indefinite number of ways using a variety of assumptions, algorithms, notations, programming languages, and architectures. None of the resulting computational descriptions constitute computations performed by the modeled system. The computational descriptions play a modeling role fully analogous to the role played by differential equations, diagrams, and other modeling tools. Just as the same equation can describe systems that are physically very different, in the present sense the same computational description can describe systems that are physically very different. Just as the same system can be described by many different equations, some of which may approximate its behavior better than others, the same system can be described by many different computational descriptions, some of which may approximate its behavior better than others. Just as being described by a system of equations does not entail being a system of equations in any interesting sense, being described as a computing system in the present sense does not entail being a computing system in any deep sense. So, computational descriptions in the present sense say nothing about whether something literally computes. They are not the basis for computational explanation in computer science or cognitive science.

[4] Some authors have argued that some physical systems have dynamical evolutions whose state transitions are not computable by Turing machines, and therefore by ordinary digital computers (e.g., Penrose 1994). If one is strict about approximation and there are systems whose dynamical evolution involves state transitions that are not computable by Turing machines, then the thesis that everything is an (ordinary) computing system in the present sense becomes strictly false. I discuss whether all physical processes are computable by Turing machines in more detail in Chapters 15 and 16.

7. Computational Modeling: Cellular Automata

In a second type of computational description, the states of a system *S* are represented directly by the discrete states of an ordinary digital computing system *C* (such as a cellular automaton), and *C*'s state transitions represent *S*'s state transitions. If *S* is analyzed as a system with inputs and outputs, then *C*'s inputs and outputs represent *S*'s inputs and outputs, and given any inputs and outputs of *C* (representing any inputs and outputs of *S*), *C* goes into internal states and generates outputs that represent the states that *S* goes into and the outputs it generates.

Prima facie, not everything is describable as a computing system in this sense. For most things do not seem to have (discrete) inputs, internal states, and outputs like ordinary digital computing systems do, so it is not obvious how to compare their behavior to the behavior of a digital computing system to determine whether they are the same.

A natural suggestion might be that, for any system *S*, there is a digital computing system whose state transitions map onto *S*'s state transitions under its ordinary dynamical (microphysical) description (cf. mapping accounts of computation, Chapter 2). But this will not work. In modern science, dynamical descriptions are usually given not by means of digital computing systems but by systems of differential equations, which determine a continuous state space, which assigns an uncountable number of possible states and state space trajectories.[5] But ordinary digital computing systems, such as Turing machines (TM), can only take a finite number of states. Even if we combine the internal states of a TM with the content of the machine's tape to increase the number of possible states, the total number of states that a TM can be in is only countably infinite. Moreover, TMs can only follow a countable number of state space trajectories. The same point applies to any ordinary digital computing system of the kinds used in scientific modeling. So ordinary digital computational descriptions do not have a cardinality of states and state space trajectories that is sufficient for them to map onto ordinary mathematical descriptions of natural systems. Thus, from the point of view of strict mathematical description, the thesis that everything is a computing system in this second sense cannot be supported.[6]

This second sense in which things can be described as computing systems may be loosened by allowing the computational description *C* of a system *S* to approximate, rather than strictly map onto, the states and behavior of *S*. This kind of approximation is

[5] Any real number within a relevant interval specifies a different initial condition of a dynamical system. For any initial condition, there is a separate state space trajectory. And within any real interval, there are uncountably many real numbers. Therefore, there are uncountably many state space trajectories. This is true not only in physics but also biology, including neuroscience (for an introduction to theoretical neuroscience, see Dayan and Abbott 2001).

[6] The same argument applies, of course, to the kind of computational modeling described in the previous section, where we reached the same conclusion by a different route.

behind the use of cellular automata as a modeling tool (Rohrlich 1990; Hughes 1999). As with any model, the computational model of a system *S* only represents what is known about *S*. More importantly, discrete computational models require that *S* be discretized, namely, they require the partitioning of *S*'s states into discrete states, of *S*'s state transitions into discrete state transitions, and (in the case of cellular automata models) of *S*'s spatial regions into discrete spatial regions. This can be done in an indefinite number of ways using an indefinite variety of formalisms, some of which may be more accurate than others for some modeling purposes.[7] Finally, computational accuracy still requires computational resources, which are always finite.

Once approximation is allowed, the caveat discussed in the previous section applies. If we don't care how good our approximations are, then the thesis that everything can be described as a computing system becomes trivially true in the present sense. Otherwise, whether something can be described as a computing system in the present sense depends on whether it can be computationally approximated to the degree of accuracy that is desired in any given case.

In any case, this second type of computational description is irrelevant to computer science and cognitive science, because it applies to anything depending merely on how discrete it is at the relevant level of description, that is, on whether it has discrete inputs, outputs, internal states, and state transitions, and perhaps on one's criteria for acceptable approximations. For example, few people would count hard bodies as such as computing systems. Yet, at a high level of abstraction hard bodies can be in either of two states, whole or broken, depending on how much pressure is applied to their extremities. A simple two-input, two-state TM, or a simple cellular automaton, can approximate the transition of a hard body from one state to the other. Nevertheless, there seems to be no useful sense in which this turns every hard body into a computing system. If you hit an ordinary desktop computer sufficiently hard, you will break it. The resulting state transition of the computer, far from constituting a computation, will prevent the computer from performing computations in the future. So, this type of computational description says nothing about whether something computes. It cannot be the notion employed in computer science or cognitive science.

Someone may reply that at the appropriate level of description, even a computer that breaks is performing a computation, albeit an uninteresting one. According to

[7] As we discussed in Section 4, some proponents of this kind of modeling suggest that the universe may be fundamentally discrete in the relevant sense, so that it is possible to build exact computational models of the universe (e.g., Vichniac 1984; Toffoli 1984; Wolfram 2002). Even if true, this would make these models exact only when the most fundamental physical variables are represented at the most fundamental orders of magnitude. All other computational modeling would remain approximate. As far as I can tell, this includes all modeling done to date. For no one knows what the most fundamental physical level is. Furthermore, there is no independent evidence that the universe is fundamentally discrete in the relevant ways. As I argued in Section 4, the view that the universe is fundamentally discrete appears to be motivated by the combination of a desire for exact discrete computational models and a dubious ontology rather than by empirical evidence or independent theoretical considerations.

this line of thought, the same system can perform different computations at different levels of description, and the breaking of a computer is just one computation at one level of description among many. Some authors like to call any activity of any system a computation (e.g., Wolfram 2002). Ascribing computations in this way, though, does not make this kind of computational description relevant to the sciences of computation. There are two reasons for this.

First, computation in this sense plays little if any explanatory role. What explains the breaking of a hard body is the physical properties of the body and the amount of pressure applied to it. Different bodies of different shapes and hardness break under different pressures applied in different ways—the indefinitely many computational descriptions that are common to them are post hoc and give no information about when something will break. Unlike this example, cellular automata and other computational formalisms can have a nontrivial modeling role, but the important point still applies. The explanation for the system's behavior is given by the properties and initial conditions of the system, not by the models' computations. Since this kind of computational description would not play an explanatory role in computer science or cognitive science, it's not what computer science or cognitive science should appeal to.

The second reason is that computations ascribed in the present sense (as well as in the previous one), unlike computations properly so called, cannot go wrong. When a computer or person who is computing function *f* gives the wrong output for a given input, we say that a mistake was made (e.g., because of distraction in the case of a person, or component failure in the case of a machine), and the resulting event may be called *miscomputation*. The possibility of specifying the function to be computed independently of the performance during execution, so that one can point to mistakes in the computation, is an important reason why computation is used as an ingredient in theories of cognition. But there is no sense in which something that breaks under pressure can fail to generate the appropriate output: it is simply a law of physics that it will break—the system can't do anything different. Even if we weaken or strengthen a system so that it won't break under certain conditions, there is no useful sense in which the modified system is doing something right or wrong. To the extent that computer science and cognitive science require a notion of computation such that mistakes can be made during computations, the present type of computational description is irrelevant to them. Again, this is not what people who are interested in explaining computers and brains should be concerned with.

8. Representations, Functions, and Computational Explanation

To obtain a more robust notion of computational description, with some explanatory purchase to be employed in computer science and cognitive science, philosophers have explored two routes: representation and function. This section briefly discusses

their benefits and limitations, paving the way for an improved understanding of computational explanation.

If we assume that some things are representations and some aren't, and if we define genuine computations as manipulations of representations, perhaps we obtain the notion of computation that we need for computer science and cognitive science. According to this line of thought, which is of a piece with semantic accounts of computation, only processes defined over representations count as genuine computations, so only systems that manipulate representations count as genuine computing systems (Chapter 3). There is consensus that most systems, including most systems that are paradigmatic examples of non-computing systems—such as the weather, stomachs, and planetary systems—do not manipulate representations. So, according to typical semantic accounts of computation, most systems do not count as genuine computing systems. This strategy has the great virtue of tailoring a robust notion of computation to the common assumption that cognitive states are representations. If computers and brains manipulate representations, then they qualify for being members of the class of genuine computing systems. If they are computing systems in this sense, then their behavior is explained by the computations they perform. This is one way in which semantic accounts are superior to mapping accounts: they provide an account of computational explanation (desideratum 2).

Unfortunately, accounting for computational explanation in terms of representations is at best only part of the story. There are two reasons for this (Chapter 3). First, contrary to the semantic account, computation does not require representation. When computation occurs without representation, there must be something other than representation that accounts for computational explanation. Second, even when computation does involve representation, there is a lot more to computational explanation than an appeal to representation: there are computational vehicles, algorithms, architectures, etc. Accordingly, I set aside any strategy for avoiding pancomputationalism that relies solely on representation. The mechanistic account I will propose is compatible with representation playing a role in some forms of computational explanation, but there is a lot more to computational explanation than appealing to representations.

The second route explored by philosophers in search of a robust notion of computation is function—not mathematical function but the notion of function we use when we say that the function of the heart is to pump blood. Roughly speaking, a functional analysis of a system is an explanation of the capacities, or functions, of a system in terms of its sub-capacities, or sub-functions (Cummins 1983, 2002). In many cases, a system's sub-capacities are assigned to its components. A standard example is the circulatory system of organisms, which may be partitioned into the heart, the arteries, and the veins, each of which are assigned specific functions. The heart has the function to pump blood from the veins into the arteries, and the capacity of the circulatory system to circulate blood is explained by the functions performed by the heart, the arteries, and the veins.

When a functional analysis of a system's capacities into sub-capacities is paired with a localization of the sub-capacities in the system's components, the result is a mechanistic explanation of the system's capacities (Bechtel and Richardson 1993). In the next chapter, I will argue that functional analyses are mechanism sketches and, therefore, functional analysis is (perhaps partial) mechanistic explanation. For now, I will continue to use the traditional terminology of functional analysis because it was used to account for computational explanation by authors who thought of it as *distinct* from mechanistic explanation.

Functional analysis is an explanatory style that, plausibly, applies only to some systems among others. If so, functional analysis has nontrivial ontological implications for the system being described: it gives functional significance to the behavior of the system and its components; namely, it attributes to the system and its components the *function* of acting in certain ways under certain conditions. In fact, only artifacts and biological systems are usually said to have functions, which are then invoked in explaining their behavior. Perhaps we could try to exploit this fact to account for computational explanation in terms of functional analysis.

To look for an account of computational explanation in terms of functional analysis, we may begin by looking at the way the notion of function employed within functional analysis plays a legitimate role within computational models of biological systems and artifacts, where functional analysis is fruitfully combined with computational modeling. This can be done with either of the two kinds of computational model discussed above.

In the first kind of computational modeling, the mathematical description of a system's dynamical evolution may embody assumptions about the system's functions and sub-functions. For instance, standard equations for the action potential of neurons, such as the classical Hodgkin-Huxley equation, include terms corresponding to several electric currents. The different currents are assumed to be the effects of different components and properties of a neuron's membrane (such as various ion channels) under normal conditions, so the different terms in the equations embody these assumptions about the functional analysis of the neuron. When computer programs are built to compute representations of action potentials based on the relevant equations, they implicitly rely on the functional analysis embodied in the equations.

Mutatis mutandis, the same point applies to the use of discrete computational models such as cellular automata. The only difference is that now the functional analysis of a system is embodied directly in the topological and dynamical structure of the model. Typically, different regions of a finite automaton represent different regions of the modeled system, and the transition rules between states of the finite automaton represent the dynamical properties of the regions of the system. If the system is functionally analyzed, then different regions of a finite automaton may correspond to different components of the modeled system, and the transition rules between states of the finite automaton may represent the functions performed by

those components under normal conditions. In all these cases, functional analysis is incorporated within the computational model. This, however, does not give us the robust notion of computational explanation that we are looking for.

Computational models that embody functional analyses explain the capacities of a system in terms of its sub-capacities. But this explanation is given by the functional analysis, not by the computations performed by the model. In the case of standard computational models, the functional analysis is embodied in the dynamical equations that describe the dynamical evolution of the system or in the assumptions behind the topology and transition rules of a cellular automaton. The situation is analogous to other computational models, where the purpose of the computation is to generate successive representations of the state of a system on the basis of independent assumptions about the system's properties. What does the explaining is the set of assumptions about the system's properties—in this case, the functional analysis that is the basis for the model. And the functional analysis may or may not attribute computations to the system being modeled. For example, just because we construct a computational model of the circulatory system that embodies its functional analysis, it doesn't follow that the circulatory system performs computations or that the computations performed by the model explain the circulation of the blood. What explains the circulation of the blood is the capacities of the circulatory system that are described by its functional analysis.

The same point may be made by comparing the normativity inherent in functional analysis to the normativity inherent in computation. In the case of computational models that embody a functional analysis, the two sets of norms are independent—they may be broken independently. The system may fail to perform its functions (e.g., a heart may cease to pump) in a variety of ways under a variety of circumstances. This may or may not be represented by a computational model of the system; to the extent that the model is accurate, it should represent malfunctions and functional failures of the modeled system under the relevant circumstances. This has nothing to do with miscomputation.

The computations that generate representations of the modeled system within a computational model may also go wrong in a variety of ways under a variety of circumstances. For instance, the system may run out of memory, or a component of the hardware may break down. But if the computation goes wrong, the result is not a representation of a malfunction in the modeled system. It is simply a misrepresentation of the behavior of the modeled system, or, more likely, the failure to generate a representation of the modeled system. This shows that in this kind of modeling, the normativity of the functional analysis is independent of the normativity of the computation. The system is supposed to do what its functional analysis says; the model is supposed to compute what the equations (or other relevant assumptions) say. So, in this case, still, computation is not explaining the behavior of the system.

At this point, a tempting way to assign explanatory force to computation is to simply equate the two and assert that functional analyses are themselves

computational explanations (and vice versa). This move was implicitly made by the two original philosophical proponents of computational theories of cognition (Fodor 1968b; Putnam 1967b; more on how this came about and why it's problematic in Chapter 5). After that, the mongrel of functional and computational explanation became entrenched in the philosophy of psychology and neuroscience literature, where it can be found in many places to various degrees (e.g., Cummins 1975, 1983; Dennett 1978; Haugeland 1978, Marr 1982; Churchland and Sejnowski 1992; Eliasmith 2003). According to this view, the functions performed by a system's components according to its functional analysis are also mathematical functions computed by that system. Hence, one can explain the behavior of the system by appealing to the computations it performs. The virtue of this move is that it assigns computation (and hence computational theories of cognition) explanatory force. The vice is that it assigns explanatory force by definitional fiat.

Aside from the desire to assign explanatory force to computation, there is no independent motivation for calling all functional analyses computational, and all activities described by functional analysis computations. Functional analysis applies to many kinds of systems engaged in many kinds of activity, ranging from pumping blood to generating electricity to digesting food. As we'll see in Chapter 7, functional analysis (or, better, mechanistic explanation) applies to what are ordinarily called computing systems too, for ordinary computing systems engage in a certain type of activity—computation—in virtue of the functions performed by their components under normal conditions. For instance, Turing machines perform their computations in virtue of the activities performed by their active device (whose function is to write and erase symbols in appropriate ways) on their tape (whose function is to store symbols). But if we take computation in anything like the sense in which it is used in computability theory and computer science, we must conclude that most functional analyses do not ascribe computations to the systems they analyze.

If we like, we may start calling every function of every functionally analyzed system a computation. As a result, computation will acquire explanatory force by piggybacking on the explanatory force of functional analysis. But this also turns every system that is subject to functional analysis into a computing system. Functional analysis applies to artifacts, organisms, and their components, including stomachs and other paradigmatic examples of non-computing systems. This way of ascribing computations is too liberal to be directly relevant to computer science and cognitive science in the sense of using genuine computation (as opposed to a system's functions) to explain behavior. So appealing to functional analysis is a step in the right direction—the direction of an adequate account of computational explanation. But in order to be relevant to the sciences of computation, computational explanation cannot be *equated with* functional analysis—it must be restricted further.[8]

[8] To solve this problem, Bontly proposes to conjoin the approach to computational explanation based on functional analysis with a semantic account of computation (1998: 570). His proposal is ingenious but suffers from the limitations of semantic accounts, which I discussed in Chapter 3.

In Chapter 7 I will argue that computational explanation is a *special form* of functional analysis (or better, mechanistic explanation), which applies only to systems with special functional and structural properties. When this is done, we will finally have an adequate account of computational explanation, which will reinforce our rejection of pancomputationalism.

9. Leaving Pancomputationalism Behind

There are many ways to describe a system computationally. Different computational descriptions carry different ontological implications about whether the system itself performs computations. And the assertion that everything is describable as a computing system takes a different significance depending on which kind of computational description is at stake.

Most computational descriptions are forms of computational modeling, in which the computations are performed by the model in order to generate representations of the modeled system at subsequent times. Computational models need not ascribe computations to the systems they model, and a fortiori they need not explain their behavior by postulating computations internal to the systems they model. Even so, the claim that everything is describable by a computational model is only true in the most trivial sense. For computational models are approximate. If we care how good our approximations are—and if our models are to serve our scientific purposes, we'd better care—then whether something has a computational description depends on whether it can be computationally approximated to the degree of accuracy that is desired in a given case.

Some computational descriptions, however, ascribe computations to the systems themselves and are explanatory. They describe systems as rule-following, rather than merely rule-governed, because they explain the behavior of the systems by appealing to the rule they normally follow in generating their outputs from their inputs (and perhaps their internal states). This kind of computational description is suited to formulating genuinely explanatory theories of rule-following systems, and it applies only to very select systems, which manipulate special sorts of input and output in special sorts of way. As we shall see in more detail in Chapter 7, it is far from true that everything is a computing system in this sense.

Given all this, different sources of evidence need to be used to support different versions of the claim that something is a computing system, and the version that is relevant to computer science and cognitive science turns out to be empirical in nature. This is an important conclusion, because naturalistically inclined philosophers should prefer to settle computational theories of cognition empirically rather than a priori. Whether something is a computing system properly so called turns out to depend on whether it has certain mechanistic—i.e., structural as well as functional—properties.

As to the thesis that everything is a computing system, it turns out to be motivated by a superficial understanding of the role of computation in modeling. Once pancomputationalism is made more precise, it loses both plausibility and relevance to computer science and cognitive science. Given that, I will leave pancomputationalism behind and turn to the relation between functional analysis and mechanistic explanation. This will take us another step closer to the mechanistic account of computation.

5

From Functional Analysis to Mechanistic Explanation

1. Is Computational Explanation the same as Functional Analysis?

In previous chapters, I argued that traditional accounts of computation are inadequate vis-à-vis the desiderata laid out in Chapter 1. In this chapter, I will dig deeper into why they fail desideratum 2: computational explanation.[1] Insofar as traditional accounts attempt to capture computational explanation, they usually rely on functional analysis. The view is that functional analysis is a kind of explanation that is distinct and autonomous from mechanistic explanation and that functional analysis ascribes computations to a system (Fodor 1965, 1968b; Cummins 1983, 2000). Therefore, computational explanation is the same as functional analysis. Variants of this view differ on how to construe functional analysis or how widely it applies, but the core idea is the same.

I will argue that this traditional account of computational explanation is incorrect. One reason is that, as I pointed out in Section 8 of the previous chapter, not all functional analyses ascribe computations to the systems they analyze. Therefore, computational explanation is not functional analysis simpliciter but, at most, a special kind of it. To explain what distinguishes computational explanation from functional analysis (or, better, mechanistic explanation) more generally will be the burden of Chapter 7. Before we get to that, I need to examine the relation between functional analysis and mechanistic explanation (this chapter) and flesh out the relevant notion of mechanism (next chapter). As it turns out, a second flaw in the traditional account of computational explanation based on functional analysis is that it misconstrues the relation between functional analysis and mechanistic explanation.

Functional analysis and mechanistic explanation should not be confused with run-of-the-mill causal explanation. When we explain the behavior of a system, our explanation typically makes reference to causes that precede the behavior and make a difference to whether and how it occurs. For instance, we explain that

[1] This chapter is heavily indebted to Piccinini and Craver 2011, so Carl Craver deserves partial credit for most of what is correct here.

Anna ducked because she saw a looming ball. This is ordinary causal explanation, and it's *not* what I'm talking about.

By contrast, when scientists explain capacities such as stereopsis or working memory, they typically do so by showing that these complex capacities are made up of more basic capacities organized together. I focus exclusively on the latter sort of explanation, which is traditionally referred to as *functional analysis*.

I will argue that functional analyses, far from being distinct and autonomous from mechanistic explanations, are *sketches of mechanisms*, in which some structural aspects of a mechanistic explanation are omitted. Once the missing aspects are filled in, a functional analysis turns into a full-blown mechanistic explanation at one level of organization. By this process, functional analyses are seamlessly integrated with multilevel mechanistic explanations.

Caveat: I am *not* defending reductionism.[2] Instead, I am defending the integration of explanations at different mechanistic levels—explanatory unification is achieved through the integration of findings from different levels into a description of multilevel mechanisms.

In the next two sections, I discuss how functional analysis and computational explanation ended up being conflated in the literature and begin to disentangle the two. In the following section, I outline the received view that functional analysis is distinct and autonomous from mechanistic explanation. After that, Section 5 sketches the basic elements of mechanistic explanation, emphasizing that functional properties are an integral part of mechanisms. Sections 6–8 discuss the three main types of functional analysis, arguing that each is a sketch of a mechanism.

2. Psychological Theories as Functional Analyses

According to Fodor (1965), psychological theories are developed in two logically distinct phases. Fodor calls phase one theories *functional analyses* and phase two theories *mechanical analyses*. Fodor explicates the distinction between functional and mechanical analysis by the example of internal combustion engines. Functional analysis identifies the functions of engine parts, namely their contribution to the activities of the whole engine. For an internal combustion engine to generate motive power, fuel must enter the cylinders, where it detonates and drives the cylinders. In order to regulate the flux of fuel into the cylinders, functional analysis says there must be valves that are opened by valve lifters. Valve lifters contribute to the activities of the engine by lifting valves that let fuel into the cylinders.

[2] I do endorse reductionism in the sense that every concrete thing is made out of physical components and the organized activities of a system's components explain the activities of the whole. Setting aside dualism and spooky versions of emergentism, I take these theses to be uncontroversial. But my rejection of the autonomy of functional analysis is not reductionism as usually understood, which entails the rejection of multiple realizability. I offer an account of multiple realizability within a mechanistic framework in Piccinini and Maley 2014.

Given a functional analysis of an engine, mechanical analysis identifies physical structures that correspond to the functional analysis of the engine. In certain engines, camshafts are identified as physical structures that function as valve lifters, although in other engines the same function may be performed by other physical structures. By the same token, according to Fodor, phase one psychological theories identify psychological functions (functional analysis), whereas phase two psychological theories identify physiological structures that perform those functions (mechanical analysis). From his notion of functional analysis as explicated by his example of engines, Fodor infers that psychological theories have indefinitely many realizations or "models:" that is, different mechanisms can realize a given functional analysis (Fodor 1965, 174–5).

Fodor also argues that the relationship between phase one (functional analysis) and phase two (mechanical analysis) psychological theories is not a relation of reductive "microanalysis" in the sense of Oppenheim and Putnam 1958, in which there is a type-type correspondence between the predicates of the two descriptions (Fodor 1965, 177).

Fodor points out that his analysis of psychological theories has "a particular indebtedness" to a book by psychologist J. A. Deutsch (1960) and to an article by Hilary Putnam (1960) (Fodor 1965, 161). The Deutsch acknowledgment is straightforward enough because Fodor's account follows quite closely the account of psychological theories proposed by Deutsch, who also infers that there is a "theoretically infinite variety of counterparts" of any type of mechanism postulated by phase one psychological theories (Deutsch 1960, 13).

The Putnam acknowledgment is revealing. In his article, Putnam (1960) proposes an analogy between psychological states and Turing machine (TM) states. Prima facie, Fodor could have proposed his analysis of psychological theories independently of Putnam's. On one hand, TMs are individuated by a finite number of internal *state* types and what state transitions must occur under what conditions, regardless of what component types must make up the system that realizes the TM or what their functions must be. On the other hand, functional analyses—based on Fodor's examples and Deutsch's formulation—are specifications of mechanism types, consisting of different *component* types and their assigned functions without specifying the precise state types and state transitions that must occur within the analyzed system. A functional analysis of an engine does not come in the form of a TM table, nor is it obvious how it could be turned into a TM table or whether turning it into a TM table would have any value for explaining the functioning of the engine. TM tables can be analyzed into subroutines, and subroutines can be analyzed into sequences of elementary operations, but this is not a functional analysis in Deutsch's sense. Even the fact that both TM tables and functional analyses can be multiply realized seems to originate from different reasons. A functional analysis can be multiply realized because systems with different physical properties can perform the same concrete function (e.g., generate motive power), whereas a TM table can be

multiply realized because systems with different physical properties, no matter what functions they perform, can realize the same abstractly specified state transitions. At the very least, the thesis that the two are related requires analysis and argument. So, prima facie there is little reason to think that giving a functional analysis of a system is equivalent to describing that system by using a TM table. In fact, neither Fodor nor his predecessor Deutsch mention computers or TMs, nor do they infer, from the fact that psychological descriptions are functional analyses, that either the mind or the brain is a TM.

Nevertheless, when Fodor describes phase one psychological theories in general, he departs from his example of the engine and from Deutsch's view. Fodor describes psychological functional analyses as postulations of internal *states*, i.e., as descriptions that closely resembled TM descriptions: "Phase one explanations purport to account for behavior in terms of internal states" (Fodor 1965, 173). This way of describing functional analyses was clearly under the influence of Putnam's 1960 analogy between minds and TMs. In a subsequent book, where Fodor repeats his two-phase analysis of psychological theories, he explicitly attributes his formulation of phase one psychological theories to Putnam's 1960 analogy (Fodor 1968a, 109). In his 1965 article, however, Fodor does not discuss TMs explicitly, nor does he explain how his formulation of phase one psychological theories in terms of states squares with his example of the engine.

Thus, Fodor (1965) introduced in the literature both the notion that psychological theories are functional analyses and the notion that functional analyses are like TM tables in that they are descriptions of transitions between state types. Both themes became very influential in the philosophy of psychology.

3. Functional Analysis and Explanation by Program Execution

Fodor's 1965 description of phase one psychological theories as having the same form as TM tables paved the way for the later *identification* of psychological functional analysis and computational explanation. In a paper published a few years later (Fodor 1968b), Fodor repeats his view, already present in Fodor 1965, that psychological theories provide descriptions of psychological functions.

But this time, he adds that psychological theories are canonically expressed as lists of instructions: "the paradigmatic psychological theory is a list of instructions for producing behavior" (Fodor 1968b, 630). Although this flies in the face of the history of psychology, which is full of illustrious theories that are not formulated as lists of instructions (e.g., Freud's psychoanalysis, or Skinner's behaviorism), Fodor does not offer evidence for this statement.[3] Fodor says that each instruction in a psychological

[3] At the time of Fodor's writing, though, some psychologists did propose such a view of psychological theories (Miller, Galanter, and Pribram, 1960), and at least according to Gilbert Harman (personal correspondence), their work influenced Fodor.

theory can be further analyzed in terms of a list of instructions, which can also be analyzed in the same way. This does not lead to infinite regress because for any organism, there is a finite list of elementary instructions in terms of which all psychological theories for that organism must ultimately be analyzed. This type of explanation of behavior, based on lists of instructions, is explicitly modeled by Fodor on the relationship between computers, computer programs, and the elementary instructions in terms of which programs are ultimately formulated.[4]

Of course, the thesis that functional analysis is the same as explanation by program execution does not entail by itself that minds are programs, nor does Fodor suggest that it does. The mind would be a program only if all psychological capacities, states, and processes could be correctly and completely explained by appeal to program execution. For Fodor, whether this is the case is presumably an empirical question. Still, Fodor's paper firmly inserted into the philosophical literature a thesis that begs the question of whether all psychological capacities can be explained by program execution: the thesis that psychological theories are canonically formulated as lists of instructions for producing behavior. This left no room for alternative explanations to be considered. In particular, the general type of functional analysis identified by Deutsch (1960), which explained mental capacities by postulating types of components and their functions, was transformed, through Fodor's 1965 reinterpretation, into a kind of computational explanation. After that, the mongrel of functional analyses and computational explanation remained in the literature, where there are many statements to the effect that psychological theories are functional analyses, and that psychological functional analyses (or sometimes all functional analyses) are computational explanations.[5] In later chapters we shall see that, on the contrary, only very specialized kinds of mechanistic explanations are computational. But first we need to see that functional analyses are mechanism sketches.

4. The Received View: Functional Analysis as Distinct and Autonomous from Mechanistic Explanation

As we've seen, there is consensus that many high level phenomena, such as psychological capacities, are explained by *functional analysis*. Functional analysis is the

[4] Notice that although computer programs replace TM tables in Fodor's account of functional analysis, there are important differences between these two types of description (cf. Chap. 11).

[5] For example, similar conflations can be found in works by Dennett (1975, 1978); Cummins (1975, 1983); Marr (1982); and Churchland and Sejnowski (1992). Even Harman, who criticizes Fodor and Putnam's construal of functional analysis, follows their lead in this respect. Harman calls Fodor's explanations by functional analysis *narrow* because they attempted to explain an organism only in terms of its internal states, inputs, and outputs, without reference to how the system interacts with its environment. Harman argues that a complete explanation of an organism requires a *wide* functional story, i.e., a story that takes into account the relations between the organism and its environment. Nevertheless, even Harman identifies the narrow functional story about an organism with the description of the organism by a program (Harman 1988, esp. 240–1).

analysis of a capacity in terms of the functional properties (sub-capacities, functions) of a system and their organization. Three main types of functional analysis may be distinguished, depending on which functional properties they invoke. One type is *task analysis*: the decomposition of a capacity into sub-capacities and their organization (Cummins 1975, 1983, 2000; cf. also Fodor 1968b and Dennett 1978). A second type is *functional analysis by internal states*: an account of how a capacity is produced in terms of a set of internal states and their mutual interaction (Fodor 1965, 1968b; Stich 1983; cf. also Putnam 1960, 1967a, b). A third type is *boxology*: the decomposition of a system into a set of functionally individuated components (black boxes), the processes they go through, and their organization (Fodor 1965, 1968b, 1983). Like task analysis, boxology involves decomposing a system's capacities into sub-capacities, but it but also involves decomposing the system into subsystems that perform its sub-capacities; by contrast, task analysis may attribute the sub-capacities to the system itself without decomposing it into subsystems.

I focus on these three types of functional analysis because they figure prominently in the literature. I will address each kind of functional analysis separately and argue that each amounts to a mechanism sketch. If there are yet other types of functional analysis, the present argument can be extended to cover them. For I will argue that a complete constitutive explanation of a phenomenon in terms of functional properties must respect constraints imposed by the structures that possess those functional properties—that is, it requires fitting the functional properties within a mechanism.

The received view of the relationship between functional analysis and mechanistic explanation may be summarized as follows:

> **Distinctness**: Functional analysis and mechanistic explanation are distinct kinds of explanation.
>
> **Autonomy**: Functional analysis and mechanistic explanation are autonomous from one another.

One way to see that proponents of the received view endorse distinctness is that they often claim that a complete explanation of a capacity includes both a functional analysis and a matching mechanistic explanation.[6] This presupposes that functional analyses are distinct from mechanistic explanation.

Distinctness is defended in slightly different ways depending on which form of functional analysis is at issue. With respect to task analysis, distinctness has been defended as follows. Unlike mechanistic explanation, which attributes sub-capacities

[6] E.g.:

> Explanation in psychology consists of a functional analysis and a mechanistic analysis: a phase one theory and a determination of which model of the theory the nervous system of the organism represents (Fodor 1965, 177).
>
> Functional and mechanistic explanations must be matched to have a complete explanation of a capacity (Cummins 1983, 31).

to the *components* of a mechanism, task analysis need not attribute sub-capacities to the components of the system because the sub-capacities may all belong to the whole system.[7]

While talking about components *simpliciter* may be enough to distinguish between task analysis and mechanistic explanation, it is insufficient to distinguish between functional analysis in general and mechanistic explanation. This is because some functional analyses—specifically, functional analyses that appeal to black boxes—do appeal to components, although such components are supposed to be individuated purely functionally, by what they do. To avoid ambiguity, I use the term *functional components* for components as individuated by their functional properties and *structural components* for components as individuated at least in part by their structural properties.

Paradigmatic structural properties include the size, shape, location, and orientation of extended objects. Anatomists tend to study structures in this sense, as do x-ray crystallographers. Paradigmatic functional properties include being a neurotransmitter, encoding an episodic memory, or generating shape from shading. Physiologists tend to study function. Nothing in the present argument turns on there being a metaphysically fundamental divide between functional and structural properties; indeed, if I am right, we cannot characterize functions without committing ourselves to structures and vice versa.

Those who think of functional analysis in terms of internal states and boxology defend distinctness on the grounds that internal states or black boxes are individuated solely by their functional relations with each other as well as with inputs and outputs, and not by their structural properties. While mechanistic explanation appeals to the structural features of the components of the mechanism, functional analysis allegedly does not.[8]

Distinctness is a necessary condition for autonomy: if functional analysis is *a kind of* mechanistic explanation, as I argue, then functional analysis cannot be *autonomous from* mechanistic explanation. But distinctness is not *sufficient* for autonomy: two explanations may be distinct from each other and yet mutually dependent. Nevertheless, those who endorse distinctness typically endorse autonomy as well—

[7] Cf. Cummins:

> Form-function correlation is certainly absent in many cases . . . and it is therefore important to keep functional analysis and componential analysis [i.e., mechanistic explanation] conceptually distinct. Componential analysis of computers, and probably brains, will typically yield components with capacities that do not figure in the analysis of capacities of the whole system (Cummins 1983, 2000, 125).

[8] Cf. Fodor:

> If I speak of a device as a "camshaft," I am implicitly identifying it by reference to its physical structure, and so I am committed to the view that it exhibits a characteristic and specifiable decomposition into physical parts. But if I speak of the device as a "valve lifter," I am identifying it by reference to its function and I therefore undertake no such commitment (Fodor 1968b, 113).

in fact, I suspect that defending autonomy is the primary motivation for endorsing distinctness.[9]

What is autonomy (in the relevant sense)? There are many kinds of autonomy. One scientific enterprise may be called *autonomous* from another if the former can choose (i) which phenomena to explain, (ii) which observational and experimental techniques to use, (iii) which vocabulary to adopt, and (iv) the precise way in which evidence from the other field constrains its explanations. The term 'autonomy' is sometimes used for one or more of the above (e.g., Aizawa and Gillett ms. argue that psychology is autonomous from neuroscience in sense (iv)). I have no issue with these forms of autonomy. Higher level sciences may well be autonomous from lower level sciences in these four ways.

In another sense, a scientific explanation may be said to be autonomous from another just in case the former refers to properties that are distinct from and irreducible to the properties referred to by the latter. This form of autonomy is sometimes claimed to obtain between psychology and neuroscience. For instance, psychological properties are sometimes claimed to be distinct from and irreducible to the neural properties that realize them.

This latter form of autonomy thesis suggests that properties are stacked into levels of being. It is not clear how levels of being can be distinct from one another without being ontologically redundant (Kim 1992; Heil 2003). Doing justice to this topic would take us beyond the scope of this Chapter and is mostly orthogonal to our topic. My solution is that higher level properties are proper subsets of the lower level properties that realize them (Piccinini and Maley 2014). If so, then higher level properties are neither identical and reducible to lower level properties nor fully distinct from lower level properties. Properties at different mechanistic levels stand in the superset-subset relation; therefore, they are not ontologically autonomous from one another.

In yet another sense, autonomy may be said to obtain between either laws or theories when they are irreducible to one another (cf. Fodor 1997, 149; Block 1997), regardless of whether such laws or theories refer to ontologically distinct levels of being. As I pointed out before, I am not defending reductionism. Nevertheless, I reject this kind of autonomy. The dichotomy between reduction and autonomy is

[9] E.g.:

> The conventional wisdom in the philosophy of mind is that psychological states are functional and the laws and theories that figure in psychological explanations are autonomous (Fodor 1997, 149).
>
> Why . . . should not the kind predicates of the special sciences cross-classify the physical natural kinds? (Fodor 1975, 25; see also Fodor 1997, 161–2).
>
> We could be made of Swiss cheese and it wouldn't matter (Putnam 1975b, 291).

It is worth noting that Fodor's writings on psychological explanation from the 1960s were less sanguine about autonomy than his later writings, although he was already defending distinctness (cf. Aizawa and Gillett ms.).

a false one. Neither psychology nor neuroscience discovers the kind of law or theory for which talk of reduction makes the most sense (cf. Cummins 2000). What they do discover are aspects of mechanisms to be combined in full-blown multilevel mechanistic explanations. Therefore, I reject autonomy as irreducibility of laws or theories in favor not of reduction but of explanatory integration.

A final kind of autonomy thesis maintains that two explanations are autonomous just in case there are no *direct constraints* between them. Specifically, some authors maintain that the functional analysis and the mechanistic explanation of one and the same phenomenon put no direct constraints on each other.[10] While proponents of this kind of autonomy are not very explicit in what they mean by "direct constraints," the following seems to capture their usage: a functional analysis directly constrains a mechanistic explanation if and only if the functional properties described by a functional analysis restrict the range of structural components and component organizations that can exhibit those capacities; a mechanistic explanation directly constrains a functional analysis if and only if the structural components and component organization described by the mechanistic explanation restrict the range of functional properties exhibited by those components thus organized.

Of course, every participant in this debate agrees on one important (if obvious) *indirect* constraint: the mechanism postulated by a true mechanistic explanation must *realize* the functional system postulated by a true functional analysis.[11] Aside from that, autonomists suggest that the two explanatory enterprises proceed independently of one another. As this *no-direct-constraints* kind of autonomy is generally interpreted, it entails or at least strongly suggests that those who are engaged in

[10] E.g.:

> Phase one explanations [i.e., functional analyses by internal states] purport to account for behaviour in terms of internal states, but they give no information whatever about the mechanisms underlying these states (Fodor 1965, 173).
>
> [F]unctional analysis puts very indirect constraints on componential analysis (Cummins 1983, 29; 2000, 126).

While these specific statements by Cummins and Fodor suggest the autonomy of mechanistic explanation from functional analysis rather than the converse, the rest of what they write makes clear that they also maintain that functional analysis is autonomous from mechanistic explanation. For an even stronger formulation of the "no constraint principle" that is pervasive in the literature on functional analysis, see Aizawa and Gillett ms.

[11] Cf. Cummins:

> Ultimately, of course, a complete theory for a capacity must exhibit the details of the target capacity's realization in the system (or system type) that has it. Functional analysis of a capacity must eventually terminate in dispositions whose realizations are explicable via analysis of the target system. Failing this, we have no reason to suppose we have analyzed the capacity as it is realized in that system (Cummins 1983, 31; 2000, 126).

Although I along with every other participant in this debate assume that functional systems are realized by mechanisms, some dualists disagree; they maintain that a functional system may be a non-physical, non-mechanistically-implemented system. I disregard this possibility on the usual grounds of causal closure of the physical and lack of an adequate account of the interaction between physical and non-physical properties. In any case, dualism is not my present target.

functional analysis need not know or pay attention to which mechanisms are present in the system; by the same token, those who are engaged in mechanistic explanation need not know or pay attention to how a system is functionally analyzed. For instance, according to this kind of autonomy, psychologists and neuroscientists need not pay attention to what the other group is doing—except that, in the end, of course, their explanations ought to match.

The assumption of autonomy as lack of direct constraints is appealing. It neatly divides the explanatory labor along traditional disciplinary lines and thus relieves members of each discipline of learning overly much about the other discipline. On one hand, psychologists are given the task of uncovering the functional organization of the mind without worrying about what neuroscientists do. On the other hand, neuroscientists are given the task of discovering neural mechanisms without having to think too hard about how the mind works. Everybody can do their job without getting in each other's way. Someday, if everything goes right, the functional analyses discovered by psychologists will turn out to be realized by the neural mechanisms discovered by neuroscientists. And yet, according to this autonomy thesis, neither the functional properties nor the structures place direct constraints on one another.

A number of philosophers have resisted autonomy understood as lack of direct constraints. Sometimes defenders of mechanistic explanation maintain that functional analysis—or "functional decomposition," as Bechtel and Richardson call it—is just a step towards mechanistic explanation (Bechtel and Richardson 1993, 89–90; Bechtel 2008, 136; Feest 2003). Furthermore, many argue that explanations at different mechanistic levels directly constrain one another (Bechtel and Mundale 1999: P.S. Churchland 1986; Craver 2007; Feest 2003; Keeley 2000; Shapiro 2004).

I agree with both of these points. But they do not go deep enough. The same authors who question autonomy seem to underestimate the role that distinctness plays in defenses of autonomy. Sometimes proponents of mechanistic explanation even vaguely hint or assume that functional analysis is *the same as* mechanistic explanation (e.g., Bechtel 2008, 140; Feest 2003; Glennan 2005), but they don't articulate and defend that thesis. So long as distinctness remains in place, defenders of autonomy have room to resist the mechanists' objections. Autonomists may insist that functional analysis, properly so called, is autonomous from mechanistic explanation after all. By arguing that functional analysis is actually a kind of mechanistic explanation, we get closer to the bottom of this dialectic. Functional analysis cannot be autonomous from mechanistic explanation because the former is just an elliptical form of the latter.

In the rest of this chapter, I argue—along with others—that functional analysis and mechanistic explanation are not autonomous because they constrain each other; in addition, I argue that they can't possibly be autonomous in this sense because functional analysis is just a kind of mechanistic explanation. Functional properties are an undetachable aspect of mechanistic explanations. Any given explanatory text might accentuate the functional properties at the expense of the structural properties,

but this is a difference of emphasis rather than difference in kind. The target of the description in each case is a mechanism.

In the next section, I briefly introduce the contemporary notion of mechanistic explanation and show that mechanistic explanation is rich enough to incorporate the kinds of functional properties postulated by functional analysis. In subsequent sections I argue that each kind of functional analysis is a mechanism sketch—an elliptical description of a mechanism.

5. Mechanistic Explanation

Mechanistic explanation is the explanation of the capacities (functions, behaviors, activities) of a system as a whole in terms of some of its components, their properties and capacities (including their functions, behaviors, or activities), and the way they are organized together (Bechtel and Richardson 1993; Machamer, Darden, and Craver 2000, Glennan 2002). Components have both functional properties—their activities or manifestations of their causal powers, dispositions, or capacities—and structural properties—including their location, shape, orientation, and the organization of their sub-components. Both functional and structural properties of components are aspects of mechanistic explanation.

Mechanistic explanation has also been called "system analysis," "componential analysis" (Cummins 1983, 28–9; 2000, 126), and "mechanistic analysis" (Fodor 1965). Constructing a mechanistic explanation requires decomposing the capacities of the whole mechanism into subcapacities (Bechtel and Richardson 1993). This is similar to task analysis, except that the subcapacities are assigned to structural components of a mechanism. The term "structural" does *not* imply that the components involved are neatly spatially localizable, have only one function, are stable and unchanging, or lack complex or dynamic feedback relations with other components. Indeed, a structural component might be so distributed and diffuse as to defy tidy structural description, though it no doubt has one if we had the time, knowledge, and patience to formulate it.

Mechanistic explanation relies on the identification of relevant components in the target mechanism (Craver 2007). Components are sometimes identified by their structural properties. For instance, some anatomical techniques are used to characterize different parts of the nervous system on the basis of the different kinds of neurons they contain and how such neurons are connected to one another. Brodmann decomposed the cortex into distinct structural regions by characterizing cytoarchitectonic differences in different layers of the cortical parenchyma. Geneticists characterize the primary sequence of a gene. Such investigations are primarily directed at uncovering structural features rather than functional ones.

But anatomy alone cannot yield a mechanistic explanation—mechanistic explanation requires identifying the functional properties of the components. For example, case studies of brain-damaged patients and functional magnetic resonance imaging

are used to identify regions in the brain that contribute to the performance of some cognitive tasks. Such methods are crucial in part because they help to identify regions of the brain in which relevant structures for different cognitive functions might be found. They are also crucial to assigning functions to the different components of the mechanism. Each of these discoveries is a kind of progress in the search for neural mechanisms.

Functional properties are specified in terms of effects on some medium or component under certain conditions. Different structures and structure configurations have different functional properties. As a consequence, the presence of certain functional properties within a mechanism constrains the possible structures and configurations that might exhibit those properties. Likewise, the presence of certain structures and configurations within a mechanism constrains the possible functions that might be exhibited by those structures and configurations.

Mechanistic explanation is iterative in the sense that the functional properties (functions, capacities, activities) of components can also often be mechanistically explained. Each iteration in such decomposition adds another level of mechanisms to the mechanistic articulation of a system, with levels arranged in component/subcomponent relationships and ultimately (if ever) bottoming out in components whose behavior has no mechanistic explanation.

Descriptions of mechanisms—mechanism schemas (Machamer, Darden, and Craver 2000) or models (Glennan 2005; Craver 2006)—can be more or less complete at one or more levels of organization. Incomplete models—with gaps, question-marks, filler-terms, or hand-waving boxes and arrows—are mechanism sketches. Mechanism sketches are incomplete because they leave out crucial details about how the mechanism works. Sometimes a sketch provides just the right amount of explanatory information for a given context (classroom, courtroom, lab meeting, etc.). Furthermore, sketches are often useful guides to the future development of a mechanistic explanation. Yet there remains a sense in which mechanism sketches are incomplete or elliptical.

The common crux of mechanistic explanation, both in its current form and in forms stretching back through Descartes to Aristotle, is to reveal the causal structure of a system. Explanatory models are evaluated as good or bad to the extent that they capture, even dimly at times, aspects of that causal structure. The present argument is that the motivations guiding prominent accounts of functional analysis commit its defenders to precisely the same norms of explanation that mechanists embrace.

6. Task Analysis

A task analysis breaks a capacity of a system into a set of sub-capacities and specifies how the sub-capacities are (or may be) organized to yield the capacity to be explained. Cummins calls the specification of the way sub-capacities are organized into capacities a "program" or "flow-chart" (Cummins 1975, 1983, 2000), which is an

instance of the conflation of functional analysis and computational explanation that I discussed earlier in Section 3. The organization of capacities may also be specified by, say, a system of differential equations linking variables representing various sub-capacities. What matters is that one specifies how the various sub-capacities are combined or interact so that they give rise to the capacity of the system as whole.

Cummins remains explicitly ambiguous about whether the analyzing sub-capacities are assigned to the whole mechanism or to its components. The reason is that *at least in some cases*, the functional analysis of a capacity "seems to put no constraints at all on . . . componential analysis [i.e., mechanistic explanation]" (1983, 30). I discuss the different types of functional analysis separately. First, "functional analyses" that assign sub-capacities to *structural* components are mechanistic explanations (Craver 2001). Second, functional analyses that assign sub-capacities to *functional* components (black boxes) are boxological models (see Section 8). Finally, functional analyses that assign sub-capacities to the whole system rather than its components are task analyses. Cummins' examples of capacities subject to task analysis are assembly line production, multiplication, and cooking (Cummins 1975, 1983, 2000). Task analysis is the topic of this section.

Task analysis of a capacity is one step in its mechanistic explanation in the sense that it partitions the phenomenon to be explained into units that correspond to manifestations of specific causal powers, which are possessed by specific components. For example, contemporary memory researchers partition semantic memory into encoding, storage, and retrieval processes, appeal to each of which will be required to explain performance on any memory task. Contra Cummins, the partition of the phenomenon places direct constraints on components, their functions, and their organization. For if a sub-capacity is a genuinely explanatory part of the whole capacity, as opposed to an arbitrary partition (a mere piece or temporal slice), it must be exhibited by specific components or specific configurations of components. Systems have the capacities they have in virtue of their components and organization.

Consider Cummins's examples: stirring (a sub-capacity needed in cooking) is the manifestation of (parts of) the cook's locomotive system coupled with an appropriate stirring tool as they are driven by a specific motor program; multiplying single-digit numbers (a sub-capacity needed in multiplying multiple digit numbers) is the manifestation of a memorized look-up table in cooperation with other parts of the cognitive system; and assembly line production requires different machines and operators with different skills at different stages of production. Likewise, the correctness of the above-mentioned task analysis of memory into encoding, storage, and retrieval depends on whether there are components that encode, store, and retrieve memories.

Task analysis is constrained, in turn, by the available components and modes of organization. If the study of brain mechanisms forces us to lump, split, eliminate, or otherwise rethink any of these sub-capacities, the functional analysis of memory will have to change (cf. Craver 2004). If the cook lacks a stirring tool but still manages to

combine ingredients thoroughly, we should expect that the mixture has been achieved by other means, such as shaking. If someone doesn't remember the product of two single digit numbers, she may have to do successive additions to figure it out. The moral is that if the components predicted by a given task analysis are not there or are not functioning properly, you must rule out that task analysis in favor of another for the case in question (cf. Craver and Darden 2001).

In summary, a task analysis is a mechanism sketch in which the capacity to be explained is articulated into sub-capacities, and most of the information about components is omitted. Nevertheless, the sub-capacities do place direct constraints on which components can engage in those capacities. For each sub-capacity, we expect a structure or configuration of structures that has that capacity. This guiding image underlies the very idea that these are explanations, that they reveal the causal structure of the system. If the connection between analyzing tasks and components is severed completely, then there is no clear sense in which the analyzing sub-capacities are aspects of the *actual* causal structure of the system as opposed to arbitrary partitions of the system's capacities or merely possible causal structures. Indeed, components often call attention to themselves as components only once it is possible to see them as performing what can be taken as a unified function.

So task analysis specifies sub-capacities to be explained mechanistically, places direct constraints on the kinds of components and modes of organization that might be employed in the mechanistic explanation, and is in turn constrained by the components, functional properties, and kinds of organization that are available.[12]

At this point, a defender of the distinctness and autonomy of functional analysis may object that although many task analyses are first-stage mechanistic theories, as I maintain, the really interesting cases, including the ones most pertinent to cognitive phenomena, are not. The paradigmatic putative example is the *general purpose digital computer*. General purpose digital computers can do an indefinite number of things depending on how they are programmed. Thus, one might think, a task analysis of a computer's capacities places no direct constraints on its structural components and the structural components place no direct constraints on task analysis, because the

[12] This account of task analysis is borne out by the historical evolution of task analysis techniques in psychology. Psychologists developed several techniques for analyzing complex behavioral tasks and determine how they can be accomplished efficiently. Over time, these techniques evolved from purely behavioral techniques, in which the sole purpose is to analyze a task or behavior into a series of operations, into *cognitive* task analysis, which also aims at capturing the agents' cognitive states and their role in guiding their behaviors. To capture the role of cognitive states, agents ought to be decomposed into their components (Crandall, Klein and Hoffman 2006, 98). The motivation for such a shift is precisely to capture more accurately the way agents solve problems. Focusing solely on sequences of operations has proved less effective than analyzing also the agents' underlying cognitive systems, including their components. As the perspective I am defending would predict, task analysis in psychology has evolved from a technique of behavioral analysis, closer to task analysis as conceived by Cummins, towards a more mechanistic enterprise.

components are the same regardless of which capacity a computer exhibits. The only constraint is indirect: the computer must be general purpose, so it must have general purpose components. By extension, if the same kind of autonomous task analysis applies to human behavior, then the type of task analysis with which the received view is concerned is not a mechanism sketch.

This objection makes an important point but draws the wrong conclusion. True, general purpose digital computers are different from most other systems precisely because they can do so many things depending on how they are programmed. But computers are still mechanisms, and the explanation of their behavior is still mechanistic. Furthermore, the task analysis of a general purpose digital computer does place direct constraints on its mechanistic explanation and vice versa; in fact, even the task analysis of a general purpose computer is just an elliptical mechanistic explanation. These theses will be substantiated in detail in Chapter 11. Here is a quick preview of what's coming.

To begin with, the explanation of the capacity to execute programs is mechanistic: a processor executes instructions over data that are fed to it and returns results to other components (Figure 5.1). In addition, the explanation of any specific computational capacity the computer has is also mechanistic: the processor executes *these particular instructions*, and it is *their* execution that results in the capacity. Because the program execution feature is always the same, in many contexts it is appropriate to omit that part of the explanation. But the result is *not* a non-mechanistic task analysis; it is, again, an elliptical one. It is a mechanistic explanation in which most of the mechanism is left implicit—the only part that is made explicit is the executed program.

A defender of the received view may reply as follows. Granted, whether a system is a general purpose computer is a matter of which mechanisms it contains. But *if* we can assume that a system is a general purpose digital computer, *then* we can give task analyses of its behavior that say nothing about its mechanisms. For when we describe

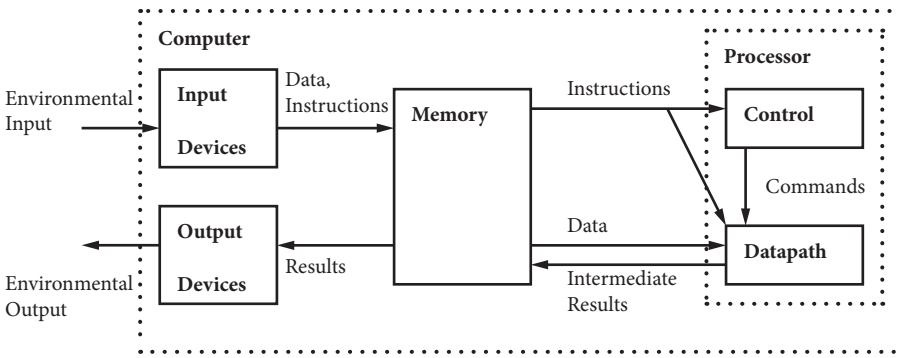


Figure 5.1 Functional organization of a general purpose digital computer.

the program executed by the computer, we are not describing any components or aspects of the components.[13] We are abstracting away from the mechanisms.

This objection misconstrues the role programs play in the explanation of computer behavior. There are two cases. First, if a computer is hardwired to perform a certain computation, the computer's behavior may still be described by a flow-chart or program.[14] In the case of hardwired computations, it is true that a program is not a component of the computer. But it is false that the program description is independent of the description of the machine's components. In fact, the program describes precisely the activity of the special purpose circuit that is hardwired to generate the relevant computation. A circuit is a structural component. Therefore, a task analysis of a computer that is hardwired to perform a computation is a precise description of a specific structural component. Incidentally, an analogous point applies to neural networks that come to implement a task analysis through the adjustment of weights between nodes.

The second case involves those computers that produce a behavior by *executing* a program—that is, by being caused to produce the behavior by the presence of the program within them. Here, the program itself is a stable state of one or more computer components. Programs are typically stored in memory components and sent to processors one instruction at a time. If they weren't physically present within computers, digital computers could not execute them and would not generate the relevant behaviors. Thus programs are a necessary feature of the mechanistic explanation of computers' behaviors; any task analysis of a computer in terms of its program is an elliptical mechanistic explanation that provides only the program and elides the rest of the mechanism (including, typically, the subsidiary programs that translate high level code into machine executable code).

In addition, a good task analysis of a general purpose digital computer is much more precise than a simple series of operations. It specifies that the machine takes two kinds of input (program and data) coded in certain specific ways (say, using high level programming language Java), performs certain specific manipulations of the data based on the program, and yields outputs that are also coded in certain specific ways. Lots of details go into this. The details of the coding, program, and operations place high level constraints on the structures that can implement such a task analysis. Then, task analysis can be iterated by analyzing the tasks executed by the operating system, the assembler, the compiler, and finally the circuits. The machine language places direct constraints on the circuits by requiring memory registers of a certain size, specifying which bits they must contain, requiring datapath circuits that perform certain operations, etc.

[13] Cf. Cummins on programs: "programs aren't causes but abstract objects or play-by-play accounts" (Cummins 1983, 34).

[14] In these cases, I find it misleading to say, as Cummins and others do, that the computer is *executing* the program, because the program is not an entity in its own right that plays a causal role. The same point applies to computers that are programmed by rearranging the connections between their components.

A final objection might be that some computational models focus on the flow of information through a system rather than the mechanisms that process the information (cf. Shagrir 2006b, 2010a). In such cases, someone might say, nothing is added to the explanation by fleshing out the details of how the information is represented and processed. Certainly, many computational explanations in psychology and neuroscience have this form. My response is that such descriptions of a system place direct constraints on any structures that can possibly process such information—on how the different states of the system can be constructed, combined, and manipulated—and are in turn constrained by the structures to be found in the system. It is, after all, an empirical matter whether the brain has structural components that satisfy a given informational description, that is, whether the neuronal structures in question can sustain the information processing that the model posits (under ecologically and physiologically relevant conditions). If they cannot, then the model is a false hypothesis—a merely how-possibly model that does not describe how the system actually works. Computational or informational explanations are still tethered to structural facts about the implementing system. Once we know the information processing task, we might think that details about how the information is encoded and manipulated are no longer relevant, and in some explanatory contexts this is true, but details about how the information is encoded and manipulated are, in fact, essential to confirm our hypotheses about information processing.

My reason for thinking that task analysis boils down to (elliptical) mechanistic explanation can be summarized as follows. Of course, there are many ways of saying how a system exhibits a capacity—some more detailed than others, some more fine-grained than others. Sometimes we abstract away from some performance details or describe only course-grained operations, depending on our explanatory goals. The usual contextual factors that govern explanation apply to task analysis too. That being said, the question remains: are these task analyses accurate descriptions of how the system performs its task? Either there is a correct task analysis for a given system capacity at a given time or not.

If explanations must be true (allowing for approximations and idealizations in the explanatory texts), a correct task analysis for a given system capacity amounts to there being units of activity between inputs and outputs that transform the relevant inputs and internal states into outputs. In real systems, structural components lie between the input and the output, and they are organized to exhibit that capacity. Whether the components satisfy the given task analysis depends on whether the components include structures that complete each of the tasks. If no such structures can be identified in the system, then the task analysis must be incorrect.[15]

[15] Lest there be some confusion on this point, I emphasize again that components need not be neatly localizable, visible, or spatially contained within a well-defined area. Any complex particular with a robustly detectable configuration of structural properties might count as a component.

One possible reply is that the components that satisfy a task analysis may be purely functional components, individuated by what they do rather than by their structural properties. I will address this idea later in this chapter in Section 8.

Another possible reply is that there is no unique correct task analysis for a given system behavior (at a given level of detail or grain). Cummins sometimes seems to embrace this view (see Cummins 1983, 43). Perhaps my critic maintains that the beauty of functional analysis is that it allows us to rise above the gory details, details that often vary from species to species, individual to individual, and time to time, and thereby to "capture" features of the causal structure of the system that are invisible when we focus on the microstructural details. She might think that any model that describes correctly and compactly the behavior of the system is equally explanatory of the system's behavior. Or she might think that higher-level descriptions can render the phenomenon intelligible even when they do not have any tidy echo in microstructural details.

I agree that models can be predictively adequate and intellectually satisfying even when they fail to describe mechanisms. My opponent, however, maintains that predictive adequacy and/or intelligibility are enough for explanation. But this view of explanation is untenable. Neither predictive adequacy nor intelligibility is sufficient for explanation. We can predict an empty gas tank without knowing how it came to be empty. And anyone who has ever misunderstood how something works is familiar with the idea that intelligible explanations can be terrible explanations. Some explanations make phenomena intelligible, but not all intelligible models are explanatory. In short, to give up on the idea that there is a uniquely correct explanation, and to allow that any predictively adequate and/or intelligible model is explanatory, is essentially to give up on the idea that there is something distinctive about explanatory knowledge. No advocate of functional analysis should want to give that up.

Either there is a unique correct task analysis, in which case the underlying mechanism must have structural components corresponding to the sub-tasks in the analysis, or there is not a unique correct task analysis, in which case task analysis has been severed from the actual causal structure of the system and does not count as explanation. In short, either task analysis is an elliptical form of mechanistic explanation or it is no explanation at all.

7. Functional Analysis by Internal States

The capacities of a system, especially cognitive capacities, are sometimes said to be explained by the system's internal states (and internal processes, defined as changes of internal states). Common examples of internal states include propositional attitudes, such as beliefs and desires, and sensations, such as pain. In an important strand of the philosophy of psychology, internal states are held to be functional states, namely, states that are individuated by their relations to inputs, outputs, and other

internal states (Putnam 1960, 1967a, b; Fodor 1965, 1968b; Block and Fodor 1972). Within this tradition, the functional relations between inputs, outputs, and internal states are said to constitute the functional organization of the system, on which the functional analysis of the capacities of the system is based.[16]

This account of mental states as internal functional states has been challenged. But my concern here is not whether the account is adequate. My concern is, rather, how functional analysis by internal functional states relates to mechanistic explanation.[17] In order to assess functional analysis by internal states, we must ask the further question: In what sense are such functional states *internal*?

The notion of state may be taken as primitive or analyzed as the possession of a property at a time. Either way, in general, there is no obvious sense in which a state of a system, per se, is internal to the system. For example, consider the distinction between the solid, liquid, and gaseous states of substances such as water. There is no interesting sense in which the state of being liquid, solid, or gaseous is internal to samples of water or any other substance. Of course, the constitutive explanation for why, at certain temperatures, water is solid, liquid, or gaseous involves the components of water (H_2O molecules) and their temperature-related state, which, together with other properties of the molecules (such as their shape), generates the molecular organization that constitutes the relevant global state of water samples. In this explanation we find a useful notion of internal state because individual water molecules are contained within the admittedly imprecise spatial boundaries of populations of water molecules.

The moral is this: in general, a system's states (*simpliciter*) are not internal in any interesting sense; they are global, system-level states. But there is something interestingly internal to the state of the *components* of a system, which play a role in explaining the global state of a system (as well as its behavior). This is because the components are *inside* the system. Are the internal states invoked in functional analysis system-level states or states of components? The qualifier "internal" suggests the latter.[18]

[16] Although this functionalist account is often said to be neutral between physicalism and dualism, this is an oversimplification. The internal states interact with the inputs and outputs of the system, which are physical. Thus, if the internal states were non-physical, they must still be able to interact with the physical inputs and outputs. This violates the causal closure of the physical and requires an account of the interaction between physical and non-physical states. So, while a version of functionalism may be *defined* to be compatible with dualism, only physicalistic versions of functionalism are metaphysically respectable.

[17] Notice that insofar as we are dealing with explanations of cognitive capacities, we are focusing on states that are internal to the cognitive system, whether or not the system's boundaries coincide with the body or nervous system of the organism.

[18] In any case, etymology does support my conclusion. Putnam (1960) imported the notion of an internal state into the philosophy of psychology as part of his analogy between mental states and Turing machine states. Turing described digital computers as having "internal states" (Turing 1950) that belong to the read-write head, which is the active component of Turing machines. In the case of digital computers, internal states are states of an internal component of the computer, such as the memory or processor. Initially, Putnam did not call states "internal" but "logical" (1960, 1967a), and then he called them

Here is why the internal states postulated by a functional analysis must be states of the system's components. Functional analysis by internal states postulates a system of multiple states. Such states are capable of interacting with inputs, outputs, and each other, in order to produce novel states and outputs from previous states and inputs. Notice three features of these systems. First, it must be possible for many states to occur at the same time. Second, inputs and outputs enter and exit through specific components of the system. Third, inputs and outputs are complex configurations of different physical media, such as light waves, sound waves, chemical substances, and bodily motions.

The only known way to construct a system of states that can occur at the same time and mediate between such inputs and outputs is to transduce the different kinds of input into a common medium, different configurations of which are the different states (e.g., configurations of letters from the same alphabet or patterns of activation of mutually connected neurons). For different configurations of the same medium to exist at the same time and yet be active independently of one another, they must be possessed by different components.[19] The agents in a causal interaction have to be distinct from one another. Furthermore, such components must be able to create the relevant configurations of the medium, distinguish between them, and interact with each other so as to produce the relevant subsequent states and outputs from the previous states and inputs. Thus, functional analysis by internal states requires that the states belong to some of the system's components and constrains the properties of the components.

A system can surely possess different global states at the same time, e.g., a color, a speed, and a temperature. Such global states can affect each other as well as the behavior of the system. For instance, a system's color influences heat absorption, which affects temperature, which in turn affects heat dissipation. But global states can only influence global variables—they cannot mediate between complex configurations of different physical media coming through different input devices and generate specialized configurations of outputs coming from different output devices. Heat absorption and dissipation are not complex configurations of a physical medium—they are global variables themselves. Global variables such as color and temperature can affect other global variables such as heat absorption and dissipation—they cannot transform, say, a specific pattern of retinal stimulation into a specific pattern of muscle contractions. Or, at any rate, no one has ever begun to show that they can.

"functional" states (1967b). But Fodor called them "internal" states (1965; cf. also 1968a; Block and Fodor 1972). Via Putnam and Fodor, Turing's phrase "internal state" has become the established way to talk about functional states. As we have seen, it originally referred to the states of a component (of a computing machine).

[19] In some neural networks, a pattern of neural activation may be taken to store multiple superposed representations. Notice how in this case the different representations cannot operate independently of one another.

Someone might insist that functional analysis in terms of functional states makes no reference (directly or indirectly) to components, and so it need not be a mechanism sketch. The goal of an explanation is to capture in a model how a system behaves; models need not describe components in order to capture how a system behaves; models need not describe components in order to explain.

The problem with this view, as with the analogous conception of task analysis above, is that it confuses explaining with modeling. A resounding lesson of 50 years of sustained discussion of the nature of scientific explanation is that not all phenomenally and predictively adequate models are explanations. We can construct models that predict phenomena on the basis of their correlations (barometers predict but do not explain storms), regular temporal successions (national anthems precede but do not explain kickoffs), and effects (fevers predict but do not explain infections). Furthermore, there is a fundamental distinction between redescribing a phenomenon (even in law-like statements) and explaining the phenomenon. Snell's law predicts how light will bend as it passes from one medium to another, but it does not explain why light bends as it does. We might explain that the light bent because it passed from one medium to another, of course. But that is an etiological explanation of some light-bending events, not a constitutive explanation of why light bends when it passes between different media.

Finally, someone can build predictively adequate models that contain arbitrarily large amounts of superfluous (i.e., nonexplanatory) detail. Explanations are framed by considerations of explanatory relevance. If functional analysis by internal states is watered down to the point that it no longer makes any commitments to the behavior of components, then it is no longer possible to distinguish explanations from merely predictively adequate models and phenomenal descriptions of the system's behavior. Nor does such a modeling strategy tell us how to eliminate irrelevant information from our explanations—a crucial explanatory endeavor. In short, "explanation" of this sort is not worthy of the name.

In conclusion, "internal" states either are not really internal, in which case they constitute a system-level explanandum for a mechanistic explanation, or they are internal in the sense of being states of components. As we have seen, there are two ways to think of components. On one hand are structural components. In this case, functional analysis by internal states is a promissory note on (a sketch of) a mechanistic explanation. The analysis postulates states of some structural components, to be identified by a full-blown mechanistic explanation.[20] On the other hand, components may also be functionally individuated components or black boxes. (For

[20] As we have seen, Fodor says that functional analyses give no information about the mechanism underlying these states (1965, 177), but, at least, they entail that there are components capable of bearing those states and capable of affecting each other so as to generate the relevant changes of states:

> [I]t is sufficient to disconfirm a functional account of the behaviour of an organism to show that its nervous system is incapable of assuming states manifesting the functional characteristics that account requires . . . it is clearly good strategy for the psychologist to construct

instance, the read-write head of Turing machines is a paradigmatic example of a black box.) When components are construed as black boxes, functional analysis by internal states becomes boxology.

8. Boxology

Black boxes are components individuated by the outputs they produce under certain input conditions. In this sense, they are functionally individuated components. In another important strand of the philosophy of psychology, the capacities of a system are said to be functionally explained by appropriately connected black boxes. For example, Fodor distinguishes the functional identification of components from their structural identification (Fodor 1965, 1968a; for a similar distinction, see also Harman 1988, 235).[21] Black boxes are explicitly *internal*—spatially contained within the system.

Proponents of boxology appear to believe that capacities can be satisfactorily explained in terms of black boxes, without identifying the structural components that implement the black boxes. What proponents of boxology fail to notice is that functional and structural properties of components are interdependent: both are necessary, mutually constraining aspects of a mechanistic explanation. On one hand, the functional properties of a black box constrain the range of structural components that can exhibit those functional properties. On the other hand, a set of structural components can only exhibit certain functional properties and not others.

Consider Fodor's example of the camshaft. Internal combustion engines, Fodor reminds us, contain valves that let fuel into the pistons. For fuel to be let in, the valves need to be lifted, and for valves to be lifted, there must be something that lifts the valves. So here is a job description—valve lifting—that can be used to specify what a component of an engine must do for the engine to function. It is a functional description, not a structural one, because it says nothing about the structural properties of the components that fulfill that function, or how they manage to fulfill it. There may be indefinitely many ways to lift valves; as long as something does, it qualifies as a valve lifter. (Hence, multiple realizability.)

What are the components that normally function as valve lifters in internal combustion engines? Camshafts. This is now a structural description, referring to a kind of component individuated by its shape and other structural properties. So there

> such theories in awareness of the best estimates of what the neurological facts are likely to be (Fodor 1965, 176).

Some of what Fodor says in his early works on functional analysis is in line with the present argument and goes against the autonomy assumption that Fodor defended. For simplicity, in the main text I assimilate Fodor's early writings to his later (pro-autonomy) writings. In any event, even Fodor's early writings fall short of pointing out that functional analyses are mechanism sketches.

[21] E.g.: "In functional analysis, one asks about a part of a mechanism what role it plays in the activities characteristic of the mechanism as a whole" (Fodor 1965, 177).

are two independent kinds of description, Fodor concludes, in terms of which the capacities of a system can be explained. Some descriptions are functional and some are structural. Since Fodor maintains that these descriptions are independent, there is a kind of explanation—boxology—that is autonomous from structural descriptions. Functional descriptions belong in boxological models, whereas structural descriptions belong in mechanistic explanations. Or so Fodor maintains. But is it really so?

In the actual "functional analysis" of a mechanism, such as an engine, the functions are specified in terms of physical effects either on a physical medium or on other components, both of which are structurally individuated. The functional term "valve lifter" contains two terms. The first, "valve," refers to a kind of component, whereas the second, "lifter," refers to a capacity. Neither of these is independent of structural considerations. Lifting is a physical activity: for *x* to lift *y*, *x* must exert an appropriate physical force on *y* in the relevant direction. The notion of valve is both functional and structural. In this context, the relevant sense is at least partially structural, for nothing could be a valve in the sense relevant to valve lifting unless it had weight that needs to be lifted for it to act as a valve. As a consequence, the "valve lifter" job description puts three mechanistic constraints on explanation: first, there must be valves (a type of structural component) to be lifted; second, lifting (a type of structurally individuated capacity) must be exerted on the valves; and third, there must be valve lifters (another type of component) to do the lifting. For something to be a valve lifter in the relevant respect, it must be able to exert an appropriate physical force on a component with certain structural characteristics in the relevant direction. This is not to say that only camshafts can act as valve lifters. Multiple realizability stands. But it is to say that all valve lifters suitable to be used in an internal combustion engine share certain structural properties with camshafts.

This point generalizes. There is no such thing as a purely functional analysis of the capacity of an engine to generate motive power. Any attempt to specify the nature of a component purely functionally, in terms of what it is for, runs into the fact that the component's function is to interact with other components to exhibit certain physical capacities, and the specification of the other components and activities is inevitably infected by structural considerations. Here is why. The inputs (flammable fuel, igniting sparks) and the outputs (motive power, exhaust) of an internal combustion engine are concrete physical media that are structurally individuated. Anything that turns structurally individuated inputs into structurally individuated outputs must possess appropriate physical causal powers—powers that turn those inputs into those outputs. (This, by the way, does not erase multiple realizability: there are still many ways to build an internal combustion engine.)

What about boxological models in computational sciences such as psychology and neuroscience? A boxologist committed to autonomy may suggest that while the present assimilation of boxological models to mechanism sketches is viable in most domains, including internal combustion engines, it does not apply to computing systems. In this special domain, our boxologist continues, the inputs and outputs can

be specified independently of their physical implementation, and black boxes need not correspond to concrete components.

In particular, Marr (1982) is often interpreted as arguing that there are three autonomous levels of explanation in cognitive science: a computational level, an algorithmic level, and an implementational level. According to Marr, the computational level describes the computational task, the algorithmic level describes the representations and representational manipulations by which the task is solved, and the implementational level describes the mechanism that carries out the algorithm. Marr's computational and algorithmic levels may be seen as describing black boxes independently of their implementation. Because the functional properties of black boxes are specified in terms of their inputs and outputs (plus the algorithm, perhaps), the black boxes can be specified independently of their physical implementation. Thus, at least in the case of computing systems, boxology does not reduce to mechanistic explanation.

This reply draws an incorrect conclusion from two correct observations. The correct observations are that black boxes may not correspond one-to-one to structural components and that the inputs and outputs (and algorithms) of a computing system can be specified independently of the physical medium in which the inputs and outputs are implemented. As a result, the same computations defined over the same computational vehicles can be implemented in various—mechanical, electromechanical, electronic, etc.—physical media.

But it doesn't follow that computational inputs and outputs put no direct constraints on their physical implementation. In fact, any physical medium that implements a certain computation must possess appropriate physical degrees of freedom that result in the differentiation between the relevant computational vehicles. Furthermore, any component that processes computational vehicles must be able to reliably discriminate between tokens of the relevant types so as to process them correctly. Finally, any components that implement a particular algorithm must exhibit the relevant kinds of operations in the appropriate sequence. The operations in question are different depending on how black boxes map onto structural components.

Consider a staple of functionalist philosophy of psychology: belief and desire boxes. As many have pointed out, belief and desire boxes need not be two separate memory components. But this doesn't entail lack of direct constraints between functional properties and structural components. For the alternative means of implementing belief and desire boxes is to store beliefs and desires in one and the same memory component, while setting up the memory and processor(s) so that they can keep track of which representations are beliefs and which are desires. This may be done by adding an attitude-relative index to the representations or by keeping lists of memory registers. However it is done, the distinction between beliefs and desires constrains the mechanism: the mechanism must distinguish between the two types of representation and process them accordingly (if the organism is to exhibit relevant

behavior). And the mechanism constrains the functional analysis: the representational format and algorithm will vary depending on how the distinction between beliefs and desires is implemented in the mechanism, including whether each type of representation has a dedicated memory component. Thus, even though the decomposition into black boxes may not correspond one-to-one to the decomposition into concrete components, it still constrains the properties of concrete components.

The moral is that computing systems are indeed different from most other functionally organized systems, in that their computational behavior can be mechanistically explained without specifying the physical medium of implementation other than by specifying which degrees of freedom it must possess. But such an explanation is still mechanistic: it specifies the type of vehicle being processed (digital, analog, or what have you) as well as the structural components that do the processing, their organization, and the functions they compute. So computational explanations are mechanistic too (Chapter 7).

What about Marr's computational and algorithmic "levels"? We should not be misled by Marr's terminological choices. His "levels" are not levels of mechanisms because they do not describe component/sub-component relations. The algorithm is not a component of the computation, and the implementation is not a component of the algorithm. The computational and algorithmic levels are mechanism sketches. The computational level is a description of the mechanism's task, possibly including a task analysis, whereas the algorithmic level is a description of the computational vehicles, operations that manipulate the vehicles, and order in which the operations are performed—it's a more fine-grained mechanism sketch than the computational level. All of the above—task, vehicles, and computational operations—constrain the range of components that can be in play as well as the way the components can be organized. They are constrained, in turn, by the available components and their organization. Contrary to the autonomist interpretation of Marr, his levels are just different aspects of the same mechanistic explanation.[22]

So black boxes are placeholders for structural components (with arrows indicating input-output relations between components) or sub-capacities (with arrows indicating causal relations between processes) in a mechanism. Boxology is not distinct from mechanistic explanation. It is a step toward the decomposition of a system into its structural components. It identifies a number of functional components (black boxes), which correspond to one or more structural components. When a boxology is fleshed out into a full-blown mechanistic explanation (at the same level of organization), the ambiguity in the boxology about how many structural components there are is eliminated. What replaces that ambiguity is a precise number of structural

[22] Marr recognized as much at least implicitly, judging from the way that he intermingled considerations from different levels in his theory of visual shape detection. For example, facts about the response properties of retinal ganglion cells influenced his understanding of what the visual system does and the tools that it has to work with in order to do it.

components and a specification of the vehicles they manipulate and operations they perform such that the operations of those structural components fulfill the functions of the black boxes.

Computational and information processing explanations often work by abstracting away from many of the implementing details. But that's how mechanistic explanation generally works; it focuses on the mechanistic level most relevant to explaining a capacity while abstracting away from the mechanistic levels below. Whether a system implements a given computation still depends on its structural features.

9. Integrating Functional Analysis with Multi-level Mechanistic Explanations

Functional analysis of a system's capacities provides a sketch of mechanism. If the functional analysis is just the explanation of the capacities of the system in terms of the system's sub-capacities, this is an articulation of the phenomenon to be mechanistically explained that points at components possessing the sub-capacities. If the functional analysis appeals to internal states, these are states of internal components, which need to be identified by a complete mechanistic explanation. Finally, a functional analysis may appeal to black boxes. But black boxes are placeholders for structural components or capacities thereof, to be identified by a complete mechanistic explanation of the capacities of the system.

Once the structural aspects that are missing from a functional analysis are filled in, functional analysis turns into a more complete mechanistic explanation at one level of organization. By this process, functional analyses can be seamlessly integrated with multi-level mechanistic explanations. And since functional analyses are sketches of mechanisms and, as I will soon argue, computing systems are mechanisms, the proper framework for understanding computational explanation is mechanistic explanation. The next chapter articulates what I take to be the appropriate notion of mechanism, with emphasis on the relevant notion of function. In the following chapter, I will use that notion of mechanism to give an account of *computing* mechanisms.

An important consequence of the argument presented in this chapter is that structural and functional properties are not neatly separable within a mechanism. There is no such thing as a purely functional component, or purely functional property. Structural considerations are ineliminable from a proper understanding of a mechanism. In later chapters, I will argue that this applies to computing systems too. Since the notion of a functional component will play no role in the rest of the book, from now on I will simply use the term 'component' for 'structural component'.

6

The Ontology of Functional Mechanisms

1. Mechanisms with Teleological Functions

Previous chapters led to the preliminary conclusion that computing systems are mechanisms and computational explanation is a type of mechanistic explanation. This chapter articulates the notion of mechanism to be deployed in the next chapter, which contains my mechanistic account of computation.[1]

Most mechanisms do not serve goals or fulfill functions. They do what they do, and there is nothing more to it. For example, the mechanisms of chemical bonding, galaxy formation, weather, or plate tectonics do what they do without fulfilling any function in any teleological sense. Their activities are explained by the organized activities of their components, without attributing any teleological functions to the components.

By contrast, artifacts and biological mechanisms appear to be *for* something; they appear to have *teleological functions*. I call mechanisms that have teleological functions *functional mechanisms* (cf. Garson 2013). Paradigmatic physical computing systems are artifacts—like other artifacts, they appear to be for something. Specifically, computing artifacts are for *computing*. They can do many things and fulfill many functions—including browsing the internet, preparing presentations, and sending emails. Those are all functions that are accomplished by computing. This point generalizes to any putative computing systems, such as nervous systems, that occur within organisms. Their basic teleological function is computing. Or so I will argue in the next chapter. Accordingly, a sound foundation for a mechanistic account of computation that appeals to teleological functions requires an adequate account of functional mechanisms and their teleological functions, and why only artifacts and biological organisms contain mechanisms with teleological functions.

This chapter sketches an ontologically serious account of functional mechanisms. By ontologically serious, I mean that it begins with an independently motivated ontology and, insofar as possible, it grounds a system's functions in objective

[1] This chapter is heavily indebted to Maley and Piccinini, forthcoming, so Corey Maley deserves partial credit for most of what is correct here.

properties of the system or the type of system to which it belongs, as opposed to features of the epistemic or explanatory context of function attribution. Put simply: on the present account, functions are an aspect of what a system *is*, rather than an aspect of what we may or may not say about that system.

I will argue that teleological functions are stable causal contributions towards the goals of organisms. The paradigmatic objective goals of organisms are survival and inclusive fitness, but organisms may have additional goals, including *subjective* goals, contributing to which may be a function too. Truthmakers for claims about teleological functions are non-teleological features of the world.

2. Teleological Functions

It may seem unremarkable that coffeemakers—an artifact designed with a purpose—are for making coffee. To be sure, they do many things: they generate heat, they weigh down what they are placed upon, they reflect light, and they make coffee. But only one of these is the function of a coffeemaker, as indicated by its name. What may seem somewhat more remarkable is that organisms—which are not designed at all—have parts that have functions. A stomach also does many things—it digests, gurgles, and occasionally aches. But a stomach is *for* digesting, which is to say that one of its functions is digestion. What a component of a system is for, as opposed to the other things it does, is its teleological function. When a component fails to perform its teleological function at the appropriate rate in an appropriate situation, it *malfunctions*. From now on, I will simply use the term 'function' for teleological function (unless otherwise noted).

As commonplace as the notion of function is, there is no consensus on the correct account. Any of the existing accounts of function may be used to underwrite a mechanistic account of computation, though in each case some adjustments would be needed. Since I am not fully satisfied with existing accounts, I will propose my own. While I lack enough space to do justice to current accounts, I will briefly indicate why I am unsatisfied with them.

Etiological accounts have been popular among philosophers of biology: roughly, what determines the function of a stomach *here and now* is the reproductive history of ancestral organisms whose stomachs did whatever allowed them to survive (e.g., Millikan 1989; Neander 1991; Griffiths 1993; Godfrey-Smith 1994; and Schwartz 2002). Thus, the stomachs in organisms alive today have the function of digesting, and not gurgling, because it was digesting (and not gurgling) that allowed the ancestors of those organisms to reproduce.[2] According to selectionist accounts—which are similar to etiological accounts—what determines the function of a trait is

[2] Etiological accounts of function come in both strong and weak versions, depending upon whether the function was selected for (strong), or merely contributed to the organism's reproduction (weak) (Buller 1998).

the selection process that causes a trait in a system to be selectively reproduced or retained (Wimsatt 2002; Griffiths 2009; Garson 2011). With respect to artifacts, a sophisticated etiological approach maintains that what determines the function of a coffeemaker *here and now* is the way past coffeemakers were used; that use, in turn, contributed to the reproduction of coffeemakers (Preston 2013). Etiological (and selectionist) accounts of function may be useful in certain contexts, but they are inadequate for my purposes for two reasons, one epistemological and one metaphysical.

The main problem with etiological accounts of functions is epistemological: the causal histories that ground functions on these accounts are often unknown (and in many cases, unknowable), making function attribution difficult or even impossible. While our ignorance does not preclude the ontological reality of functions, functions are very often correctly attributed (or so it seems—quite compellingly) in the absence of any knowledge of a system's causal history. Learning about its causal history can, at best, show that a function has stayed the same or changed over time: learning about that causal history does not lead to changing the *current* attribution of functions to a current system. Thus, etiological accounts do not do justice to the practices of sciences that study the functions of a component or property without regard to evolutionary history (e.g., psychology, neuroscience, functional anatomy, physiology, etc.).

Another problem with etiological accounts, which affects their treatment of both biological and artifact functions, is that they violate an important metaphysical principle concerning causal powers. Consider a real U.S. coin and its perfect, molecule-for-molecule duplicate. One is genuine, the other a counterfeit, and what determines which is which is their respective causal histories. Thus, in one sense there is a real difference between these two entities: they have different kinds of histories. There may even be a way of characterizing this difference as a *physical* difference if we think of objects in a four-dimensionalist way. Nevertheless, the difference between the genuine and the counterfeit cannot result in a difference between the causal powers of the two. We could not build a vending machine that accepted one but not the other, and it would be misguided to demand that a physicist or chemist devise a method for detecting such counterfeits. Similarly, the history of an organism's ancestors cannot contribute to the causal powers of that organism's components or properties. If a tiger were to emerge from the swamp following a lightning strike (à la Davidson's swamp man), its stomach would have the power, and thus the function, of digesting even though it had no ancestors whatsoever.

Etiological theorists may reply that considerations about causal powers are question-begging. For them, attributing a function to a trait is not the same as attributing current causal powers. Rather, it's precisely analogous to calling something a *genuine* U.S. coin—it says something about its origin and history and thus distinguishes it from counterfeit coins (*mutatis mutandis*, from swamp-organisms), regardless of any other similarities in causal powers between genuine and fake coins. But this reply

only highlights that, insofar as the etiological theorist is interested in functions, she is interested in a different notion of function than I am. I am after functions that are grounded in the current causal powers of organisms and their environments, and thus can be discovered by studying those causal powers. In other words, I am after functions that can be shared among organisms, swamp-organisms, and artifacts, regardless of their exact reproductive or selection histories.

To be sure, there are accounts that do not rely on causal histories, but I find them inadequate for my purposes. Causal role accounts (such as Cummins 1975 and Craver 2001) reject teleological functions and instead consider functions as causal contributions to an activity of a complex containing system. As a consequence, according to causal role accounts, everything or almost everything (of sufficient complexity) ends up having functions, which clashes with the fact that only organisms and artifacts have functions in the sense that I am interested in, that is, the sense in which things can malfunction.

One way of fixing this weakness is to appeal to explanatory interests and perspectives (Hardcastle 1999; Craver 2012). From the perspective of the survival of organisms, the function of the heart is to pump blood. From the perspective of diagnosing heart conditions, the function of the heart is to make thump-thump noises. From the perspective of the mass of the organism (to speak loosely), the function of the heart is to contribute a certain amount of mass. And so on. This perspectivalism makes functions observer-dependent and hence subjective. But functions seem perfectly objective.

Contrary to perspectivalism, the function of the heart is to pump blood, not to make noises or to possess a certain mass, and it has this function even if there is no one around to observe it. Some traits do have multiple functions, but not in virtue of multiple perspectives. From the very same perspective—the perspective of identifying the functions of the medulla oblongata—the medulla oblongata has many functions, including regulating breathing, circulation, and blood pressure, initiating the gag reflex, and initiating vomiting. The reason we are sometimes interested in the noises the heart makes—the reason we listen to the heart's noises at all—is *not* that the noises are another function of the heart in addition to pumping blood; rather, the noises are useful in diagnosing how well the heart performs its function: pumping blood. Consider what would happen if a cardiologist discovered that a completely silent heart nevertheless pumps blood perfectly; the cardiologist would not declare that a new kind of heart malfunction has been discovered; rather, she would try to figure out how this heart can perform its function silently. The converse—the perfectly sounding, thump-thumping heart that pumped blood poorly—would be considered malfunctioning.

In summary, perspectivalism does not do justice to the perspectives we actually take in the biological sciences. If we could identify non-teleological truthmakers for teleological claims, we would avoid perspectivalism and deem functions real without deeming them mysterious. That is my project.

Accounts that ground artifact functions in terms of the intentions of designers and users (Houkes and Vermaas 2010) face the problem that intentions are neither necessary nor sufficient for artifacts to have functions. That intentions are insufficient to confer genuine functions is illustrated by artifacts such as amulets and talismans, which lack genuine functions even though their designers and users have all the right intentions and plans for their proper use. That intentions are unnecessary to confer functions is illustrated especially well by artifacts created by non-human animals such as spiders and termites.

Accounts that identify functions with propensities (Bigelow and Pargetter 1987) cannot account for malfunctioning items, which have a function yet lack the propensity to perform it. Accounts based on special features of living systems (e.g., self-maintenance, self-preservation, reproduction; cf. Albert, Munson, and Resnik 1988; Christensen and Bickhard 2002; McLaughlin 2001; Mossio, Saborido, and Moreno 2009; Schlosser 1998; Schroeder 2004; Wouters 2007) are on the right track and I will retain what I take to be right about them.

Goal-contribution accounts (such as Nagel 1977; Adams 1979; Boorse 2002) are right that functions contribute to a system's goal(s). That is the core idea behind my account as well. But traditional goal-contribution accounts maintain one or more of the following: that a system has functions only if it is goal-directed, that a system is goal-directed only if it is guided via feedback-control, or that a system is goal-directed only if it represents its goals. The problem is that plenty of things—e.g., doormats—have functions without being goal-directed, without being guided via feedback-control, or without representing goals. Thus we need a more inclusive account of goals and the relation between goals and functions than those offered by traditional goal-contribution accounts.

Unlike the account I am about to propose, previous accounts—even when they are on the right track—often suffer from one or more of the following: a lack of a plausible ontology, a lack of coordination with a multi-level mechanistic framework, or a lack of a unified treatment of both organismic functions and artifact functions. I have already mentioned the benefits of an ontologically serious account, and the utility of including a multi-level mechanistic framework is obvious for my purpose in this chapter. What about unifying organismic and artifact functions? While some have argued against this possibility (Godfrey-Smith 1993; Lewens 2004), these arguments have primarily been concerned with etiological accounts.

A unified account provides a desirable foundation for taking seriously the analogies between the functions of biological traits and the functions of artifacts that are part and parcel of many scientific explanations. Most relevantly, computers and brains are often compared to one another. For example, computers are said to be intelligent to some degree, and brains are said to perform computations. If biological traits and artifacts perform functions in different senses, the analogies between them and their functions become correspondingly opaque: saying that brains perform computing functions means something different from saying that artificial

computers perform computing functions. If, on the contrary, biological traits and artifacts perform functions in the same sense, then the analogies between them that are ubiquitous in the special sciences become correspondingly clear and well grounded. A unified account is also more parsimonious and elegant, so that is what I offer here.

3. Ontological Foundations

I assume an ontology of particulars (entities) and their properties understood as causal powers.[3] I remain neutral on whether properties are universals or modes (tropes). A similar account could be formulated in terms of an ontology of properties alone, with entities being bundles thereof, or in terms of processes.

Activities are manifestations of properties (powers). Some have objected that we can only observe activities and not powers and hence activities must be fundamental (e.g. Machamer 2004). I set this concern aside on the grounds that activities may be evidence of powers.

When many entities are organized together in various ways, they form (constitute) more complex entities. Such complex entities have their own properties (causal powers), which are constituted by the way the causal powers of the constituting entities are organized and perhaps modified by the way such entities and their causal powers are organized. For instance, when atoms chemically bond to one another, they form molecules with properties constituted by those of the individual atoms, including properties of individual atoms that have changed because they are so bonded. The subsystems that constitute complex entities and make stable causal contributions to their behavior are what I call *mechanisms* (cf. Craver 2007 and Machamer, Darden, and Craver 2000, among many others).

The causal powers of mechanisms have special subsets; they are special because they are the causal powers whose manifestation are their most specific (peculiar, characteristic) interactions with other relevant entities. These are the most characteristic "higher-level" properties of complex entities and their mechanisms (Piccinini and Maley 2014). Since higher level properties are subsets of the causal powers of the lower level entities that constitute the higher level entities, they are no "addition of being" (in the sense of Armstrong 2010) over and above the lower-level properties; therefore, they do not run into problems of ontological redundancy such as causal exclusion (Kim 2005). Thus we establish a series of non-redundant levels of entities and properties, each level constituted by lower-level entities and properties.

Organisms are a special class of complex entities. What is special about them is some of their general properties. First, they are organized in ways such that

[3] A special version of this ontology is articulated by Heil (2003, 2012) and Martin (2007). Heil and Martin also equate causal powers with qualities. Since qualities play no explicit role in this book, I prefer to stay neutral on the relationship between causal powers and qualities.

individual organisms preserve themselves and their organization for significant stretches of time. This is typically accomplished by collecting and expending energy in order to maintain a set of states so that the organism is resistant to various types of perturbation. For example, mammals expend energy in order to maintain a certain body temperature; without these homeostatic mechanisms, fluctuations in temperature outside a very narrow range would disrupt activities necessary for mammalian life. I call the characteristic manifestation of this first property *survival*. Many organisms are also organized in ways that allow them to make other organisms similar to themselves by organizing less complex entities; that is, to reproduce. In some cases, although individual organisms are not organized to reproduce, they *are* organized to work toward the preservation of their kin. Honeybee workers, for example, are infertile, but still contribute to the survival and reproduction of the other members of their hive. I call the characteristic manifestation of either of these latter properties *inclusive fitness*.[4]

Survival and inclusive fitness as I have characterized them are necessary for the continued existence of organisms. Although individual organisms can last for a while without reproducing and even without having the ability to reproduce, these individuals will eventually die. If no individuals reproduced, there would soon be no more organisms. Even quasi-immortal organisms (i.e., organisms that have an indefinitely long lifespan) will eventually be eaten by a predator, die of a disease, suffer a fatal accident, or succumb to changed environmental conditions. So, barring truly immortal, god-like creatures impervious to damage, which are not organisms in the present sense anyway, survival and inclusive fitness are necessary for organisms to exist.

That these two properties are essential for the existence of biological organisms is obviously a special feature of them both. Another special feature is that the manifestation of these properties requires organisms to expend energy. I call the state toward which such a special property manifestation is directed, and which requires work on the part of the organism via particular mechanisms, an *objective goal* of the organism.

This is reminiscent of the cybernetic accounts of goal-directedness as control over perturbations (Rosenblueth et al. 1943; Sommerhoff 1950). While I am sympathetic to cybernetic accounts of goal-directedness and rely on it in my appeal to goals, I depart from previous goal-contribution accounts of functions (Nagel 1977; Adams 1979; Boorse 2002) because I do not maintain that being goal-directed is necessary or sufficient to have functions. Instead, I ground functions directly in the special organismic goals of survival and inclusive fitness.

Note that mine is a technical sense of "goal", which does not entail any kind of goal representation, mental or otherwise. Furthermore, there is no requirement that goals

[4] Some organisms do not pursue their inclusive fitness at all. They are the exception that proves the rule: if their parents had not pursued inclusive fitness . . .

be always *achieved*: all that is required is that these goals are a sort of state toward which organisms must expend energy in order for organisms to exist. This notion of goal underwrites the primary notion of teleological function that is used in some sciences. It also seems to underlie much of our commonsense understanding of functions.[5]

It may be objected that there are systems and behaviors that "survive" but lack goals in the relevant sense. For instance, a gas leak may poison any plumber who tries to fix it, thereby preserving itself. Does it follow that, on the present account, the gas leak has the objective goal of surviving? Or consider a crack addict who steals in order to buy crack, thereby preserving the addiction, and sells crack to others, thereby "reproducing" crack addiction. Does it follow that, on the present account, a crack addiction has the objective goal of maintaining and reproducing itself?

These two putative counterexamples are quite different. As to the gas leak, it does not reproduce and does not pursue its inclusive fitness. (It also does not extract energy from its environment in order to do work that preserves its internal states, etc.) Therefore, it is not an organism in the relevant sense. If there were artificial systems that did pursue survival and inclusive fitness in the relevant sense (i.e., self-reproducing machines), they would be organisms in the relevant sense and would have objective goals (more on this below). As to crack addiction, its preservation and reproduction are not objective goals because they are detrimental to the survival of the addicts and yet they depend on the objective goal of survival in a way in which survival does not depend on them. That is, crack addiction requires organismic survival in order to persist and reproduce; but organismic survival itself does not require crack addiction (quite the contrary). In a rather loose sense, someone *might* think of crack addiction as a kind of parasite, and thus as a special kind of organism that may have objective goals. But this sense is so loose as to be unhelpful. To name just one immediate difficulty, it is quite problematic to reify the complex pattern of behaviors, dispositions, desires, etc. that constitute an addiction as an entity separate from the addict. A better way of addressing cases like crack addiction is to treat them as a *subjective* goal of some organisms (more on this below).

Another objection runs as follows: having mass is necessary for survival, whereas survival is not necessary for having mass; by parity of reasoning with the crack addiction case, it seems to follow that survival is not an objective goal, whereas having mass is. But having mass is necessary for survival only in the sense that massless objects cannot organize themselves into organisms. It takes a special organization for

[5] Two interesting questions are whether there are objective goals other than the survival and inclusive fitness of organisms and whether entities other than organisms and their artifacts have functions. Additional objective goals may include the survival of the group, or of the entire species, or of the entire biosphere. Additional entities with functions may include genes, groups, societies, corporations, ecosystems, etc. I leave the exploration of these possibilities to another occasion. For now, I limit myself to the paradigmatic cases of teleological functions of organisms and their artifacts based on their objective goals of survival and inclusive fitness.

massive objects to turn themselves into organisms. When they organize themselves into organisms, such suitably organized massive objects either survive or perish. Thus, there is a big difference between having mass and surviving. Only the latter is a defining characteristic of organisms, which distinguishes them from other systems. Having mass is something organisms share with many other systems, which are not goal-directed towards survival. But notice that organisms may need to maintain their mass within a certain range in order to stay alive. If so, then maintaining their mass within that range is a subsidiary goal of organisms, which serves the overarching goal of survival.

4. Teleological Functions as Contributions to Objective Goals of Organisms

We now have the ingredients for an account of teleological functions in organisms.

> A *teleological function in an organism* is a stable contribution by a trait (or component, activity, property) of organisms belonging to a biological population[6] to an objective goal of those organisms.

By 'contribution', I mean a positive contribution—an effect that increases the probability of reaching a goal. By 'stable' contribution, I mean a contribution that occurs regularly enough as opposed to by accident; the boundary between regularity and accident is vague and I leave it to empirical scientists to sort it out in specific cases. The contexts and frequencies required for traits to have functions, for their functions to be performed at an appropriate rate in appropriate situations, and for traits to malfunction, are made precise elsewhere (Garson and Piccinini 2014).

Construed generally, a trait's function (and sometimes the successful performance of that function) depends on some combination of the organism and its environment (which may include other organisms: kin, conspecifics, or unrelated organisms, specified or unspecified). In other words, the truthmakers for attributions of functions to an organism's trait are facts about the organism and its environment. Different types of function depend on factors outside the organism to different extents. It is worth considering some real-world examples in order to get a feel for the range of facts upon which functions can depend.

Some functions, such as the blood-pumping of tiger hearts that pump at an appropriate rate, depend primarily on the individual organism: blood-pumping contributes to the survival of a single tiger, independent of the existence of any other organisms. But this function also depends on the environment: the tiger must be in the right kind of atmosphere with the right pressure, located in the right kind of gravitational field, etc.

[6] More precisely, to a reference class within a population. Reference classes are classes or organisms divided by sex and developmental stage (Boorse 2002).

If, in an appropriate situation, a previously well-functioning tiger's heart were to stop pumping blood, then the tiger would die; we can safely say its heart has malfunctioned (or is malfunctioning). A less severe malfunction would result if the tiger's heart were to pump at an inappropriate rate. Determining the appropriate situations for a trait's functioning and the appropriate rate at which a trait ought to function in an appropriate situation may require comparing the rates of functioning of different trait tokens of the same type in different organisms in the same population. The trait tokens that provide a sufficient contribution to the objective goals of an organism are the well-functioning ones, the others are malfunctioning. Thus, whether a trait has a function, and thus a malfunction, may depend on the way other traits of the same type function in other organisms.[7] In addition, the environment is important here because what may be a malfunction in one environment might not be in some other environment. An enlarged heart on Earth would result in a malfunction; but in an environment with, say, higher atmospheric pressure, a non-enlarged heart might be a malfunction.

Nanay (2010) objects that comparing a trait token to other traits of the same type in order to determine its function requires a function-independent way of individuating trait types, and he argues that there is no function-independent way of individuating types. I believe that there are function-independent ways of individuating types. But I won't defend this thesis here because, *pace* Nanay, there is no need for a (purely) function-independent way of individuating traits.

To see why, consider a biological population. Begin with a function—that is, begin with a certain stable contribution to the pursuit of an objective goal of the organisms in that population. Then, find the traits that perform that function. Find the well-functioning trait tokens, each of which performs the same function. The trait tokens can be typed together *because* they perform the same function (as well as by their morphological and homological properties). Now that we have established a type, we can type other (less-than-well-functioning) traits: they belong in the same type insofar as they share a combination of the following: less-than-appropriate performance of their function, morphological properties, and homological properties. By typing well-functioning trait tokens first and then typing less-than-well-functioning tokens later, we need not rely on a function-independent way of individuating trait types.

This way of individuating functional types may well recapitulate the way functions are discovered and attributed empirically, but that is not my point. My point is that there is an asymmetric ontological dependence between the functions of

[7] What if an organism is the last survivor of its species? We can include organisms that lived in an organism's past as part of the truthmaker for attributions of functions to its traits. Nothing in my account requires that all organisms relevant to function attribution live at the same time. This feature does not turn my account into an etiological account, however, because my account does not make the reproductive history of a trait (let alone selection for certain effects of a trait) constitutive of its function.

malfunctioning tokens and the functions of well-functioning tokens. The functions of malfunctioning tokens are grounded in part in the functions of well-functioning tokens (which in turn are constituted by some of their causal powers), but not vice versa. In other words, the truthmakers for functional attributions to malfunctioning tokens include the causal powers of well-functioning tokens, but not vice versa.

Some functions depend on the environment because they depend on other species. Consider the eyespots of the Polyphemus moth, which have the function of distracting would-be predators (a contribution to the moth's survival). This function depends on the existence of would-be predators disposed to be distracted by these eyespots. Another example is the giant sphinx moth, which has a proboscis long enough to drink nectar from the ghost orchid. The function of the proboscis is to draw in nectar from this particular orchid species. In both cases, these traits would have no function without the environment: the eyespots would not have the function of distracting would-be predators if there were no would-be predators, and the proboscis would not have the function of drawing up ghost orchid nectar if there were no ghost orchids. If the environment were different, the traits might have no function, or might even acquire new functions.[8]

Finally, some functions—particularly those that contribute to inclusive fitness—depend on the existence of kin, and often other species as well. The female kangaroo's pouch, for example, has the function of protecting the young joey as it develops and nurses. If there were no joeys, the pouch would have no function. The stingers of honeybee workers have the function of stinging, which deters would-be hive intruders. This is certainly not a contribution to the honeybee's survival—using a stinger usually results in the death of the individual honeybee—but it *is* a contribution to the survival of its kin, and hence to its inclusive fitness. Thus, the stinger's function depends on the existence of both the honeybee's kin and would-be hive intruders.

As many have pointed out (e.g., Craver 2007), organisms contain mechanisms nested within mechanisms: mechanisms have components, which are themselves mechanisms, which themselves have components, etc. Mechanisms and their components have functions, and the functions of components contribute to the functions of their containing mechanisms. Thus, a contribution to an objective goal may be made by the organism itself (via a behavior), or by one its components, or by one of its components' components, and so on.

Examples of such multi-level mechanisms abound in neuroscience. One such example is the way various species of noctuid moth avoid bat predation by way of their tympanic organ (Roeder 1998 describes the discovery of this organ's function

[8] This accords well with how biologists describe the evolutionary beginnings of vestigial structures: the environment in which a trait once had a function changes, leaving the trait with no function (often resulting in non-adaptive evolutionary changes to the trait). The vestigial hind legs of the blue whale have no function, but these hind legs presumably did for the whale's land-dwelling ancestors.

and the mechanism responsible). Roughly, this organ's function is detecting approaching bats; when it does, it sends signals to the moth's wings, initiating evasive maneuvers, often allowing the moth to avoid the oncoming predator (turning away from a bat that is some distance away, and diving or flying erratically when a bat is very close). The tympanic organ does this by responding differentially to the intensity of a bat's ultrasonic screeches (which the bat uses for echolocation). When we look at the components of this organ, we can identify mechanisms—and their functions—within these components. For example, the so-called A-neurons have the function of generating action potentials in response to ultrasonic sounds hitting the outer tympanic membrane. These action potentials are part of the tympanic organ's activity, which in turn drives the moth's response to the predator. We can then look at the mechanisms of components of these neurons, such as ion channels, that have the function of allowing ions to flow into or out of the neuron. Each of these components has a functional mechanism that contributes, either directly (e.g., the initiation of evasive maneuvers) or indirectly (e.g. allowing the flow of ions) to the objective goal of survival. All of these functions are more than a matter of mere explanatory interest, or part of an analysis of one system or other: these functions are contributions to the survival of the organism.

There is a related notion of functionality in organisms that the present framework accounts for. Sometimes physiologists distinguish between functional and non-functional conditions based on whether a condition is found *in vivo* (in the living organism) or only *in vitro* (in laboratory preparations). Presumably, the underlying assumption is that unless a condition is found in the living organism, it is unlikely that anything that happens under that condition makes a contribution to the survival (let alone inclusive fitness) of the organism. What happens under conditions that are "non-functional" in this sense may be by-products of the way organisms are built for their ordinary conditions.

An important feature of the present account is that it extends nicely to artifacts, particularly those of non-human animals. Some behaviors modify environments in stable ways that contribute to either survival or inclusive fitness. The stable ways in which environments are modified (e.g., burrows, nests, spider webs, tools) are artifacts in an important, general sense that includes more than ordinary human artifacts. Needless to say, artifacts in this general sense have teleological functions too. And those teleological functions are typically contributions to the survival or inclusive fitness of the organisms that create the artifacts. Thus, to a first approximation, the teleological functions of artifacts are contributions to the survival or inclusive fitness of organisms.

> A *teleological function of an artifact* is a stable contribution by an artifact to an objective goal of the organism(s) that created the artifact.

In many cases, the functions of artifacts are individuated in terms of artifact types and the stable contributions of the well-functioning members of the type to the goal

of a population of organisms. (In this context, 'well-functioning' is shorthand for providing a stable contribution to an objective goal of an organism.) This is the same way that the functions of biological traits are individuated, which allows for a parallel way to ground functions and malfunctions in artifacts. Thus, malfunctioning artifacts are those that belong to a type with a certain function but are unable to perform that function at the appropriate rate in appropriate situations.

But artifact tokens can acquire functions on their own too, regardless of which type they belong to or whether there are other members of their type. This includes unique artifacts, of which there is only one copy, and artifacts that are put to uses for which they were not made—for instance, a screwdriver that is used as a knife. The present account of artifact functions is general enough to accommodate not only artifact tokens that belong to types of which there are many members, but also unique artifacts and artifacts that are put to novel uses. In the case of unique artifacts and artifacts that are put to novel use, their stable contribution to an objective goal of an organism is their function. The stability and frequency of these contributions determine whether an artifact is simply *functioning as* something else or it acquires a new function: screwdrivers are infrequently (and not very successfully) used for cutting, so they may not *have* the function of cutting, although they can function *as* knives. Alternatively, screwdrivers (flathead screwdrivers, anyway) are frequently (and successfully) used to pry lids off of metal cans, so prying is a function of screwdrivers. If an artifact fails to perform its function at the appropriate rate in appropriate situations, it malfunctions.

We thus obtain a unified account of teleological function for both organisms and artifacts.

5. Some Complications

I have characterized survival and inclusive fitness as the objective goals of organisms, both of which underwrite the present account of teleological function: traits or parts of organisms have functions insofar as they contribute to at least one of these goals. There can be a tension between contributing to survival and contributing to inclusive fitness. They are mutually supportive to some extent, because survival is necessary for reproduction, which is necessary for inclusive fitness, which in turn is necessary for the survival of the species. But survival and inclusive fitness may also be in competition. Most obviously, one organism may sacrifice itself for the sake of the survival of its kin. More generally, traits that lengthen the life of an organism may lower its chances of reproducing, and traits that increase the probability of reproducing (or even attempting to reproduce) may put the organism at risk of death.

A related tension appears with respect to reproductive behavior. For sexual organisms, finding a mate, or even better, a good mate, is necessary to increase inclusive fitness, but it requires considerable energy (and in some cases, the growth of energetically expensive appendages such as the heavy antlers that may have

contributed to the extinction of the Irish Elk (Gould 1974; Moen, Pastor, and Cohen 1999)). For nurturing organisms, increasing the chance that offspring survive and are viable is also necessary for inclusive fitness (and for survival too, in the case of organisms prone to deep depression when their offspring dies) but it also requires expenditures of energy, and in some cases it may require the ultimate sacrifice.

Thus, a trait or part of an organism may very well have a function with respect to one goal, but at the same time, that trait or part may *not* contribute to another goal, and may even be detrimental to it. The bee's stinger contributes to inclusive fitness by repelling threats to the hive, thus grounding the function of the stinger. But bees that use their stinger may lose part of their abdomen, resulting in the death of that bee; thus, stinging is detrimental to the survival of the bee. Nevertheless, the stinger is functional relative to the goal of inclusive fitness, and thus it has a function *simpliciter*, even though it is a hindrance to the other goal of survival.

An additional area of conflict could result from putative artifacts that have the ability to survive and reproduce, such as very advanced (though not yet extant) robots. These are artifacts because they would (originally) be constructed by organisms, but they also have their own objective goals because they are able to survive and reproduce. In an important sense, these artifacts are organisms too, and surely their offspring are organisms. The objective goals of such artifacts might be in conflict with the objective goals of their creators—as many science fiction writers have explored. So an activity or trait may be functional relative to the objective goals of such an artifact but it may be dysfunctional relative to the objective goals of its creator, and vice versa. Either way, a function remains a stable contribution to an objective goal of an organism (either the artifact-organism or its creator).

One more difficulty regards artifacts that were designed to fulfill a function but in practice do not, or cannot, do so. Perhaps there is a flaw in the design, there are unforeseen difficulties in building the artifact, or unforeseen defeating conditions prevent the artifact from working. Or perhaps there is simply a radically mistaken belief that such artifacts can perform a certain function when in fact they cannot. For instance, amulets and talismans are designed, built, and used because they are believed to protect from evil and bring good luck, respectively, but such beliefs are false (cf. Preston 2013 on "phantom" functions). Insofar as these artifacts do not contribute to the relevant goal of an organism, the present account deems them functionless. Yet, intuitively, one might think these artifacts simply have the function their designers, makers, or users intend them to have. A bit of reflection reveals that this is not the case.

Paradigmatically, if something has a function of *X*-ing, then it should be able to *X* (or do *X*, or function *as* an *X*-er). If something has making coffee as one of its functions, then it should be able to make coffee—it should be able to function *as* a coffeemaker. The converse does not hold: a hat can function *as* kindling, but a hat does not have the function of helping to start fires. Thus, a machine merely *intended* to make coffee, but incapable of doing so, does not have the function of making

coffee, and cannot function as a coffee maker, even if its maker intended otherwise. By the same token, amulets and talismans do not have the function of, respectively, protecting from evil and bringing good luck, regardless of what their designers, makers, and users believe. The easiest way of seeing this is that amulets, talismans, and other artifacts that cannot fulfill their putative function cannot *mal*function, for the simple reason that, by assumption, they cannot fulfill their putative function to begin with. Insofar as sometimes we might *say* they have functions, they have functions only by courtesy—their functions are entirely in the eyes of the beholder.

A few caveats: consider an object that once was able to make coffee, but no longer can. It is simply malfunctioning, so it still has the function of making coffee. This is not because of its history but because it belongs to a type some of whose tokens (including its former self) can make coffee. When it comes to something having the function of *X*-ing, there is no metaphysical difference due to whether something was once capable of doing *X*. Consider two physically indistinguishable devices, both of which were designed to make coffee but are currently incapable of doing so because of some defect *D*. Now suppose that one of these devices *acquired D* due to damage, whereas the other device always had *D* due to a manufacturing defect, such that the former was *once* capable of making coffee, whereas the latter was never capable of doing so. Both devices are malfunctioning coffee makers, because both belong to a type of devices some of whose tokens can make coffee.

Another case is a device that never actually makes (and never did make) coffee—perhaps it is on display at the MOMA—but it *could* make coffee and belongs to a type whose function is making coffee. Then it is a bona fide coffee maker—it has the function of making coffee. What about a putative coffee maker that is currently defective, and has never made coffee? If it is a token of a type some of whose tokens *can* make coffee, then it has coffee-making as one of its functions just in virtue of belonging to such a type. It is a genuine coffee maker, though a malfunctioning one. But if it is so radically defective that it is not a token of any coffee-making type, then it does not have the function of making coffee at all. It is not a genuine coffee maker.[9]

Without these caveats, we could not make sense of malfunctions: even though neither the defective coffee maker nor my hat can make coffee (i.e., neither can now function as a coffeemaker), my hat is not a malfunctioning coffee maker. And neither is a pair of sticks tied together, even if a designer says so. Consider an inventor who announces she has built a rocket designed to take her to Mars, yet all she shows us is a useless mess of loosely connected components. We would not say that this is a rocket at all, let alone that it has the function of taking anyone to Mars.

Finally, what to make of physical duplicates, such as the swamp tiger mentioned earlier? Clearly, this duplicate will have some functions: its heart will have the

[9] It is misleading to call such radical departures "defects": correcting a defect ought to result in a functional artifact, whereas a departure so dramatic that it renders a token no longer of the relevant type is such that the token seems incapable of being fixed. This observation reinforces the point in the main text.

function of pumping blood, and its stomach will have the function of digesting. Insofar as a miraculously-created physical duplicate of a tiger is alive, these organs contribute to the objective goal of surviving. Contributions to reproduction work the same way.

Contributions to inclusive fitness that go beyond reproduction are a difficult case. Consider a swamp worker bee. It may seem that its stinger has a function, just as a non-swamp bee's stinger has a function. But the non-swamp bee's stinger contributes to inclusive fitness, which means that the stinger's function requires the existence not only of the bee with its intrinsic physical properties, but also some of that bee's kin or conspecifics. But the swamp bee seems to have no kin, just as it has no evolutionary history. Does this mean that its stinger has no function?

The answer depends on how kinship is defined. If kinship is defined by having a certain ancestral relation, then the swamp bee has no kin, and its stinger has no function. If kinship is defined by a sufficient degree of physical similarity with certain other bees, however, then both the original bee and its physical duplicate have the same kin. Just as the stinger of a regular bee has a function *even if* it never in fact contributes to inclusive fitness, but *would* in the right circumstances, so too does the stinger of a duplicate bee have that function: it too would, in the right circumstances, contribute to inclusive fitness. Of course, if there were swamp organisms that were created out of a random lightning strike, but were not physical duplicates of *any* extant organisms, then the present account says nothing about the functions of those swamp organisms beyond those that contribute to survival (and reproduction, assuming that such a swamp organism could reproduce in a meaningful way). But that is an acceptable result: my concern is with physical duplicates of actual organisms, with actual functional parts.

There is one more objection I will consider: if you fail to pursue your survival or inclusive fitness, are you failing to pursue one of your objective goals, and is this morally objectionable? If the present account entailed that people *ought* to pursue their survival or inclusive fitness, there would be something wrong with the present account. But the present account does not entail that. I am entirely neutral on ethical matters—what anyone *ought* to do. For all I say, organisms have no obligation to survive, let alone pursue inclusive fitness. It does not follow from the present account that any particular organism ought to pursue inclusive fitness, or even that *some* organisms ought to. All I am pointing out is that were *all* organisms to stop pursuing survival *and* inclusive fitness, eventually all organisms would cease to exist. That is enough to differentiate organisms from other physical systems and to ground the attribution of functions to organismic traits and artifacts.

6. A Generalized Account of Teleological Functions

One problem remains before I can offer a completely unified account of teleological functions: sentience and sapience. Due to sentience and sapience, the goals that we, as

agents, pursue can align with, oppose, or be completely orthogonal to our objective goals.

Comfort, lack of pain, and pleasure may be seen as goals for sentient organisms, even though their pursuit may be in competition with their objective goals of survival and inclusive fitness. Sapience, with or without sentience, may give rise to goals of its own (e.g., knowledge), which may also be in competition with other goals. One consequence of sentience and sapience is ethics or morality, which may give rise to further goals such as the survival of non-kin.

I call goals due to sentience and sapience *subjective goals*. I don't need to list all subjective goals or answer the question of whether contributions to these goals are teleological functions: it is enough that these subjective goals are clearly distinct from objective goals, and nothing on the present account hinges on whether contributions to subjective goals are functions. If we choose to say that contributions to subjective goals *are* functions, the present account can be extended to cover them: just as in the case of objective goals, a trait or activity that contributes to a subjective goal in a stable way has that stable contribution as a function. It seems reasonable to allow for this extension of the present account, so I will adopt it.

The general difficulty regarding subjective goals extends to some of the artifacts of sentient beings. Tobacco smoking is detrimental to survival and inclusive fitness. But the function of a pipe is to hold tobacco so that it can be smoked. What grounds the functions of these artifacts has to do with the subjective goals of these artifacts' users. We thus obtain a generalized account of functions in both organisms and artifacts.

> A *teleological function* (generalized) is a stable contribution to a goal (either objective or subjective) of organisms by either a trait or an artifact of the organisms.

There is a whole research program of sorting out more precisely the relationship between objective and subjective goals and the functions they give rise to. I cannot take it up here.

I do note, however, that in our everyday life and its concerns, including the application of our sciences (such as biology and medicine), all of the above issues are mixed together. This makes matters complicated. It may make it difficult to distinguish whether, for example, when we say that a coffee maker is for making coffee, we are attributing a function based on the coffee maker's contribution to our objective goals, our subjective goals, or both. Similarly, when we say that people should exercise, are we making an ethical judgment, a judgment about their subjective well-being, or a judgment about how to increase their chances of survival? We might be making all these judgments at once. None of this eliminates the objective fact that keeping our body efficient (e.g., through exercise and diet) increases the chances of long-term survival. That is the kind of fact on which teleological functions in the most basic sense are grounded, according to the present account.

This concludes my account of functional mechanisms—mechanisms that have teleological functions. Teleological functions are stable contributions toward the goals of organisms. This account applies equally well to the components of organisms as well as to artifacts and their components. This account does justice to the scientific practices pertaining to teleological functions by grounding functions in current causal powers without appealing to etiology. This account provides the foundations for a mechanistic account of computation that relies on the teleological functions of computing systems. That's what we now turn to.

7

The Mechanistic Account

1. A Mechanistic, Non-semantic Account

I laid out desiderata for an account of concrete computation (Chapter 1), argued that existing accounts do not adequately satisfy them (Chapters 2–4), argued that computation should be explicated mechanistically (Chapter 5), and introduced the relevant notion of functional mechanism (Chapter 6). I am now in a position to articulate my account of what it is for a physical system to be a computing system—a system that performs concrete computations.

The primary aim of the mechanistic account is doing justice to the practices of computer scientists and engineers as well as cognitive scientists; the secondary aim is to help guide the practices of other computational scientists who wish to employ the notion of computation employed in computer science. I will argue that, unlike mapping accounts (Chapter 2) and semantic accounts (Chapter 3), the mechanistic account satisfies the desiderata laid out in Chapter 1.

The present account is *mechanistic* because it deems computing systems a kind of functional mechanism—mechanism with teleological functions. Computational explanation—the explanation of a mechanism's capacities in terms of the computations it performs—is a species of mechanistic explanation.

The mechanistic account is *non-semantic* in the following sense. Unlike semantic accounts, the mechanistic account doesn't require that computational vehicles be representations and individuates computing systems and the functions they compute without appealing to their semantic properties. In other words, the mechanistic account keeps the question whether something is a computing system and what it computes separate from the question whether something has semantic content and what it represents. Of course, many (though not all) computational vehicles do have semantic properties, and such semantic properties *can* be used to individuate computing systems and the functions they compute. The functions computed by physical systems that operate over representations can be individuated either semantically or non-semantically; the mechanistic account provides non-semantic individuation conditions. Contrary to what many suppose (Chapter 3), computation does *not* presuppose representation.

The mechanistic account flows naturally from these theses. Computing systems, such as calculators and computers, consist of component parts (processors, memory

units, input devices, and output devices), their functions, and their organization. Those components also consist of component parts (e.g., registers and circuits), their functions, and their organization. Those, in turn, consist of primitive computing components (paradigmatically, logic gates), their functions, and their organization. Primitive computing components can be further analyzed mechanistically but not computationally; therefore, their analysis does not illuminate the notion of computing systems.

In this chapter, I articulate the mechanistic account and argue that it satisfies the desiderata. Satisfying these desiderata allows the mechanistic account to adequately explicate both our ordinary language about computing mechanisms and the language and practices of computer scientists and engineers. In later chapters, I will explicate some important classes of computing systems and apply the mechanistic account to some outstanding issues in the philosophy of computation.

2. Computing Mechanisms

2.1 Functional mechanisms

The central idea is to explicate computing systems as a kind of functional mechanism. A system X is a *functional mechanism* just in case it consists of a set of spatiotemporal components, the properties (causal powers) that contribute to the system's teleological functions, and their organization, such that X possesses its capacities because of how X's components and their properties are organized (Chapter 6). Components that do not perform their functions are said to malfunction or be defective. To identify the components, properties, and organization of a system, I defer to the relevant community of scientists.

This notion of functional mechanism applies to ordinary computers and other computing systems in a way that matches the language and practices of computer scientists and engineers.[1] Computing systems, including computers, are mechanisms whose function is computing. Like other mechanisms, computing mechanisms and their components perform their activities *ceteris paribus*, as a matter of their function. In the rest of our discussion, we will mostly focus on their normal operation, but it's important to keep in mind that they can malfunction, break, or be malformed or defective. This will help demonstrate that the mechanistic account satisfies desideratum 5 (miscomputation).

A similar notion of functional mechanism applies to computing systems that are defined purely mathematically, such as (unimplemented) Turing machines. Turing machines consist of a tape divided into squares and a processing device. The tape and processing device are explicitly defined as spatiotemporal components. They have functions (storing letters; moving along the tape; reading, erasing, and writing letters

[1] For a standard introduction to computer organization and design, see Patterson and Hennessy 1998.

on the tape) and an organization (the processing device moves along the tape one square at a time, etc.). Finally, the organized activities of their components explain the computations they perform. Mathematically defined computing mechanisms stand to concrete ones in roughly the same relation that the triangles of geometry stand to concrete triangular objects. Like the triangles of geometry, mathematically defined computing mechanisms may be idealized in various ways. Typically, they are assumed to (i) never break down and (ii) have properties that may be impossible to implement physically (such as having tapes of unbounded length).

Functional mechanisms, unlike mechanisms *simpliciter* and physical systems in general, have successes and failures depending on whether they fulfill their functions. Some conditions are relevant to explaining successes and failures while others are irrelevant. This distinction gives us the resources to distinguish the properties of a mechanism that are relevant to its computational capacities from those that are irrelevant. But mechanistic structure per se is not enough to distinguish between mechanisms that compute and mechanisms that don't. For instance, both digestive systems and computers are subject to mechanistic explanation, but only the latter appear to compute. The main challenge for the mechanistic account is to specify properties that distinguish computing mechanisms from other (non-computing) mechanisms—and corresponding to those, features that distinguish computational explanations from other (non-computational) mechanistic explanations.

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

I will begin with a broad notion of computation in a generic sense and then illustrate it by looking more closely at the paradigmatic species of computation—digital computation. In later chapters, I will discuss several classes of computing systems in more detail.

2.2 Generic computation

In order to capture all relevant uses of 'computation' in computer science and cognitive science, we need a broad notion of computation. I will use the banner of *generic computation*. Specific kinds of computation, such as digital and analog computation, are species of generic computation. The mechanistic account of concrete computation is the following:

A **Physical Computing System** is a physical system with the following characteristics:

• It is a functional mechanism—that is, a mechanism that has teleological functions.

• One of its teleological functions is to perform computations.

Generic Computation: the processing of vehicles by a functional mechanism according to rules that are sensitive solely to differences between different portions (i.e., spatiotemporal parts) of the vehicles.

Rule: a mapping from inputs **I** (and possibly internal states **S**) to outputs **O.**

In other words, a physical computing system is a mechanism whose teleological function is computing mathematical function *f* from inputs **I** (and possibly internal states **S**) to outputs **O**. The mathematical function *f* is an abstract (macroscopic) description of the behavior of a physical computing system when it fulfills its teleological function. The mathematical function *f* is also the rule followed by the physical system in manipulating its vehicles. The process of manipulating such vehicles according to such a rule is a physical *computation*. In order to understand computation in the generic sense, we need to understand the ingredients in the above account.

The term 'vehicle' can be used to refer to one of two things: either a variable, that is, a state that can take different values and change over time, or a specific value of such a variable.[2] This ambiguity is harmless because context will make clear what is meant in any given case. A vehicle consists of spatiotemporal parts or portions. For example, a string of digits (in one usage of this term) is a kind of vehicle that is made out of digits concatenated together (Figure 7.1). A digit is a portion of a vehicle that can take one out of finitely many states during a finite time interval. Typically, a digit is a binary vehicle; it can take only one out of two states. Another type of vehicle is a real (or continuous) variable. The values of real variables change continuously over time; they can take (in principle) one out of an uncountably infinite number of states at any given time.

A *rule* in the present sense is a map from inputs (and possibly internal states) to outputs. It need not be represented within the computing system; it may be followed by the system without representing it anywhere within it. It need not be spelled out by giving an algorithm or listing the <input, output> pairs (or <input × internal state,

#* $ ## $ ** $ ## $ #* $ **

Figure 7.1 A string of ternary digits. Each position in the string is a digit, and each '#', '$', and '*' is the value of a digit.

[2] A variable can be defined either as a purely mathematical or symbolic entity or as a physical state that can change over time. For present purposes, a mathematical variable is just a way of referring to a (possibly hypothetical) physical variable.

output> pairs); it may be given by specifying the relation that ought to obtain between inputs (and possibly internal states) and outputs. It need not be defined in terms of a special kind of vehicle (such as digits); it can be defined in terms of any appropriate kind of vehicle. For instance, a rule might map segments of a real variable onto the integral of those segments. Accordingly, an integrator is a device that follows the integral rule in the sense of taking as input a real variable over a time interval and generating as output the integral of that portion of real variable. Given this broad notion of rule, the present account covers all systems that are generally deemed to compute, whether or not they are "rule-based" using more restrictive notions of rule (e.g., Horgan and Tienson 1989; O'Brien and Opie 2006).

The *processing* or *manipulation* of a vehicle is any transformation of one portion of a vehicle into another. Processing is performed by a functional mechanism, that is, a mechanism with teleological functions. In the present case, a function of the mechanism is processing its vehicles in accordance with the relevant rules. Thus, if the mechanism malfunctions, a miscomputation occurs.

When we define concrete computations and the vehicles that they manipulate, we need not consider all of their specific physical properties. We may consider only the properties that are relevant to the computation, according to the rules that define the computation. A physical system can be described at different levels of abstraction (Chapter 1, Section 2). Since concrete computations and their vehicles can be defined independently of the physical media that implement them, I call them *medium-independent* (borrowing the term from Garson 2003). That is, computational descriptions of concrete physical systems are sufficiently abstract as to be medium-independent.

In other words, a vehicle is medium-independent just in case the rule (i.e., the input-output map) that defines a computation is sensitive only to differences between portions (i.e., spatiotemporal parts) of the vehicles along specific dimensions of variation—it is *in*sensitive to any other physical properties of the vehicles. Put yet another way, the rules are functions of state variables associated with certain degrees of freedom, which can be implemented differently in different physical media. Thus, a given computation can be implemented in multiple physical media (e.g., mechanical, electro-mechanical, electronic, magnetic, etc.), provided that the media possess a sufficient number of dimensions of variation (or degrees of freedom) that can be appropriately accessed and manipulated and that the components of the mechanism are functionally organized in the appropriate way. By contrast, typical physical processes, such as cooking, cleaning, exploding, etc., are defined in terms of specific physical alterations of specific substances. They are not medium-independent; thus, they are not computations.

Medium-independence is a stronger condition than multiple realizability, which is an often discussed feature of functional mechanisms. A property is multiply realizable just in case it can be fulfilled by sufficiently different kinds of causal mechanisms (Piccinini and Maley 2014). For instance, lifting corks out of bottles can be realized

by a waiter's corkscrew, a winged corkscrew, etc., all of which are different mechanisms for the same function. Medium-independence entails multiple realizability: if a process is defined independently of the physical medium in which it is realized, it may be realized in different media. But the converse does not hold; multiple realizability does not entail medium independence. In our example, every mechanism that implements a corkscrew must face specific constraints dictated by the medium within which they must operate: corks and bottles. Thus, lifting corks out of bottles is multiply realizable but not medium-independent.

Nevertheless, medium-independence shares an important and underappreciated feature with multiple realizability. That is, any medium-independent property—just like any multiply realizable property—places structural constraints on any mechanism that fulfills it (Chapter 5). In the case of lifting corks out of bottles—which is multiply realizable but not medium independent—we've already seen that there are obvious structural constraints coming from having to lift corks (as opposed to, say, caps) out of bottles (as opposed to, say, bathtubs). In the case of a medium-independent property, the structural constraint comes from requiring that the medium—any medium that realizes the property—possesses the degrees of freedom that are needed to realize the property.

In the case of digits, their defining characteristic is that they are unambiguously distinguishable by the processing mechanism under normal operating conditions. Strings of digits are ordered sets of digits; i.e., digits such that the system can distinguish different members of the set depending on where they lie along the string. The rules defining digital computations are defined in terms of strings of digits and internal states of the system, which are simply states that the system can distinguish from one another. No further physical properties of a physical medium are relevant to whether they implement digital computations. Thus, digital computations can be implemented by any physical medium with the right degrees of freedom.

Since the definition of generic computation makes no reference to specific media, generic computation is medium-independent—it applies to all physical media. That is, the differences between portions of vehicles that define a generic computation do not depend on specific physical properties of the medium but only on the presence of relevant degrees of freedom.

Analog computation is often contrasted with digital computation, but analog computation is a vague and slippery concept. The clearest notion of analog computation is that of Pour-El (1974). Roughly, abstract analog computers are systems that manipulate continuous variables to solve certain systems of differential equations. Continuous variables are variables that can vary continuously over time and take any real values within certain intervals.

Analog computers can be physically implemented, and physically implemented continuous variables are different kinds of vehicles than strings of digits. While a digital computing system can always unambiguously distinguish digits of different

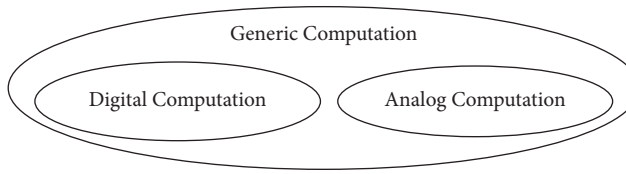


Figure 7.2 Types of generic computation and their relations of class inclusion.

types from one another, a concrete analog computing system cannot do the same with the exact values of (physically implemented) continuous variables. This is because the values of continuous variables can only be measured within a margin of error. Primarily due to this, analog computations (in the present, strict sense) are a different kind of process than digital computations.

In conclusion, generic computation includes digital computation, analog computation, and more (Figure 7.2).

In addition to the structural constraints on computation that derive from the vehicles to be realized, there are structural constraints that come from the specific operations that are realized. We shall soon see that in order to perform certain operations, components must have specific structural properties. Thus, computational explanation counts as full-blown mechanistic explanation, where structural and functional properties are inextricably mixed—because they are mutually constraining—within the mechanism (Chapter 5).

This point addresses a dilemma posed by Sabrina Haimovici (2013). Haimovici argues that since computation is medium-independent, computational explanation is purely[3] functional—whereas mechanistic explanation is both structural and functional. Thus, we must choose between individuating computations purely functionally or also structurally. Horn 1: if we individuate computations purely functionally, we cannot rely on mechanistic explanation because mechanistic explanation must involve structural properties. Horn 2: if we individuate computations structurally as well as functionally, we can rely on mechanistic explanation but we lose multiple realizability, because mechanistic explanation requires the specification of *all* relevant structural and functional properties at *all* levels of mechanistic organization, and that rules out alternative realizations.

Reply: computation is individuated both functionally and structurally, because even medium-independent properties place structural constraints on the media that realize them and the mechanisms that operate on them. In addition, I reject the view that mechanistic explanation requires the specification of structural and functional properties at all levels of organization. Instead, mechanistic explanation requires the specification of all *relevant* structural and functional properties at the *relevant*

[3] Actually, when she is careful Haimovici writes "mainly" functional (2013, 158, 159, 163, 178), which is consistent with structural properties also playing a role in individuating computations.

level(s) of mechanistic organization—in other words, mechanistic explanation of any given phenomenon requires performing appropriate abstractions from lower level details—and that leaves just the right amount of room for multiple realizability (Piccinini and Maley 2014). Mechanistic explanation in general requires abstraction, and computational explanation is an especially abstract form of mechanistic explanation—so abstract that computation is medium-independent. Therefore, we can be mechanists about computation and preserve both medium-independence and multiple realizability.

In order to understand the mechanistic account of computation further, it will help to examine a class of computing systems more closely. The rest of this section will focus on the broad characteristics of digital computing systems and how the mechanistic account sheds light on them.

2.3 Abstract digital computation

Mathematically, a digital computation, in the sense most directly relevant to computability theory and computer science, is defined in terms of two things: strings of letters from a finite alphabet and a list of instructions for generating new strings from old strings. The list of instructions is called *program*. The instructions are typically deterministic, specifying how to modify a string to obtain its successor. (In special cases, instructions may be nondeterministic, specifying which of several modifications may be made.) Given an alphabet and a list of pertinent instructions, a digital computation is a sequence of strings—sometimes called *snapshots*—such that each member of the sequence derives from its predecessor by some instruction in the list.[4]

Letters and strings thereof are often called *symbols*, because they are typically assigned semantic interpretations. Strings may or may not be assigned an interpretation; if they are interpreted, the same string may be interpreted differently—e.g., as representing a number, or a program, etc.—depending on what the theorist is trying to prove at any given time. In these computational descriptions, the identity of the computing mechanism does not hinge on how the strings are interpreted. Therefore, within computability theory, symbols do not have their content essentially. Although computability theory in a broad sense studies the computational properties of different notations (Rescorla, forthcoming), the basic mathematical theory of computation can be formulated and the core results of computability theory can be derived without assigning any interpretation to the strings of symbols being computed (e.g., Machtey and Young 1978). More generally, much research in fields that rely on computational formalisms, such as algorithmic information theory and the study of formal languages, proceeds without assigning any interpretations to the computational inputs, outputs, and internal states that are being studied. Thus, a

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

letter is simply a type of entity that (i) is distinct from other letters and (ii) may be concatenated to other letters to form lists, called *strings*. A string is an ordered sequence of letters—it is individuated by the types of letter that compose it, their number, and their order within the string.[5]

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

For all strings from an alphabet and any relevant list of instructions, there is a general rule that specifies which function is computed by acting in accordance with a program. In other words, the rule specifies which relationship obtains between the outputs produced by modifying snapshots in accordance with the program and their respective inputs. For example, a rule may say that the outputs are a series of input words arranged in alphabetical order. Such a rule has two important features. First, it is *general*, in that it applies to all inputs and outputs from a given alphabet without exception. Second, it is *input-specific*, in that it depends on the composition of the input (the letters that compose it and their order) for its application. The rule need not return an output value for all inputs; when it doesn't, the (partial) function being computed is undefined for that input. Absent a way to formulate the rule independently of the program, the program itself may count as the rule.[6]

It is important to notice that mathematically, a digital computation is a specific type of sequence defined over a specific type of entity. Many sets do not count as alphabets (e.g., the set of natural numbers is not an alphabet because it is infinite) and many operations do not count as digital computations in the relevant sense (e.g., integrating a function over a domain with uncountably many values, or generating a

[5] Dershowitz and Gurevich (2008) show that computing over standard one-dimensional strings is equivalent to computing over arbitrary data structures including two-dimensional strings (Sieg and Byrnes 1996), vectors, matrices, lists, graphs, and arrays. I focus on one-dimensional strings because they are the basis for mainstream computability theory and they are the basic data structure employed in digital computer design (more on digital computer design in the next chapters). Furthermore, any other relevant data structures may be encoded as one-dimensional strings.

[6] In Turing's original formulation, all computations begin with the same input (the empty string), but there are still rules that specify which strings are produced by each computation (Turing 1936–7). More generally, it is possible to define "computations" analogous to ordinary computations but without inputs, without outputs, or without both. Since these "computations" are of little interest for present purposes, I will ignore them.

random string of letters (Church 1940)). The mathematical notion of digital computation is clear enough. The remaining question is how to apply it to concrete mechanisms.

2.4 Digits and primitive digital computing components

To show how a concrete mechanism can perform digital computations, the first step is finding a concrete counterpart to the formal notion of letter from a finite alphabet. I call such an entity a *digit*. The term 'digit' may be used in one of two ways. First, it may denote a physical variable that belongs to a component of a mechanism and can take one out of finitely many states during a finite time interval. For example, a switch may be either on or off, or a memory cell may contain a '1' or a '0'. In this first usage, the switch or memory cell is a digit. Second, 'digit' may denote the specific state of a variable—e.g., a switch in the *off* position, a memory cell storing a '1'. This ambiguity is harmless because context will make clear what is meant in any given case. When not otherwise noted, I will use 'digit' in the second sense, meaning state of a physical variable.

A digit may be transmitted as a signal through an appropriate physical medium. It may be transmitted from one component to another of a mechanism. It may also be transmitted from outside to inside a mechanism and vice versa. So a digit may enter the mechanism, be processed or transformed by the mechanism, and exit the mechanism (to be transmitted, perhaps, to another mechanism).

So far, I have talked about physical variables that can take one out of finitely many states during a finite time interval. But, of course, a typical physical variable—including the variables to be found inside digital computers—may vary continuously and take (or be assumed to take) one out of an uncountable number of states at any given instant. And even when a physical variable may take only finitely many states, the number of physical microstates is many orders of magnitude larger than the number of digits in a computing system. Therefore, many physically different microstates count as digits of the same type.

As we saw in Chapters 2 and 4, some authors have supposed that any grouping of physical microstates within a system may count as identifying digits. Nothing could be further from the truth. Physical microstates have to be grouped in very regimented ways in order to identify a system's digits. Only very specific groupings of physical microstates are legitimate. To a first approximation, physical microstates must be grouped in accordance with the following three conditions.

Digital Grouping of Physical Microstates

(1) During any time interval when a mechanism processes digits, all and only the microstates that the mechanism processes in the *same* way while performing its function belong to the *same* digit type.

(2) During any time interval when a mechanism processes digits, all and only the microstates that the mechanism processes in *different* ways while performing its function belong to *different* digit types.
(3) During any time interval when a mechanism does *not* perform its function or performs its function but does not process digits, microstates do *not* belong to any digit.

When I write that two microstates are "processed in the same way," I mean that they are *always* transformed into microstates that belong to the same digit type. That is, they are transformed into microstates that belong to the same digit type not only during an actual situation but under all conditions of *normal* operation. The conditions must be normal because a computing mechanism may malfunction and generate the wrong microstate.

Once microstates are grouped into digits, they can be given abstract labels such as '0' and '1'. Which abstract label is assigned to which digit type is entirely arbitrary as long as the number of labels matches the number of digit types. When we label the digits of computing artifacts, labeling is driven by application: we choose labels that fit our purposes and we design circuits to fulfill our purposes based on the chosen labeling scheme. But from a physical standpoint, any labeling scheme that uses the right number of labels is equivalent to any other (cf. Sprevak 2010; Dewhurst ms.).

Notice that digits, the time intervals during which they are processed, and the ways they are processed are *mutually defined*. The time intervals are defined in terms of digit processing and operations on digits, the digits are defined in terms of the relevant time intervals and ways of processing the microstates, and the ways of processing microstates are defined in terms of digits and relevant time intervals. This is not a vicious circularity and does not give rise to arbitrary groupings of microstates (more on this in the next section); it simply means that when a digital computing system is designed, its digits, ways of processing them, and relevant time intervals must be designed together; when a natural system is interpreted to be a digital computing system, its digits, ways of processing them, and relevant time intervals must be identified together.

Let us now get into more details about digits and their processing. It is convenient to consider strings of one digit first, leaving strings of multiple digits for later. A logic gate is a device that takes one or two input digits and returns one or two output digits as a function of its input. Logic gate computations are so basic that they cannot be analyzed into simpler computations. For this reason, I call logic gates *primitive computing components*. Logic gates are the computational building blocks of modern digital computing technology.[7]

[7] Some computing mechanisms, such as old mechanical calculators, are not made out of logic gates. Their simplest computing components may manipulate strings of multiple digits, as opposed to a few separate digits, as inputs and outputs. Their treatment requires the notion of a concrete string, which is introduced below. Without loss of generality, we may consider primitive components with two inputs and

Digits are permutable in the sense that normally, any token of any digit type may be replaced by a token of any other digit type. Functionally speaking, the components that bear digits of one type are also capable of bearing digits of any other type. For example, ordinary digital computer memory cells must be able to stabilize on states corresponding to either of the two digit types—usually labeled '0' and '1'—that are manipulated by the computer. If memory cells lost the capacity to stabilize on one of the digit types, they would cease to function as memory cells and the digital computer would cease to work.

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

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

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

For now, I hope the example of electronic logic gates is enough to grasp the basic idea. I will add some more pertinent observations later in this chapter. Provided that

one output, since primitive components with a larger number of inputs and outputs are reducible to components with only two inputs and one output. (This condition may not hold in the case of hypercomputation, which is briefly mentioned in the next section and more extensively in Chapters 15 and 16. Here, we are focusing on ordinary, recursive digital computation.)

the relations just discussed hold, a mechanism may be described as performing elementary (atomic) digital computations, because its inputs and outputs are digits, and the relation between inputs and outputs may be characterized by a simple logical relation. But elementary digital computations are trivial. When we talk about digital computing, we are generally interested in computation over strings of nontrivial length. For that, we need to introduce a concrete notion of string, which requires a concrete ordering of the digits.

2.5 Strings of digits and complex digital computing components

Any concrete relation between digits that possesses the mathematically defined properties of concatenation may constitute a concrete counterpart to mathematical concatenation. The simplest examples of such concrete ordering relations are spatial contiguity between digits, temporal succession between digits, or a combination of both.

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

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

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

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

For example, consider a circuit that adds two four-digit strings.[8] A simple way to perform binary addition is the following: add each pair of bits; if there is a carry from

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

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

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

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

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

Yet other functional relations may be used to implement concatenation. Within modern, stored-program digital computers, computation results are stored in large memory components. Within such memories, the concatenation of digits into strings is realized neither purely spatially nor purely temporally. Rather, there is a system of memory registers, each of which has a label, called address. If a string is sufficiently long, a memory register may contain only a portion of it. To keep track of a whole string, the computer stores the addresses where the string's parts are stored. The order of register names within the list corresponds to the relation of concatenation between the parts of the string that is stored in the named registers. By exploiting this

mechanism, a digital computer can store very large strings and keep track of the digits' order without needing to possess memory components of corresponding length.

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

2.6 Components, functions, and organization

As we've seen, digits and strings thereof are equivalence classes of physical microstates that satisfy certain conditions. For instance, all voltages sufficiently close to two set values that occur within relevant time intervals in relevant components count as digit tokens of two different types; all voltages sufficiently far from those values do not count as digits at all. But voltage values could be grouped in many ways. Why is one grouping privileged within a digital computing mechanism? The answer has to do with the components of the mechanism, their functional properties, and their organization.

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

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

Given their special functional characteristics, digits can be labeled by letters and strings of digits by strings of letters. As a consequence, the same formal operations and rules that define mathematically defined computations over strings of letters can be used to characterize concrete computations over strings of digits. Within a concrete digital computing mechanism, the components are connected so that the

inputs from the environment, together with the digits currently stored in memory, are processed by the processing components in accordance with a set of instructions. During each time interval, the processing components transform the previous memory state (and possibly, external input) in a way that corresponds to the transformation of each snapshot into its successor. The received input and the initial memory state implement the initial string of a mathematically defined digital computation. The intermediate memory states implement the intermediate strings. The output returned by the mechanism, together with the final memory state, implement the final string.

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

The synchronization provision is crucial and often underappreciated. Components must be synchronized so that they update their state or perform their operations within appropriate time intervals. Such synchronization is necessary to individuate digits appropriately, because synchronization screens off irrelevant values of the variables some values of which constitute digits. For example, when a memory component changes its state, it transitions through all values in between those that constitute well defined digits. Before it stabilizes on a new value, it takes on values that constitute no digits at all. This has no effect on either the proper taxonomy of digits or the mechanism's computation. The reason is that memory state transitions occur during well-defined time intervals, during which memory components' states do not affect the rest of the mechanism. This contributes to making the equivalence classes that constitute digits functionally well-defined. Thus, synchronization is a necessary aspect of the computational organization of ordinary digital computing technology. Without synchronization there would be no complex digital computations, because there would be no well-defined digits.

In short, any system whose function is generating output strings from input strings (and possibly internal states), in accordance with a general rule that applies to all strings and depends on the input strings (and possibly internal states) for its

application, is a digital computing mechanism. The mechanism's ability to perform computations is explained mechanistically in terms of its components, their functions, and their organization. By providing an appropriate mechanistic explanation of a system, it is thus possible to individuate digital computing mechanisms, the functions they compute, and their computing power, and to explain how they perform their computations.

2.7 How some computing mechanisms understand instructions

Ordinary digital computers execute instructions, and which computation they perform depends on which instructions they execute. This special capacity has suggested to some authors that instructions must be individuated by their semantic properties. Thus, a general account of computation must address computing mechanisms that execute instructions and whether their individuation requires semantic properties.

In the practice of computer programming, programs are created by combining instructions that are prima facie contentful. For example, a high-level programming language may include a control structure of the form UNTIL P TRUE DO ___ ENDUNTIL.[9] In executing this control structure, the computer does ___ until the variable P has value TRUE and then moves on to the next instruction. The programmer is free to insert any legal sequence of instructions in the ___, knowing that the computer will execute those instructions until the value of P is TRUE. This awesome ability of computers to execute instructions is one of the motivations behind the semantic account of computation, according to which computation requires representation and computational states have their semantic content essentially (Chapter 3). For when people execute instructions—that is, they do what the instructions *say* to do—they do so because they *understand* what the instructions say. By analogy, it is tempting to conclude that, in some sense, computers respond to the semantic properties of the instructions they execute, or at least instructions and the corresponding computational states of the mechanism are individuated by their content. This temptation is innocuous to the extent that we appreciate how computers execute instructions and specify the relevant notion of computational content accordingly; otherwise, to speak of computers responding to semantic properties or of instructions being individuated by their content is misleading. So, I will briefly explain how computers execute instructions (more details in Chapters 9 and 11).

In ordinary stored-program computers, instructions are encoded as binary strings (strings of bits). Each bit is physically realized by a voltage level in a memory cell or some other state capable of physically affecting the computer in the relevant way. Before the processor of a computer can execute a binary string written in a high-level programming language, the computer must transform the string into a machine language instruction, which the machine can execute. A machine language instruction

[9] I took this example from Loop Programs, a simple but powerful programming language invented by Robert Daley at the University of Pittsburgh.

is a binary string that, when placed in the appropriate register of a computer processor, *causes* the computer's control unit to generate a series of events in the computer's datapath. For example, the sequence of events may include the transfer of binary strings from one register to another, the generation of new strings from old ones, and the placement of the new strings in certain registers.

The computer is designed so that the operations performed by the computer's processor (i.e., the control unit plus the datapath) in response to a machine language instruction correspond to what the instruction means *in assembly language*. For instance, if the intended interpretation of an assembly language instruction is to copy the content of register *x* into register *y*, then the computer is designed so that when receiving a machine language encoding of that assembly language instruction, it will transfer the content of register *x* into register *y*. This feature of computers may be used to assign their instructions (and some of their parts) an interpretation, to the effect that an instruction asserts what its execution accomplishes *within* the computer. This may be called the *internal* semantics of the computer.

Internal semantics is not quite semantics in the sense usually employed by philosophers. When philosophers say 'semantics,' they mean *external* semantics, that is, semantics that relates a state to things *other* than its computational effects within a computer, including objects and properties in the external world. But notice that contents assigned to a state by an external semantics need not be ordinary objects and properties in the environment; they may be numbers, counterfactual events, phonemes, non-existent entities like Aphrodite, etc.[10]

Internal semantics is no help to the supporters of semantic accounts of computation, for they are usually concerned with individuation by external semantic properties. This is because semantic accounts are largely motivated by computational explanations of cognitive capacities in terms of internal states and processes that are widely assumed to be individuated by their (external) contents.

Internal semantics is fully determined by the functional and structural properties of program-controlled computers, independently of any external semantics. This can be seen clearly by reflecting on the semantics of high-level programming language instructions. For instance, the semantics assigned above to UNTIL P TRUE DO ___ ENDUNTIL was ambiguous between an internal and an external reading. As I said, the instruction means to do ___ until the variable P has value TRUE. Doing ___ is a computational operation, so this component of the interpretation is internal. P is a variable of the programming language, which ranges over strings of symbols to be found inside the computer—again, this is an internal content. Finally, 'TRUE' may

[10] Dennett (1987, 224–5) uses the expressions "internal semantics" and "external semantics" in a similar sense, and Fodor (1978) discusses some related issues. Curiously, I devised and named this distinction before reading Dennett's work. The distinction between internal and external semantics should not be confused with that between semantic internalism and semantic externalism, which pertain to the identity conditions of contents (specified by an external semantics).

be taken to mean either *true* (the truth-value), or the word 'TRUE' itself as written in the relevant programming language (a case of self-reference).

When writing programs, it is convenient to think of 'TRUE' as referring to a truth-value. But for the purpose of individuating the computation, the correct interpretation is the self-referential one. For what a computer actually does when executing the instruction is to compare the (implementations of) the two strings, the one that is the value of P and the one that reads 'TRUE.' All that matters for the individuation of the computation is which digits compose the two strings and how they are concatenated together. If they are the same digits in the same order, the processor proceeds to the next instruction; otherwise, it goes back to doing ___. Whether either string (externally) means a truth-value, or something else, or nothing, is irrelevant to determining which state the computer is in and which operation it's performing for the purpose of explaining its behavior. In other words, having an internal semantics does not entail having an external semantics.

This is not to say that instructions and data, either at a high level or at the machine language level, *lack* an external semantics. Each element of the machine implementation of a high-level instruction has a job to do, and that job is determined at least in part by the high-level instruction that it implements. Besides its internal semantics, that high-level instruction may well have a semantics that is, at least in part, external. By the same token, each element of a machine language datum is a component of the machine implementation of a high-level datum, and that high-level datum typically has an external semantics. It may be difficult or impossible to univocally break down the external contents of high-level instructions and data into external contents of machine language instructions and data, but this is only an epistemic limitation. As a matter of fact, machine language instructions and data may well have external semantic properties. This is perfectly compatible with the point at issue. The point is that the states of computing mechanisms, including program-controlled computers, do not have their external contents essentially—they are fully individuated without appealing to their external semantic properties.

Assigning instructions and data a semantics, either external or internal, is indispensable to designing, programming, using, and repairing computers, because that is the only way for designers, programmers, users, and technicians to understand what computers are supposed to be doing or failing to do. But in an explanation of computer instruction execution, a complex instruction like UNTIL P TRUE DO ___ ENDUNTIL is a string of digits, which will be encoded in the computer as a binary string, which will affect the computer's processor in a certain way. A computer is a powerful, flexible, and fascinating mechanism, and we may feel compelled to say that it responds to the semantic properties of the instructions it executes. But as I briefly argued, this kind of "computer understanding" is exhaustively and mechanistically explained without ascribing any external semantics to the inputs, internal states, or outputs of the computer. The case is analogous to non-universal TMs,

whose computational behavior is entirely determined and fully individuated by the instructions that are "hardwired" in their active component.

In summary, the mechanistic account of computation holds that a program-controlled computer is a physical system with special functional and structural properties that are specified by a certain kind of mechanistic explanation. Although for practical purposes the internal states of computers are usually ascribed content by an external semantics, this need not be the case and is unnecessary to individuate their computational states and explain their behavior.

From now on, unless otherwise noted, by 'semantics' I will mean external semantics, and by 'content' I will mean content ascribed by an external semantics (unless otherwise noted).

2.8 Computing mechanisms, semantic content, and wide functions

The mechanistic account of computation bears some similarity to a view proposed by Frances Egan (1992, 1995, 1999, 2003, 2010). Egan appears to reject the semantic account of computation, because she rejects the view, championed by many philosophers, that the computational states postulated by psychological theories are individuated by the *cognitive* contents of those states (e.g., visual contents for the states of visual mechanisms, auditory contents for the states of auditory mechanisms, etc.).

Cognitive contents are often assumed to be non-individualistic—that is, they are assumed to vary based solely on differences in cognizers' environments, even when cognizers cannot tell such a difference. Be that as it may, Egan argues that computational states are individuated *individualistically*, i.e., by properties that are shared by all physical duplicates of a mechanism. But when Egan specifies how computational states are individuated, she points to their "mathematical contents," namely the "mathematical" functions whose domain and range elements are denoted by the inputs and outputs of the computations (Egan 1995, 187; 2003, 96). Although I agree with much of what Egan says, Egan's view does not capture the way computational states are individuated within computability theory and computer science; therefore, it should be replaced by the mechanistic account of computation. Egan's view also faces an internal difficulty, which is avoided by the mechanistic account.

Egan's mathematical contents behave differently from cognitive contents in some types of counterfactual reasoning. A salient difference is that mathematical contents—unlike cognitive ones—are not dependent on the relations between a mechanism and its environment. Under most views of cognitive content, whether an organism is thinking about water depends, inter alia, on whether there is H_2O in her environment (Putnam 1975c). But whether the same organism is thinking about the number seven does not seem to depend on anything in her environment. In this sense, mathematical contents are shared by physical duplicates in a way that cognitive contents (under most views of cognitive content) are not.

But there is a sense in which mathematical contents are no more intrinsic to computing systems than cognitive contents. Mathematical contents are still contents (ascribed by an external semantics)—they are still relational properties of states, which depend on the relations between a mechanism and something else (numbers, sets, or what have you). From a formal semantics perspective, there is no principled difference between mathematical and cognitive contents. Both can be assigned as interpretations to the states of a system, and both can be assigned to the system's physical duplicates. It is certainly possible to assign the same mathematical interpretation to all physical duplicates of a computing system, but in the same way, it is equally possible to assign the same cognitive interpretation to all physical duplicates of a computing system.[11] Moreover, just as internal states of the same mechanism may be given different cognitive interpretations, it is well known that the same set of symbolic strings may be given different mathematical interpretations. And the same sets of vehicles can be given different mathematical interpretations in different contexts (cf. Rescorla 2013). In this sense, mathematical contents are shared by physical duplicates neither more nor less than cognitive contents. If the latter are not individualistic enough for Egan's purposes, the former shouldn't be either.

If someone wants to individuate computational states in a rigorously individualistic way, she should drop the individuation of computational states by their semantic properties—cognitive or mathematical—altogether. She might opt for an individualistic version of the mechanistic account of computation: under a narrow construal of mechanistic explanation, the properties of computing mechanisms are individualistic in precisely the sense desired by Egan.[12]

My own version of the mechanistic account of computation is *not* individualistic but *anti*-individualistic. My immediate reason is the kind of consideration used by Shagrir (2001) in his argument from the multiplicity of computations (discussed in Chapter 3, Section 3). Shagrir shows that looking at the system's narrow properties is insufficient to determine which computational description is correct among the many that are satisfied by a system. Shagrir's conclusion is that only a correct semantic interpretation of the system allows us to determine the semantically individuated computation actually performed by the system. My alternate conclusion

[11] Of course, some of those interpretations may turn out to be intuitively anomalous within a cognitive theory of an organism, in the sense that they may fail to capture the way the organism relates to her actual environment (as opposed to a possible environment). In computer science, however, all that matters for interpreting computational states is the formal adequacy of a candidate interpretation, that is, whether the states can be systematically interpreted in one way or another. There is nothing intuitively anomalous about interpreting a computer on Twin Earth as computing something about H_2O, even if there is no H_2O on Twin Earth. In this respect, the semantics of artificial computing mechanisms is different from that of organisms. Perhaps this is because the semantics of computing mechanisms is derived, whereas that of organisms is original.

[12] This is individualism about *computing* mechanisms, not about *psychological* mechanisms. A narrow reading of the mechanistic account of computation is compatible with there being psychological computing mechanisms that include features of both individuals and their environment, as argued by Wilson (1994, 2004).

is that only by taking into account the functional interaction between the system and its context can we determine the *non*-semantically individuated computation actually performed by the system.

More generally, I embrace a wide (non-individualistic) construal of mechanistic explanation. For present purposes, it is important to distinguish between wide individuation and individuation based on wide semantic content. Individuation based on wide content is one type of wide individuation, but wide individuation is a broader notion. Wide individuation appeals to the relations between a mechanism and its context, relations which may or may not be semantic. For my purposes, of course, what is needed is wide individuation that does not appeal to semantic relations.

Mechanisms have many intrinsic properties, only some of which are functionally relevant. In order to know which intrinsic properties of mechanisms are functionally relevant, it may be necessary to consider the interaction between mechanisms and their contexts.[13] For instance, plants absorb and emit many types of electromagnetic radiations, most of which have little or no functional significance. But when radiation within certain frequencies hits certain specialized molecules, it helps produce photosynthesis—an event of great functional significance. Without knowing which external events cause certain internal events and which external effects those internal events have, it may be difficult or impossible to distinguish the functionally relevant properties of a mechanism from the irrelevant ones. As a consequence, scientific theories typically individuate the functional properties of mechanisms widely.[14]

The same point applies to the functional properties of computing systems. As Harman (1988) points out, many philosophers have assumed that computing systems are individuated purely individualistically (Putnam 1967b; Fodor 1980; Stich 1983). But this assumption is false. Concrete computing systems, like all other mechanisms, have many intrinsic properties, only some of which are relevant to the results of their computations. For instance, many ordinary computers would not work for very long without a fan, but the fan is not a computing component of the computer, and blowing air is not part of the computer's computations. As with any other mechanism, we need to distinguish the properties of a computing system that are functionally relevant from the ones that are irrelevant. We also need to distinguish the functional properties that are relevant to computation from the irrelevant ones. In order to draw these distinctions, we need to know which of a computing system's properties are relevant to its computational inputs and outputs and how

[13] The context of a mechanism need not coincide with the environment of an organism. If a mechanism is an internal component of a larger system, its context is constituted by other relevant components of the system and their activities.

[14] A similar view is defended by Kitcher 1985; Harman 1988; and Shapiro 1994. These authors do not refer to mechanistic explanation but to functional analysis. I addressed the relationship between functional analysis and mechanistic explanation in Chapter 5. Here, suffice it to say that, in my view, functional analysis is a kind of mechanistic explanation, and the same considerations that favor wide functional explanation over narrow functional explanation favor, more generally, wide mechanistic explanation over narrow mechanistic explanation.

they are relevant. In order to know that, we need to know what are the computational inputs and outputs of the mechanism. That, in turn, requires knowing how the mechanism's inputs and outputs interact with their context. In Chapter 3, I adapted Shagrir's argument from the multiplicity of computations—originally intended to support the semantic account of computation—to support this conclusion. I conclude that the mechanistic account of computation should be based on a wide construal of functional properties.[15]

At this juncture, someone might worry that at least in the case of computing mechanisms, wide functional individuation and individuation by wide content are equivalent. For instance, a wide function of an internal state might be to co-vary with an external variable. Under some theories of content, this is the same as representing that variable. If so, it may seem that wide functional individuation is the same as individuation by wide content, and that the mechanistic account of computation collapses into the semantic account. In response to this worry, I have two points to make.

First, the functional properties that are relevant to computational individuation, even when they are wide, are not *very* wide. They have to do with the normal interaction between a computing mechanism and its immediate mechanistic context via its input and output transducers. In the case of artificial computing mechanisms, the relevant context is, at one end, the relation between the forces exerted on input devices (such as keyboards) and the signals relayed by input devices to the computing components, and at the other end, the relation between the computing components' outputs and the signals released by the output devices. Those relations, together with the internal relations between components and their activities, determine whether a computation is performed by a mechanism and which computation it is.

By the same token, in the case of organisms, the wideness of putative computational properties of nervous systems does not even reach into the organisms' environment; it only reaches sensory receptors and muscle fibers, for that is enough to determine whether a nervous system performs computations and which computations it performs. As a matter of fact, the main piece of empirical evidence that was originally employed by McCulloch and Pitts (1943) to justify the first computational theory of cognition was the all-or-none properties of neural signals, and those properties were originally discovered and identified to be functionally significant by studying the interaction between neural signals and muscle fibers.[16]

[15] Again, this is compatible with Wilson's wide computationalism (1994, 2004), according to which a psychological computing mechanism may spatially extend beyond the boundaries of an organism, but it is also compatible with the negation of Wilson's view. I have argued that functional (including computational) properties are partially individuated by their interactions between a mechanism and its context. I am officially neutral on whether the components of psychological computing mechanisms extend beyond the spatial boundaries of organisms.

[16] For more on the discovery of the all-or-none properties of neural signals, see Frank 1994. For a detailed study of the considerations about the physiology of neurons that are at the origin of the computational theory of cognition see Piccinini 2004a.

Second, the extent to which wide functional properties are the same as wide contents depends on which theory of content one adopts. In most of the literature on wide contents, wide contents are largely ascribed by intuition, and theories of content are tested by determining whether they agree with the relevant intuitions. By contrast, according to the mechanistic account of computation, the functional properties that are relevant to the computational individuation of a mechanism are to be found by elaborating mechanistic explanations under the empirical constraints that are in place within the natural sciences. This establishes the computational identity of a mechanism without appealing to any semantic intuitions. Perhaps, under some theories of content, some wide semantic properties will turn out to supervene on some computational properties. But this is not a weakness of the mechanistic account—it's a strength. For under the mechanistic account, computational properties can be discovered and individuated without appealing to semantic properties, thereby providing kosher naturalistic resources that may be used in a theory of content.

The same point may be put in the following way. One problem with naturalistic theories of content that appeal to computational properties of mechanisms is that, when conjoined with the semantic account of computation, they become circular. For such theories explain content (at least in part) in terms of computation, and according to the semantic view, computational states are individuated (at least in part) by contents (Chapter 3, Section 2). The mechanistic account breaks this circle: computations are individuated by (somewhat wide) functions; contents may then be explained (at least in part) in terms of computations, without generating any circularity.

This defense of wide mechanism completes the articulation of the mechanistic account of computation. According to the mechanistic account, concrete computing systems are mechanisms that process vehicles according to rules that are sensitive to differences between different portions (i.e., spatiotemporal parts) of the vehicles. What counts as a vehicle and what manipulations count as computations may be determined in part by the way the mechanism interacts with its mechanistic context.

3. The Mechanistic Account Satisfies the Six Desiderata

The mechanistic account satisfies the desiderata listed in Chapter 1.

3.1 Objectivity

Given the mechanistic account, computational descriptions are neither vacuous nor trivial. The account relies on the way the relevant communities of scientists analyze mechanisms into their components, functional properties, and organization. As a result, whether a concrete system is a (nontrivial) computing mechanism and what it computes are matters of empirical fact.

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

What we want to avoid is observers who share the same explanatory goal and yet ascribe different computations to the same system. Under the mechanistic account, this is not an option any more than it's an option for different observers to attribute different noises to the same heart. For example, either something is a memory register or not, an arithmetic-logic unit or not, etc., depending on what it contributes to its containing mechanism. It is certainly possible to label the digits processed by a digital computing mechanism using different letters. But these do not constitute alternative computational descriptions of the mechanism; they are *merely notational variants*, all of which attribute the same computation to the mechanism. In short, the mechanistic description of a computing system is no less objective than any other mechanistic description in biology or engineering. What is computed by which mechanism is a matter of fact.

3.2 Explanation

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

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

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

Program execution is an interesting capacity of certain mechanisms, including computing mechanisms, and it is explained mechanistically. When combined with the capacity to perform computations, which is also explained mechanistically, program execution results in a powerful kind of computing mechanism—soft-programmable computers—whose computations are explained in part by program execution. I discuss the notion of program execution and its application to computing in more detail in Chapter 11.

3.3 The right things compute

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

The mechanistic account also counts (typical) neural networks as performing computations. Neural networks can be decomposed into units with functions and an organization, and hence they are mechanisms in the present sense. Many neural networks take input strings of digits and return output strings of digits in accordance with an appropriate rule, and hence they are digital computing mechanisms. Unlike ordinary computing mechanisms, the units of neural networks need not be logic gates; therefore, it may be impossible to decompose neural network computations into simpler computations (such as those performed by logic gates). The capacities of paradigmatic neural networks still have a mechanistic explanation, but such an explanation does not involve the decomposition of their computations into simpler computations performed by their components. Like logic gates, many neural networks are computationally primitive.

The units of some neural networks have activation values that vary along a continuum, so such activation values may appear to be something other than digits. But in fact, in many such neural network formalisms, these activation values are

"read" as inputs to and outputs from the whole system only when they approximate certain standard values at functionally significant times. In this respect, they aren't different from the activation values of the components of digital computers, which also vary along a continuum but are functionally significant only when they approximate certain standard values at functionally significant times. As a result, the functionally significant activation values of input and output units of this type of neural network constitute digits, and the activation values of whole input and output layers of units constitute strings of digits. An appropriate rule can then be used to characterize the relationship between the input and output strings. In fact, this is precisely how this type of neural network is described when theorists study the functions they compute (cf. Hopfield 1982; Rumelhart and MacClelland 1986; Minsky and Papert 1988; Siegelmann 1999; cf. Chapter 12).

Other neural networks process non-digital vehicles, but such vehicles are still defined independently of the physical medium of implementation. Thus, they all count as computing systems (cf. Chapter 12).

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

The only functions that are known to be physically computable are the recursive ones. There is an ongoing controversy over the physical possibility of hypercomputers—mechanisms that compute non-recursive functions (Copeland 2002; Cotogno 2003). That controversy should not be resolved by stipulating that hypercomputers don't perform computations, as is sometimes done. A good account of computing mechanisms should be able to accommodate hypercomputers (cf. Chapters 14 and 15). This highlights another advantage of the mechanistic account.

Many traditional accounts are formulated in terms of either a canonical formalism, such as Turing machines (Putnam 1967b), or the standard notion of computer program (Fodor 1975; Cummins 1983). Since standard computer programs and formalisms can only compute recursive functions, accounts based on them cannot accommodate hypercomputers. The mechanistic account, by contrast, is formulated in terms of generic rules defined over the vehicles of the computation. If those rules are recursive, we obtain the usual class of computing systems. If those rules are not recursive, we obtain various classes of hypercomputers.

But the mechanistic account does distinguish between genuine and spurious hypercomputers. Genuine hypercomputers are mechanisms that have the function of generating output strings of digits from input strings of digits in accordance with a non-recursive rule. Alan Turing's oracle machines are an example (Turing 1939; cf. Copeland 2000 and Piccinini 2003a for discussion). Spurious hypercomputers are

physical processes that are non-recursive in some way, but do not have the function of generating strings of digits in accordance with a non-recursive rule. Genuine random processes are an example.

The distinction between genuine and spurious hypercomputers clarifies the debate over hypercomputation, where considerations pertaining to spurious hypercomputers are often mixed up with considerations pertaining to genuine hypercomputers. If we don't draw this distinction, it is relatively easy to show that "hypercomputation" is possible. Any genuine random process will do. But this is not an interesting result, for a random process cannot be used to generate the desired values of a function. Hypercomputation is interesting in so far as it promises desirable strings of output digits related to their inputs in a non-recursive way. For something to be a genuine hypercomputer, it must be possible to specify the rule relating the inputs to the outputs without waiting for the physical process to take place.

Finally, analog computers do not manipulate strings of digits but real (i.e., continuous) variables. Since such real variables are defined independently of the physical medium of implementation, analog computers are covered by the present account. A more detailed account of analog computers in terms of their components, functions, and organization is presented in Chapter 11.

3.4 The wrong things don't compute

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

To begin with, most systems—including planetary systems and the weather—are not functional mechanisms in the present sense, because they are not collections of components functionally organized to fulfill specific teleological functions.[17] Also, most systems—including, again, planetary systems and the weather—do not receive inputs from an external environment, process them, and return outputs distinct from themselves. It is not difficult to cook up notions of input that apply to all systems. For instance, sometimes initial conditions or time instants are said to be inputs. But these are not entities or states that can enter the system, persist within the system, and finally exit the system. Hence, they do not count as computational inputs in the present sense.

[17] To be sure, there are accounts of function according to which planetary systems and the weather have functions. But no one disputes that they lack *teleological* functions. As I pointed out in Chapter 6, I am working with a teleological notion of function.

In addition, most functional mechanisms—including functional mechanisms that manipulate inputs and return outputs distinct from themselves and their states—lack the function of manipulating vehicles that are defined independently of the physical medium in which they are implemented. Digestive systems are a good example. As a preliminary observation, there is no prominent scientific theory according to which digestion is a computational process. In other words, the current science of digestion does not identify the inputs and outputs of digestion in a medium-independent way. Instead, the science of digestion identifies food types first and foremost by the *family* of macromolecules to which its constituents belong (carbohydrates, fats, or proteins). Different families are processed differently. What matters most to digestive function are not the details of how molecules of different chemical types, or belonging to different families, form pieces of food. On the contrary, digestion mixes and breaks down pieces of food by mechanical and chemical means, obliterating temporal, spatial, and chemical connections between molecules until the resulting products can be either absorbed by the body or discarded.

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

A more serious difficulty is that the most important taxonomy of inputs for the science of digestion has little to do with food bites. Bites of food come in indefinitely many sizes, shapes, and compositions, but the processes that take place during digestion are not defined in terms of the size, shape, or composition of food bites. Furthermore, even if bites of food could somehow be classified into computationally relevant types, their temporal order would not constitute a string of digits. For digestion, unlike computation, is largely indifferent to the order in which organisms ingest their food bites.[18] On one hand, the products of digestion always come out in roughly the same order, regardless of how the inputs came in. On the other hand, the first operations typically performed by organisms on the food they ingest eliminate most differences between bites of food. Upon being ingested, food is chewed, mixed

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

with saliva, swallowed, and mixed with digestive fluid. The result, far from being responsive to any obvious differences between bites of food or their order, is a relatively uniform bolus.

The bottom line is that, to the best of our knowledge, digestion is defined in terms of specific chemical changes to specific families of molecules. These changes are quintessentially medium-*dependent*. Thus, digestion is a mechanistic process, yes, but a medium-dependent one. It is not a computation, which is a medium-*independent* mechanistic process.

The purpose of these observations is not to prove definitively that digestion is not computational. Ultimately, according to the present account, whether digestion is computational is an empirical question, to be answered by the science of digestion. What I have shown is that under the present account, treating the digestive system as computational faces considerable challenges. The common intuition that digestion is not computational might be wrong, but it's plausible for good reasons.

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

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

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

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

In conclusion, how many things taken in isolation constitute a logic gate, or other primitive computing components, is a matter that can be left vague. For primitive computing components in isolation perform computations that cannot be decomposed into simpler computations performed by their components. The mechanistic account has the most interesting things to say about mechanisms that manipulate complex vehicles and are computationally decomposable. And unlike computers and other nontrivial computing mechanisms, most systems, including most mechanisms, neither manipulate complex vehicles nor are computationally decomposable. Since paradigmatic examples of non-computing systems do not appear to be subject to the relevant kind of mechanistic explanation, the mechanistic account properly counts them as systems that do not compute in the relevant sense.

3.5 Miscomputation

The mechanistic account of computation explains what it means for a computing system to produce the wrong output. Recall from Chapter 1, Section 4 that, to a first approximation, a system M miscomputes just in case M is computing function f on input i, $f(i) = o_1$, M outputs o_2, and $o_2 \neq o_1$. This notion of miscomputation, applied either to a complete system or to one of its computing components, will be good enough for present purposes.

To give a clear account of miscomputation, we must notice an important aspect of performance evaluations. The performance of an organism's trait or an artifact can be evaluated from many perspectives. This is especially important for artifacts, because many different agents may participate in the design, construction, preparation, and use of artifacts. To a first approximation, there are five perspectives from which the performance of an artifact can be evaluated: the designers' intentions, the designers' blueprint, what the makers construct, the preparers' intentions, and the users' needs.

If everything goes well, these five perspectives are aligned—they assign the same function to the same functional mechanism because the system is correctly used to perform the very same function it was correctly designed, built, and prepared for. But these perspectives can also come apart. The same physical system may be intended by its designers to perform function F_1, actually designed to perform a different function F_2, built in such a way that it performs a third function F_3, prepared to perform a fourth function F_4, and used to perform a fifth function F_5. This is especially evident in the case of computing systems, whose teleological function is computing a mathematical function. A computing system may be intended by its designers to

compute mathematical function f_1, actually designed to compute a different function f_2, built in such a way that it actually computes a third function f_3, prepared to compute a fourth function f_4, and used to compute a fifth function f_5.

These five perspectives give rise to five notions of miscomputation: (i) miscomputation relative to the designer's goals, (ii) miscomputation relative to the designer's blueprint, (iii) miscomputation relative to what was actually built, (iv) miscomputation relative to the preparer's goals, and (v) miscomputation relative to the user's goals.

(i) Miscomputation can occur relative to the designer's intentions, because the designer made mistakes in designing the system and the mechanism that was actually designed does not in fact compute the function it was intended to compute. The design mistake may be due to a computing component that does not compute what it was intended to compute or to a non-computing component that does not fulfill its function (e.g., a clock with a too short cycle time). If the manufacturer implements the design correctly, the resulting miscomputation is certainly not the manufacturer's fault, and a fortiori not the system's fault. In other words, this kind of miscomputation is not a malfunction but a case of incorrect design.

(ii) Miscomputation can also occur relative to the designer's blueprint, when a system was designed correctly but built incorrectly. For example, suppose Intel correctly designs a new microchip but a faulty manufacturing process introduces a defect, such that, under certain conditions, the microchip computes a different function than the one that was specified in the blueprint. Whether the system is said to malfunction depends on how it is typed (Chapter 6). This gets especially tricky if Intel catches the mistake and corrects the manufacturing process, so that from a certain time on all new chips perform in accordance with the blueprint. If the defective chips are typed together with the correctly performing chips (or if they are typed simply based on the blueprint), they are malfunctioning. If the defective chips are typed on their own (as belonging to a type distinct from the correctly functioning chips), they are functioning correctly. In practice, at least when it comes to artifacts, we can choose either way of categorizing computing systems depending on our practical purposes.

(iii) Miscomputation relative to what was actually built is a straightforward *malfunction*, i.e., an event in which a functional system fails to perform its teleological function. As we saw in Chapter 6, a system can only have teleological functions that it, or at least other tokens of its type, can perform. Therefore, a system cannot have the teleological function of doing something that was *merely intended* by the designer or is *merely intended* by the user, where such intentions are incompatible with the way the system actually works. In the case of computing mechanisms, whose function is to compute,

failing to perform their function—the function they were built to perform—may result in a miscomputation. This is hardware failure, i.e., failure of a hardware component to perform its function. Hardware failure may be due to the failure of a computing component, such as a logic gate, or of a non-computing component, such as a battery.

(iv) A computing system that is designed and built correctly may still miscompute relative to the programmer's goals if it is programmed or configured incorrectly. Programming errors include the introduction in the system of ungrammatical instructions (which cannot be executed) or of instructions that are grammatically correct but do not play their intended role within the program. This type of miscomputation is not a malfunction but is due to programming mistakes.

(v) Finally, a computing system can be designed, built, and programmed correctly but still miscompute relative to the user's goals because it is used incorrectly. For example, a user may insert the wrong input or request the wrong operation on the part of a computer. The output is likely to be different from what the user wanted; if so, it is the user's fault. Again, this type of miscomputation is not a malfunction of the system—it is a mistake by the user.

There are also mixed cases of miscomputation that have more than one source. One type of mixed case is miscomputation due to the accumulation of round-off errors in the finite precision arithmetic that computer processors employ—it is due to the interaction between different kinds of software within the limitations of the hardware. Another mixed case is faulty hardware-software interaction. An example of this last type occurs when the execution of a program requires more memory than the computer has available. At least up to a few years ago, when no more memory was available, many personal computers used to "freeze" without being able to complete the computation.[19]

3.6 Taxonomy

The mechanistic account of computation explains why few systems qualify as computers properly so called: only genuine computers—in fact, only *some* computers—are programmable, stored-program, and computationally universal (Chapter 11). These properties of some computers are mechanistic properties, which the following chapters will explain mechanistically in terms of the relevant components, their functions, and their organization. Systems that have only a subset

[19] For an early discussion of several kinds of computing mistakes by computing mechanisms, see Goldstine and von Neumann 1946. For a modern treatment, see Patterson and Hennessy 1998. This section benefited greatly by reflecting on Fresco and Primiero 2013 and Dewhurst 2014. Much more remains to be said about miscomputation. For starters, the taxonomy I propose in the main text could be made more fine-grained by considering different layers of programming, distinguishing between programmers' intentions and the programs they write, and more.

of these capacities may also be called computers, but they can be distinguished from ordinary computers, and from one another, based on their specific mechanistic properties. Computing systems that lack all of these capacities deserve other names, such as 'calculators,' 'arithmetic-logic units,' etc.; they can still be differentiated from one another based on their computing power, which is determined by their functional organization. The burden of satisfying this desideratum will be discharged in Chapters 8–13.

Summing up, according to the mechanistic account of computation, a concrete computing system is a functional mechanism one of whose teleological functions is performing concrete computations. Concrete computations, in turn, are the processing of vehicles (by a functional mechanism) according to rules that are sensitive solely to differences between different portions (i.e., spatiotemporal parts) of the vehicles. The mechanistic account of computation has many appealing features. It allows us to formulate the question whether a mechanism computes as an empirical question, to be answered by a correct mechanistic explanation. It gives an account of miscomputation. It allows us to formulate a clear and useful taxonomy of computing mechanisms and compare their computing power. The full scope, power, vindication, and theoretical payoff of the mechanistic account can only be seen by examining some important classes of computing systems and some important issues in the philosophy of computation. This is the task of the remaining chapters.

8

Primitive Components of Computing Mechanisms

1. Analyzing Computing Mechanisms

Computing systems have been variously invoked to explain intelligence (e.g., Turing 1950), thought (e.g., Fodor 1975), and intentionality (e.g., Harman 1999), and to solve the mind-body problem (e.g., Putnam 1967b). Philosophers have also debated whether the internal states of computing systems have narrow or wide content (e.g., Burge 1986; Devitt 1990). Yet computing systems have been subjected to little philosophical analysis in their own right. Specifically, a negligible amount of philosophical attention has gone to the internal structure and functional organization of computing systems.

Previous chapters introduced several accounts of concrete computation and defended a mechanistic account: computing systems are mechanisms whose function is to manipulate medium-independent vehicles according to rules defined over the vehicles. This chapter and the following ones work through some important details about the structural and functional properties of computing systems. Their purpose is two-fold: to show that the mechanistic account explicates and taxonomizes computing systems more clearly and more fruitfully than other accounts (desideratum 6) and to exhibit the kind of theoretical payoff that the mechanistic account delivers. This chapter discusses the *primitive* components of computing mechanisms and their peculiar properties. The next chapter discusses *complex* components of computing mechanisms. After that, we'll have the ingredients to tackle entire computing systems and their properties.

The goal of this chapter and the next is to identify the main kinds of component that enter the mechanistic explanation of computing systems and to identify the functions of those components. The result is an account of the components of computing mechanisms that (i) matches the language and practices of computer scientists and engineers (e.g., see Patterson and Hennessy 1998), (ii) serves as basis for analyzing varieties of computers and their functional properties (Chapter 10–12), and (iii) can be used to clarify philosophical discussions that appeal to properties of computing systems.

I will focus primarily on digital computing mechanisms. According to the mechanistic account, digital computing mechanisms are systems whose function is to perform digital computations. Digital computations are transformations of input strings of digits into output strings of digits, transformations that accord to a general rule that is sensitive to the input strings, possibly sensitive to internal states, and applies to all strings. Strings of digits, in turn, are sequences of permutable digits identified by the digits' types, their number, and their order within the string. The particular kind of mechanistic explanation that applies to digital computing mechanisms explains how they can perform digital computations by analyzing digital computing mechanisms into specific kinds of component with specific kinds of function.

I'll start with the simplest components that can be attributed a computing function within a mechanism, i.e., components that can be attributed a computing function but whose components cannot be attributed a computing function. I call them *primitive computing* components. Then I'll discuss other kinds of components, whose function is something other than performing a computation. In the next chapter, I will discuss how primitive components can be put together to form more complex components, such as Boolean circuits, arithmetic logic units, etc.

2. Primitive Computing Components

Primitive computing components are mechanisms that perform computations contributing to what their containing mechanism computes, but whose components do not perform any computation relative to what primitive computing components compute. In other words, the computations performed by primitive computing components cannot be analyzed in terms of simpler computations. So their computations are primitive relative to the multi-level computing mechanism under investigation.

Primitive computing components belong to the class of input-output devices, i.e., devices whose function is to receive physical variables as inputs and yield physical variables as outputs in response to their inputs. Examples of input-output devices include washing machines, refrigerators, and mousetraps. But unlike typical input-output devices, the inputs and outputs of primitive computing components are not individuated by their specific physical properties (e.g., chemical composition, temperature, pressure, etc.) but only by their possession of appropriate degrees of freedom that the components can process in different ways, regardless of how such degrees of freedom are physically implemented. What matters is that the component can differentiate types of input and process them in different ways, so as to generate appropriate types of output given certain types of input.

Inputs and outputs of primitive *digital* computing components belong to finitely many types, each of which can be distinguished by the corresponding digital computing component and processed accordingly. If there are only finitely many types,

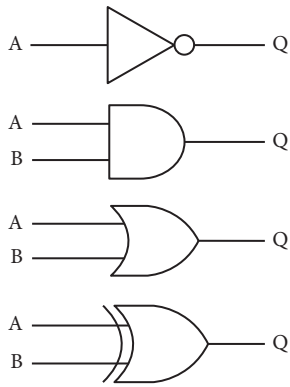


Figure 8.1 A NOT gate, an AND gate, an OR gate, and an XOR (exclusive OR) gate.

each type can be labeled by a letter from a finite alphabet, or symbol, without losing information about the function of the components. At any given time, primitive digital computing components take a finite number of digits as inputs and yield a finite number of digits as output. Typically, the output of primitive digital computing components is a deterministic function of the inputs of the component, but it can also be a stochastic function of them.

The primitive computing components of most contemporary computers are logic gates.[1] Logic gates are devices with two or three extremities. During any relevant time interval, a logic gate receives one or two input digits through one or two of its extremities and produces one output digit through another extremity. Logic gates' input and output digits belong to one of two types, usually called '1' and '0'. Digits of these types are normally called binary digits or *bits*. In analyzing and designing logic gates, '1' and '0' are respectively interpreted as truth and falsehood, so that functional operations on them can be interpreted as logical functions applied to truth values. Under the standard interpretation of logic gates' digits, the type of a logic gate's output digit corresponds to what would be generated by applying a logical connective, such as conjunction, disjunction, or negation, to the types of the input digit(s). This is why these devices are called *logic* gates. Accordingly, the input-output behavior of logic gates can be represented by truth tables or, equivalently, by logic equations written in Boolean algebra.

Figure 8.1 depicts four examples of logic gates. A NOT gate takes a digit (i.e., either a '0' or '1') as input and yields a digit that belongs to the other type (i.e., a '1' or a '0', respectively) as output. An AND gate takes two digits as input and yields a '1' if and only if two '1's were received as input, a '0' otherwise. An OR gate also takes two digits as input and yields a '1' as output if and only if it received at least one '1' as

[1] Logic gates should not be confused with logical connectives. Logical connectives are components of a logical system; logic gates are concrete components of a physical system.

input, a '0' otherwise. An XOR gate takes two digits as inputs and yields a '1' if and only if one of the two inputs (but not both) was a '1'.

Not all computing mechanisms are constructed out of logic gates. For example, analog computers contain physical devices that manipulate their inputs, which are not digits but continuous variables, such that the values of their outputs stand in a certain arithmetical relation to the values of their input (such as addition or multiplication), or they are the integral of the input, etc. When such physical devices receive certain input variables, they generate output variables whose relevant physical properties stand in relation to the same property of the inputs in a way that corresponds to a certain function of the input. These devices are primitive computing components too. As in the case of logic gates, the input-output behaviors of primitive computing components of analog computers can be composed together to execute the steps of a procedure that takes their operations as primitive, so that their combined work produces the computation of their containing mechanism.

Any physical realization of a primitive computing component is a particular mechanism made out of physical components, which have particular physical properties and interact in particular ways to generate the desired input-output behavior. For instance, the primitive computing components of most modern computing mechanisms are made out of electrical circuits, which are made out of wires, switches, resistors, and other components studied by electrical circuit theory. The same component, for example, an AND gate, can be made using different electrical circuits so long as they exhibit the relevant input-output properties. And the same electrical circuit can be made by using different physical materials.

The mechanistic explanation of a particular physical realization of a primitive computing component—say, an AND gate—explains how that component exhibits its specific input-output behavior. In our example, the components of a particular electrical circuit, their properties, and their configuration explains how it realizes an AND gate. This is still a mechanistic explanation, because it explains the behavior of the whole mechanism in terms of the functions performed by its components. But it is not a *computational* explanation, because its components are identified in terms of their specific physical properties (e.g., voltage levels) and not in terms of digits and operations on digits. Each physical realization of a primitive computing component that uses different physical components or a different design requires a specific mechanistic explanation that appeals to the specific physical properties of those components or that specific design.

What primitive computing components are made of and how they work makes a big difference to those who have to physically implement a given computational design. The many logic gates (as well as other components) that are used in building a particular computing mechanism must all be technologically compatible with each other. In implementing a design for a complex computing mechanism made out of logic gates, it makes no sense to use a certain device as a logic gate unless there are other devices (other putative logic gates) that it can be physically connected with. If

we take two devices that act as logic gates within two different machines and hook them up together, we don't reconstitute a computing mechanism unless the two devices physically interact in the right way. As I pointed out in the previous chapter, in building a computing mechanism, we must take into account a lot of details about the physics of the device. But almost all the physical stuff—except for the transformation of some computational vehicles into others—happens below the level of primitive computing components. And everything below the level of primitive computing components, such as the physical realizations of primitive computing components, can be abstracted away from computational explanation because it is irrelevant to the mechanistic explanation of computing mechanisms as such. It may be mechanistic explanation but not computational explanation.

Once we reach the level of primitive computing components from below, and assuming that the physical implementation is well designed, we can ignore lower levels and worry only about the input-output properties of primitive computing components. The behavior of primitive computing components is relevant to the mechanistic explanation of computing mechanisms only insofar as they generate certain outputs in response to certain inputs, where the inputs and outputs are defined independently of the implementing medium. These input-output behaviors, together with the way primitive computing components are interconnected, can then be used to explain the computations performed by larger components. Aside from their input-output behaviors, when we give computational explanations or design the logic circuits of computers, primitive computing components can be treated as black boxes. Since the mechanistic explanation of primitive computing components does not appeal to computations by their components, while at the same time primitive computing components' input-output behavior can be used as primitive computational operations to explain the computations performed by larger components, primitive computing components are the simplest components to be ascribed computations within a mechanism, and that's why I call them *primitive*.

The main reason for attributing computations to logic gates is that they can be composed to form more complex mechanisms that compute more complex functions, and the complex functions computed by the complex mechanisms can be deduced from those computed by their components together with the way the components are connected in combination with Boolean algebra or truth tables. This makes logic gates extremely useful and versatile in computer design, and it explains why they are usually ascribed computations even though what they do is computationally trivial. They compute because their function is to produce certain (medium-independent) outputs in response to certain (medium-independent) inputs. This function arises as part of the computational behavior of their containing mechanism, and the computational behavior of their containing mechanism can be analyzed in terms of their input-output behavior by means of Boolean algebra or some other computational formalism.

Since computers are built out of primitive computing components, the fact that primitive computing components can be built out of an indefinite number of different materials and designs is why computers can be built out of components with indefinitely many different physical properties.

This multiple realizability of primitive computing components may have tempted some people to see everything as a computer (e.g., Putnam 1967b; Searle 1992; and Churchland and Sejnowski 1992; see Chapter 4 for a discussion of pancomputationalism). But that temptation should be resisted: from the fact that many different materials can be used to build a primitive computing component, it doesn't follow that everything is a computer. This is because in order to constitute a primitive computing component, a physical system must still exhibit certain very specific and well-defined input-output properties, which can be exploited in designing and building computing mechanisms. And in order for a mechanism made out of primitive computing components to be a full-blown computer, those components must be functionally organized in very specific ways.

One of the burdens of this book is to show that for something to be a computing mechanism of nontrivial computing power, its components must be functionally organized in very specific ways so as to exhibit very specific properties. What is peculiar to computing mechanisms is that in order to understand their computations, they can be analyzed or designed (up to their primitive computing components) with complete disregard for their physical composition and specific physical properties except for those degrees of freedom that are needed for the computation and the ability to process them, using only principles of computer design and computability theory.[2]

3. Cummins on Primitive Computing Components

The only philosopher who has devoted significant attention to the primitive components of computing systems is Robert Cummins (1983). The present account of primitive computing components can be usefully contrasted with Cummins's account. Under both accounts, the input-output behaviors of primitive components are taken as primitive relative to the computing system they are part of, i.e., the input-output behaviors of primitive computing components are not analyzed in terms of simpler computations performed by them or their components. But there are some important differences between the two accounts.

First, Cummins says that primitive computing components "instantiate" the functions from their inputs to their outputs as a matter of physical law, without

[2] Real hardware designers, of course, design their machines while keeping in mind constraints coming from the technology that is being used to implement their design. This does not affect the point that, in principle, different technologies can be used to implement the same design or similar designs. For a historical example, see Cohen 2000.

specifying whether the law is strict or *ceteris paribus*. I add to Cummins's account that primitive computing components instantiate their input-output functions *ceteris paribus*, i.e., not because they fall under a strict law but because it is the *function* of the component to instantiate it. Primitive computing components, just like any other functional mechanisms, can malfunction, break, or be malformed or defective.[3] This point generalizes to non-computing components (see Section 4 of this chapter) and to anything built out of primitive computing components, for two reasons: (1) if a primitive computing component malfunctions, or breaks, or is malformed or defective, this may carry over into a malfunction, or break-down, or malformation or defect in the containing system—though this may not be the case if there is redundancy in the system; (2) the containing system is a functional mechanism too, so it has its own ways of malfunctioning, breaking, being malformed, or being defective. From now on, I will take this point for granted and won't repeat it for other kinds of components or mechanisms.

Second, for Cummins the input-output function instantiated by a primitive computing component is a step in the computation performed by the whole computing mechanism, whereas for me it is only part of the activity performed by the component that contains the primitive computing component. The activity of the containing system may or may not be a computation; if it is a computation, the input-output function instantiated by a primitive computing component may be a step in that computation but it may also be a part of a step (which, combined with other parts performed by other primitive computing components, constitutes a step).

In Cummins's account, computation is always the execution of a program, which is analyzed into primitive operations. These operations are the primitive steps and may all be performed by the same components. In my account, computation may or may not be the execution of a program. In addition, even in the case of computations that are executions of programs, before we get to see the role a primitive component plays in program execution, we have to understand the complex components made out of primitive computing components (next chapter). For now, all that a primitive component contributes to is its containing mechanism, which is not necessarily a program-executing mechanism. When we get to program execution in the next two chapters, we'll see that there is no need to assume that what a primitive computing component computes is an elementary step in the execution of a program.

Third, for Cummins it is inappropriate to say that a primitive computing component *computes* a function. We should say only that it *instantiates* a function (because for Cummins computation amounts to program execution, and a primitive computing component does not execute a program all by itself). This restriction is unjustified and unnecessary. Computer designers do say that logic gates compute, and so do other scientists, at least on occasion. If a "logic gate" is operating in isolation—not as

[3] Cummins's account of functions leaves little or no room for the notion of malfunction. I am relying on the account of functions I offered in Chapter 6.

part of a larger computing mechanism or even a non-computing functional mechanism—it may be somewhat debatable whether we should say that the "logic gate," all by itself, computes a function (e.g., that an AND gate computes whether both of its inputs are '1's). But in the context of a computing mechanism and of designing computing mechanisms, and even in the context of other functional mechanisms, it is appropriate and may be accurate to ascribe computations to primitive computing components. It is appropriate and accurate whenever a component fulfills the function of processing medium-independent vehicles according to an appropriate rule, as primitive computing components do when they are embedded in a functional mechanism. In the case of complex computing mechanisms, attributing computations to their primitive components is especially useful because it allows us to analyze the computations performed by the whole mechanism in terms of the computations performed by the primitive components using certain kinds of analytic tools, such as Boolean algebra or truth tables.

4. Primitive Non-computing Components

Not every component of a computing mechanism performs computations. We've already seen this in the case of the components of primitive computing components, which perform no computations relative to the computations of the system under analysis.

More generally, computing mechanisms contain many components whose function is something other than computing. Some of these non-computing components act as physical support for the other components, others provide energy to the mechanism, others generate and transmit the signals that keep the mechanism synchronized, yet others transmit inputs and outputs from one computing component to another, etc. Although none of these components perform any computation, they are necessary for the functioning of the whole mechanism. So, in judging whether a system is a certain kind of computing mechanism, which requires certain non-computing components for its proper functioning (such as components that keep the system synchronized), it may be useful to see whether it has those non-computing components.

In most computing mechanisms that can change their internal states, synchronization among components is obtained through the use of a *clock*. In modern computing mechanisms, a clock is a device that generates a signal with a fixed cycle time. A *cycle time* is a temporal interval divided into two subintervals, during which the clock signal takes, respectively, a high and a low value. A *fixed* cycle time is a cycle time of fixed length, which repeats itself. In computing mechanisms that have internal states, such as ordinary computers, the function of clocks is to determine how often, and at what times, the relevant components update their states. The main constraint of clocked computing mechanisms is that the clock cycle time must be

long enough to allow the signals determining state changes to stabilize on the values that are relevant to the state changes.

When a component receives an input signal that is not already synchronized with its internal clock, the signal must be synchronized with the component so that it can be properly processed. *Synchronizers* are components that take asynchronous signals and a clock's signal as input and yield a signal synchronous with the input clock as output.

Clocks and synchronizers are usually ignored in computability theory, where the synchronization of mathematically defined computations can be assumed to be perfect by theoretical fiat. But clocks and synchronizers are among the most important non-computing components of concrete computing mechanisms. A clock malfunction or a mistake in the design of the clock and related synchronization devices, such as a cycle time that's too short, can render a computing mechanism useless, because its components will not update their states at the time when their input signals have stabilized on the relevant values, or will update their states in the wrong order.

Another important function of certain non-computing components is to help isolate computing mechanisms from various aspects of their environment, which might interfere with the mechanisms' proper functioning. Like other functional mechanisms, computing mechanisms can be damaged, or their proper functioning can be disrupted, by an indefinite number of environmental factors, such as too much pressure, too high or too low temperature, electrical or magnetic fields, etc. What factors interfere depends on the specific physical and functional characteristics of a given mechanism. In order to protect computing mechanisms from the interference of relevant external factors, precautions (e.g., surrounding the mechanism by a protective shell) must be taken in designing and building them.

5. Conclusion

In summary, computing mechanisms are built out of two broad classes of primitive components: computing and non-computing components.

Primitive non-computing components serve a variety of functions that are necessary to perform computations: synchronizing the computing components, holding other components together, protecting mechanisms from environmental threats, etc.

Primitive computing components have the function of manipulating medium-independent vehicles—specifically, transforming input vehicles into output vehicles—in accordance with a general rule that applies to all vehicles; thus, their function is to perform computations. For example, the vehicles manipulated by primitive *digital* computing components are atomic strings—they consist of single digits. The computations of primitive computing components cannot be analyzed into simpler computations. Because of this, primitive computing components are

the simplest components that can be ascribed a computing function within a mechanism.

But primitive computing components can be combined with other primitive computing and non-computing components to form complex computing and non-computing components. In the next chapter, we'll look at how primitive components are put together into complex components. After that, we'll look at how complex components come together to form entire computing systems.

9

Complex Components of Computing Mechanisms

1. Complex Computing Components

Having introduced primitive computing components in the previous chapter, we can now see how primitive computing components can be combined together and with non-computing components to form complex computing components. This chapter analyzes in mechanistic terms complex computing components (e.g., arithmetic-logic units) and complex non-computing components (e.g., memory units) that are built out of simpler components and, ultimately, are analyzed in terms of primitive components. The most interesting computing components are processors, which have the function of performing operations on data. The most interesting processors are those whose function is executing instructions.

Instructions are strings of digits that have an *internal semantics*. As we saw in Chapter 7, an internal semantics assigns instructions a narrow content: roughly speaking, each instruction is interpreted as representing what the processor is going to do while executing it. An internal semantics should be distinguished from an external semantics, which interprets a representational vehicle as representing objects and properties external to the system. An external semantics ascribes wide content, that is, content that supervenes on more than the internal physical properties of a mechanism, whereas internal semantics ascribes narrow content, that is, content that supervenes solely on (some of) the intrinsic properties of the system. To flesh out the notion of internal semantics, I will specify what mechanistic properties of what components are relevant to assigning an internal semantics to instructions and vindicate the claim, made in Chapter 7, that having an internal semantics does not entail having any external semantics. This clarifies an important sense in which some computational states have narrow content, what the source of this narrow content is, and what the relationship between this kind of narrow content and wide content may be.

Besides the clarification of the notion of internal semantics, the account of complex components presented in this chapter has several virtues. First, it matches both our everyday understanding of computing mechanisms and the language and practices of computer scientists and engineers. Second, it can be used to classify

different kinds of computing mechanisms based on their computing power (desideratum 6), which in turn clarifies our terminology pertaining to computing mechanisms.

The important point is that the computation performed by a complex component can be exhaustively analyzed in terms of computations performed by its primitive computing components and manipulations performed by its non-computing components. So we can design a complex component by appealing only to the known functional properties of *its* components with respect to vehicle manipulation, without worrying about how the primitive computing components will work together with respect to their lower level properties. The details of the physical interactions between primitive components are abstracted away by relying on the assumption that those who build the mechanism use components that are physically compatible with one another and connect them in appropriate ways. Since all complex computing components are identified by their computational properties, the same point applies to complex computing components built out of simpler but non-primitive computing components.

The way the components of a computing mechanism are combined together to form the whole mechanism depends on the specific characteristics of the components. Typically, each component has appropriate extremities, which carry signals into and out of the component. The extremities of two components can be physically joined together, so that the signals coming out of one component enter the other component.

An important property of complex computing components is that their computational properties are composed, and therefore can be deduced from, the computational properties of *their* components together with their functional organization, without needing to take into account their physical composition or lower-level physical properties. I will focus on digital computing mechanisms.

2. Combinational Computing Components

Combinational computing components take a fixed finite number of input digits and yield a fixed finite number of output digits that depend only on their input (and *not* on internal states). Combinational computing components cannot assume different internal states, and thus they always yield the same output in response to the same input.

In modern computers, complex computing components are built by joining a number of logic gates at their extremities. Logic gates can be hooked up to one another to form combinational computing components called Boolean circuits (because they are combinations of logic gates, each of which computes a Boolean operation). Some sets of logic gates are called *universal*, because combinations of them are sufficient to design all Boolean circuits, i.e., circuits for all logic functions, i.e., functions expressible as truth tables or logic equations in Boolean algebra. For

example, the set {AND, NOT} is a universal set, and so is the set {OR, NOT}. This sense of universality should not be confused with the computational universality of universal Turing machines (see Appendix).

Figure 9.1 presents a simple Boolean circuit called a half adder, which is composed of two AND gates, one OR gate, and one NOT gate, connected together at their extremities. A half adder takes two digits, labeled A and B, as input and yields two digits, labeled Sum and Carry, as output. The function of this circuit is to generate an output that is naturally interpreted as the sum of the two input signals, each of which is now interpreted as representing the numbers 1 and 0. The Sum output is '1' if and only if either A or B is '1', but not both. The Carry output is '1' if and only if both A and B are '1'. Under this interpretation of the digits as representing numbers, this simple combination of logic gates can be seen as performing the arithmetical operation of two-bit addition. Due to the versatility of arithmetic, this is very convenient in designing complex computing components.

The circuit in Figure 9.1 takes only two digits (A and B) as input, which may be seen as two strings of one digit each. In designing nontrivial computing mechanisms, we need mechanisms that manipulate strings of more than one digit. For example, one way to add two strings of more than one bit each is to couple the individual bits of the two strings in the order in which they occur within the strings (the first two bits of the two strings, the second two bits, etc.), then add the individual pairs of bits while also adding any eventual carry out digits from each of the pairs of bits to the next couple of bits. In order to do this, the circuit in Figure 9.1 is insufficient, because it

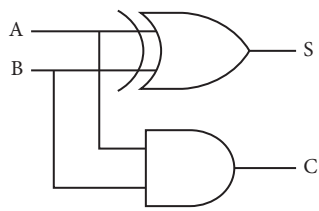


Figure 9.1 Half (two-bit) adder.

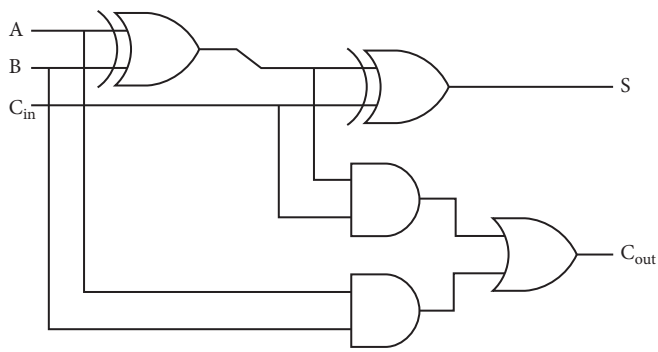


Figure 9.2 Full (two-bit) adder.

does not take a carry out signal as input. If we want to generate the sum of two strings that are longer than one digit each, we need the slightly more complex circuit depicted in Figure 9.2.

By combining two half adders like those of Figure 9.1 in the way depicted in Figure 9.2, we generate a full adder, which takes three digits—A, B and Carry in—as input and yields two digits—Sum and Carry out—as output. Any number of full adders can then be combined together, and the result is a circuit that generates a string of digits that is naturally interpreted as the sum of two strings of digits of fixed finite length.

When this is done, an ordering is introduced in the input and output digits. Until now, we dealt with operations on two single digits at a time. When we move to operations on strings of multiple digits, which digit is first in the string and how the digits are concatenated makes a difference to what output must be generated. For example, in the case of addition, the carry out from the first two bits in the string must be given as input to the full adder that adds the second two bits in the string, and so on until the end of the string. Therefore, circuits that manipulate strings of digits impose an implicit ordering on their input and output digits, so that there is a unique first digit of each string and a unique successor of each digit in the string. This ordering of digits to form strings is a crucial property of complex computing mechanisms, which is implicitly exploited in designing computing mechanisms that perform more complex computations.

Analogously to adders, one can design Boolean circuits that perform other logical or arithmetical operations, such as subtraction, identity check, conjunction, negation, etc., on two strings of digits of arbitrary (finite) length.

Boolean circuits are very convenient because given a desired input-output function constructed out of logical connectives (or equivalently, Boolean operations), there is an effective method for constructing a Boolean circuit that computes it. It is enough to replace each connective by the corresponding logic gate and connect the logic gates together in a way that corresponds to the nesting of the parentheses in the logic equation. Also, given a Boolean circuit, there is an effective method for constructing a truth table or a logic equation representing the logical function computed by the circuit. It is enough to represent each logic gate by the corresponding logical connective (or Boolean operation) and nest the logical formulae in a way that corresponds to the connections between the logic gates.

3. Arithmetic Logic Units

Boolean circuits can be combined to form more complex computing components. In modern digital computers, a particularly important complex computing component is a combination of Boolean circuits called an Arithmetic Logic Unit (ALU). An ALU operates on two data strings of fixed length, such as 16 or 32 bits, in combination with a few other input digits that act as control signals. Control signals determine

which operation the ALU performs on the two data strings. ALUs then yield one output string of fixed length, which is the result of the operation performed on the input strings. Like complex Boolean circuits, ALUs impose an ordering on their data and results, so that there is a unique first digit of each data string and result string, and a unique successor to each digit. The name 'arithmetic logic unit' comes from the fact that in modern computers, ALUs are designed so that their operations are naturally interpreted as either arithmetic operations (such as addition and subtraction) or logical operations (such as conjunction and disjunction) on their data.

ALUs are part of the processing core of modern computers, but computers do not *have* to be built around ALUs. Arithmetic and logical operations are used as primitives in modern computers because computer designers find them convenient to use in defining more complex operations. Other primitive operations could be used. Moreover, in explaining the behavior of an ALU we don't need to appeal to the fact that the functions it computes are arithmetic or logical; all we need to understand is how its components interact in the presence of certain inputs. ALUs and their operations are characterized as arithmetic or logical because this is useful in designing and programming the whole computing mechanism, just as the characterization of logic gates as logical is useful in designing complex computing components.

4. Sequential Computing Components

Sequential computing components are composed of combinational computing components and memory units. Memory units are described in more detail in Section 8. For now, suffice it to say that the presence of memory units allows sequential computing components to take a number of internal states. The output digits generated by sequential computing components depend on both the input digits the components receive and the content of their memory, that is, their internal state.

Since they use memory components, sequential computing components are more powerful than combinational computing components. Their computational behavior cannot be described by truth tables or logic equations, but it can be described by finite state automata (FSA). A FSA can be represented formally as a finite set of states plus two functions, called next-state function and output function. The set of states corresponds to all the possible contents of the memory of the component. The next-state function says, given the current state of the component and its input, what the next state will be. The output function says, given the current state of the component and its input, what the output of the component will be. The FSAs used in designing modern computers are synchronous, i.e., they update their internal state and generate their output once every clock cycle.

Just as there are effective methods for generating a circuit design given a logic equation or truth table and vice versa, there are effective methods for generating a circuit design given a FSA and vice versa.[1]

5. Multiplication and Division Components

In modern computers, multiplication and division are heavily used, so there are circuits that are hardwired to perform these operations on strings of fixed length (e.g., 32 bits). Multiplying finite strings of digits requires several additions or subtractions of strings while keeping track of both intermediate results and how many steps have been performed. Thus multiplication and division of finite strings of digits are more complex operations than addition and subtraction. They cannot be performed by standard ALUs, which have no means to store intermediate results and keep track of intermediate steps. Multiplication and division components are sequential computing components.

Multiplication and division components are simple FSAs made out of an ALU, a few memory registers to keep track of intermediate results, and a control mechanism that keeps track of the number of steps to be performed at any given time. But like ALUs, multiplication and division components take strings of fixed length as data and yield a fixed number of digits as result. They go through a cycle of internal operations that stops when the operation is completed, at which time they output their result and reset themselves to their initial state.[2]

6. The Computation Power of Complex Computing Components

The computation power of complex computing components is nontrivial and much more interesting than that of primitive computing components, but there are still interesting differences between different kinds of complex computing components. One way to understand the difference in computation power between different computing components is to distinguish between ordinary algorithms and what I call *finite-domain algorithms*.

An ordinary algorithm may define a total or partial function but, either way, it is defined over infinitely many inputs of arbitrary length such as strings of letters from a finite alphabet. In other words, an ordinary algorithm manipulates any input within an infinite domain and yields outputs for infinitely many inputs, even though there may be inputs for which it yields no output. Of course, physical systems have limitations of space and time that may prevent them from following ordinary,

[1] For details, see Patterson and Hennessy 1998, B38–9.

[2] For more details on the design of multiplication and division components, see Patterson and Hennessy 1998, 250–74.

infinite-domain algorithms for arbitrary inputs, but that is a separate point; I discuss those issues in Chapters 15 and 16.

By contrast, a finite-domain algorithm is defined over finitely many inputs of bounded length, and that's it! A finite-domain algorithm cannot manipulate inputs larger than a certain size because it's not defined over them, let alone infinitely many inputs. The distinction between infinite-domain algorithms and finite-domain algorithms is often underappreciated.[3]

Finite-domain algorithms can be further divided into *simple* ones, which require at most one operation from each component of a mechanism, and *complex* ones, which require more than one operation from at least one component of the mechanism. Complex finite-domain algorithms, such as finite-domain algorithms for 32-bit multiplication, require a mechanism to go through several cycles of activity before terminating and generating a result. This requires recurrent connections within the mechanism and a control structure that keeps track of the cycles of activity.

In simple Boolean circuits, the interest in their computations derives on one hand from the fact that the function computed by the circuit can be exhaustively analyzed in terms of the computations performed by its component logic gates using Boolean algebra or truth tables, and on the other hand from the fact that Boolean circuits can be put together to form ALUs and other complex computing components. There is no sense in which Boolean circuits can be said to execute a program or a genuine algorithm, because they operate on inputs of bounded length. But Boolean circuits can be said to follow simple finite-domain algorithms on their inputs.

ALUs are computing mechanisms capable of following a finite number of simple finite-domain algorithms, but they still have no power to either execute or follow genuine algorithms or even complex finite-domain algorithms. ALUs are computationally more powerful than ordinary Boolean circuits in that they can perform more than one operation on their data, depending on their control signal. Just as in the case of Boolean circuits, the simple finite-domain algorithms followed by an ALU are not represented by a program in the ALU but are hardwired in the functional organization of the ALU. That is why I say that they *follow*, rather than *execute*, their finite-domain algorithms.

Multiplication and division components, and other similar fixed-length sequential computing components, follow complex finite-domain algorithms, which require them to go through several cycles of activity before generating a result. In order to do this, they must have a control structure that keeps track of their cycles of activity and terminates the computation at the appropriate time. Because they require several

[3] For example, Cummins states that four-bit adders execute algorithms for addition (Cummins 1983, Appendix). This makes it sound like every computing mechanism, including a simple four-bit adder, has the same computation power; namely, the power to execute an algorithm. But four-bit adders do not literally compute the whole addition function; they compute at most four-bit addition. So they cannot possibly execute (genuine) algorithms for addition; at most they execute (or, to put it even better, follow) finite-domain algorithms for four-bit addition.

cycles of activity, multiplication and division are more complex operations than those computed by ALUs, and multiplication and division components are correspondingly more powerful components than ALUs. But multiplication and division components, like ALUs and Boolean circuits, are still limited to computing functions of data of bounded length.

None of these components have the power to execute or follow ordinary (finite-domain) algorithms, in the way that ordinary FSAs or Turing machines do. A fortiori, they don't execute or follow programs and are not computationally universal. To obtain computing mechanisms with these interesting properties, we need to reach higher levels of functional organization.

7. Complex Non-computing Components

Computing and non-computing components can be combined together to form complex non-computing components. For example, *decoders* are components that receive n input digits during a time interval and yield up to 2^n output digits during the next time interval, in such a way that only one of the output digits takes the value '1' for each of the possible input combinations while all the other output digits take the value '0'. Decoders are built out of logic gates and are useful in building larger computing and non-computing mechanisms (e.g., memory units), but their function is simply to turn a specific string of digits coming from a bundle of communication lines into a single digit with value '1' coming out of a specific line.[4] The reverse operation of decoders is performed by *encoders*.

8. Memory Units

A memory unit is an input-output component that takes and maintains one among a number of stable states, which may vary or be fixed depending on some of the memory's input digits, and can generate (when triggered by an appropriate signal) one or more output digits that reflect its internal state. After the state of a memory unit is updated, it remains the same until it is updated again. Because of this, a (digital) memory unit effectively stores one or more digits. Memory units employed in the design of modern computers are clocked, i.e., they update their state at the appropriate time during every clock cycle. This ensures that their state change is synchronized with the behavior of components that generate the memory units' inputs.

Logic gates can be combined together to form memory *cells*, i.e., memory units that store one digit. Memory cells can be combined together in arrays to form memory *registers*, which can store a string of multiple digits. Registers can be combined with

[4] For more details on the design of decoders and other relatively simple non-computing components out of logic gates, see Patterson and Hennessy 1998, B8–18.

other non-computing mechanisms (such as decoders) to form *register files* and RAMs (random access memories). These are large memory units arranged so that supplying a specific string of digits (called *address*) as input to the whole unit can be used to retrieve or update the content of a specific register.[5]

Memory cells and larger memory units don't compute by themselves, but they are necessary components of computing mechanisms that need to store data, programs, and intermediate results of computations.

The fact that memory units can be made out of logic gates shows that something that can be usefully seen as a computing component, such as a logic gate, can be used to build larger components whose function is different from computing, such as memory components. (More generally, logic gates have applications outside computer design.)

9. Datapaths

Memory components and other non-computing components can be added to ALUs and multiplication and division components to form *datapaths*, i.e., components that execute the operations that are primitive within a computer. Datapaths act on data strings of fixed (finite) length and generate the corresponding results of fixed (finite) length. Datapaths perform one out of a finite number of operations on their data depending on their control state.

We've seen that ALUs and other computing components, such as multiplication and division components, perform a finite number of operations on data strings of fixed length. For simplicity we'll only speak of ALU operations, but the same point applies to other complex computing components such as multiplication and division components. Which operation an ALU performs depends on control inputs it receives. So for any operation that an ALU can perform, there are strings of digits that—when given as input to the ALU—both determine which operation the ALU performs and specify the inputs on which it operates. Strings of digits that trigger operations on certain data strings by the ALU can be stored in memory registers and are a type of instruction called *arithmetic-logic instruction*. One function of the datapath is to retrieve arithmetic-logic instructions from the registers in which they are stored, convey them to the ALU, and put the result of the operation into an appropriate memory register. This process is a form of instruction *execution*.

After one instruction is executed, the datapath must be prepared to execute another instruction. Instructions are stored in memory registers, and the content of each memory register can be either written or retrieved by sending a certain string of digits as input to the memory. This is called *addressing* the memory. One function of the datapath is to keep track of which instruction is being executed at any given time

[5] For more details on the design of memory units, see Patterson and Hennessy 1998, B22–33.

by holding the string corresponding to the register that holds the instruction in an appropriate register, called *program counter*. A related function of the datapath is to determine which instruction comes next, which can be done by updating the content of the program counter in an appropriate way. This can be done by a combination of the following: storing instructions in consecutive memory registers, having a convention for addressing memory registers that uses consecutive strings for consecutive registers, and replacing the string in the program counter by its successor after every instruction execution.

Instead of having arithmetic-logic instructions specify the data for the arithmetic-logic operations, it is convenient to have those data stored in an appropriate register file within the datapath. The content of the register file can be altered by appropriate strings of digits, which specify which register must be altered and what must be put into it. An analogous string of digits can be used to take the content of a register within the datapath and copy it into a register in the main memory of the computing mechanism. These strings of digits, whose function is to store data in the main memory or load them into the datapath of the computing mechanism, are called *memory-reference instructions*. By using memory-reference instructions, data for computations can be moved back and forth between the datapath and the main memory independently of the instructions that specify which operations must be performed on them. One function of the datapath is to retrieve memory-reference instructions, retrieve the string of digits from the input register indicated by the instruction, and put the same string of digits into the output register indicated by the instruction.

As described so far, the datapath is confined to executing instructions in the order in which they are stored in memory, without ever going back to previously executed instructions or skipping some of the instructions. This limits the computation power of a computing mechanism.[6] To overcome this limitation, the datapath can be set up so that it can update the program counter in a way that depends on whether the content of a certain register satisfies some condition that can be checked by the ALU. If the datapath is set up this way, there will be strings of digits that determine whether the program counter should be updated by going to the next instruction or by jumping to an arbitrary instruction, and what instruction it should jump to. These strings are called *branch instructions*. One function of the datapath is to retrieve branch instructions from the memory registers where they are stored, use the ALU to determine whether the condition they indicate is satisfied, and update the program counter accordingly.

It should be clear by now that the datapath has many important functions, whose fulfillment in the appropriate order constitutes the computations performed by a computing mechanism capable of *executing* instructions. What the datapath needs in

[6] Specifically, such a computing mechanism is limited to computing primitive recursive functions. It cannot compute partial recursive functions that are not primitive recursive.

order to accomplish its various jobs is a mechanism that determines, for each type of instruction, which job the datapath must accomplish. One way to determine which type each instruction belongs to is to include within each instruction a sub-string, in a conventional fixed place (e.g., the beginning of the whole string), and to use different sub-strings for different instructions types. To react appropriately to that instruction sub-string is the function of the control unit.

10. Control Units

A control unit receives as input the part of an instruction that determines its instruction type and outputs a signal that sets up the datapath to do the job corresponding to that instruction type. When seen in isolation, a control unit can be described as a computing component, such as a Boolean circuit or a simple FSA with inputs and outputs of fixed size. But the control unit's contribution to its containing mechanism is a *control* function. A control unit has the function of setting up the datapath to perform the kind of operation that corresponds to each type of instruction when an instruction is executed.

Depending on the design and timing methodology, a control unit may be a combinational or a sequential computing component. For instance, if all the events within the datapath take place during one clock cycle, then the control unit can be a combinational component. But if different events within the datapath take place during distinct clock cycles, then the control must keep track of which event is taking place at any given time and which event comes next, so that it can send the appropriate signals to the datapath. In order to do this, the control unit must have a register that keeps track of the different stages of the execution, so it must be a sequential component.

A crucial function of the control unit is to deal with at least some kinds of events that may result in a mistake in the computation. An event of this type is the presence of an instruction whose execution requires more memory than is available.[7]

Together, the control unit and the datapath constitute the *processor*, which is the core computing component of modern computers. In modern computers, the processor is the component that actually carries out the computers' computations.

Just as in the case of ALUs and other complex computing components, the inputs and outputs of memory units and processors are strings of concatenated digits, in which each digit occupies a well-defined position. Different parts of the string may have a different functional significance depending on where they are located along the string. For example, the first part of a string may determine the instruction type, whereas other parts may determine which registers must be accessed for retrieving data. This ordering of strings is accomplished within the mechanism by the structure

[7] For more details about how control units avoid certain kinds of miscomputations, see Patterson and Hennessy 1998, 410–16.

of components together with the appropriate ordering of the communication lines between the components. Because of this, an observer can tell which digit is first in a string, which is its successor, and so on until the end of the string, so that each string is unambiguously identified. Without this ordering of the digits to form strings, complex computing and non-computing mechanisms would not be able to function.

11. Input and Output Devices

An input device is a mechanism through which a computing mechanism receives inputs coming from the environment external to it. Input to digital computers must be concatenated in appropriate ways, so that the computer can respond differentially to the first digit in a string and to the successor of each digit in a string. This ordering of the input digits is guaranteed by the input devices and is then preserved throughout the computing mechanism. In ordinary digital computers, examples of input devices include keyboards, mice, and scanners.

An output device is a mechanism that conveys the results generated by a computing mechanism, or outputs, to the mechanism's environment. Output from digital computers are also ordered, so that the receiver of an output from a digital computer can (in principle) tell which digit starts a string and which digit follows each other digit, until the end of the output string is reached. This ordinal arrangement of outputs is guaranteed by the functional properties of the output device. In ordinary digital computers, examples of output devices include monitors and printers.

12. Internal Semantics

We've seen that in designing and analyzing digital computing mechanisms, the operations computed by logic gates and Boolean circuits are naturally interpreted as logical operations, and the operations computed by arithmetic-logic units are naturally interpreted as arithmetic and logical operations. This natural interpretation is an *external* semantics, relating the inputs and outputs of these computing mechanisms to something external to the mechanism. External semantics is not necessary to individuate these components and the computations they perform because, as I argued in Chapter 7, they are fully individuated as computations defined over strings. But external semantics is useful in designing complex components out of simpler components or in analyzing the computations of complex components into those of their simpler constituents.

In discussing the operations of ordinary processors, the data, the intermediate results, and the final results of the computation are naturally interpreted as referring to numbers. The same considerations apply: this external semantics is not necessary to identify the strings of digits and the operations performed on them. Everything that processors do could be characterized in terms of operations on strings, without saying anything about what the strings represent.

Not all strings manipulated by a processor are data, intermediate results, or final results. Many are instructions or parts of instructions. It is natural and useful, in designing and programming a computer, to interpret instructions as representing what a processor is going to do in executing them. This interpretation does not assign an external semantics, because it does not relate strings to objects external to the machine. I call it *internal semantics*.

A processor shifts instructions and data between the main memory of the computing mechanism and its internal registers, shifts data between its internal registers and its datapath, and controls its datapath so that it performs certain operations on the data. All of this is possible because of how the instructions are written and how the processor is functionally organized to respond to various parts of the instructions.

Part of each instruction is fed to the control unit, and the way it affects the control unit determines which operation the processor performs. This part of the instruction is naturally interpreted as representing a type of operation (e.g., addition, subtraction, loading, writing, branching, etc.). Depending on the instruction, parts of the instruction may be fed to the datapath's register file, where they activate different registers depending on their value. These parts of the instruction are naturally interpreted as containing *addresses* of the registers within the datapath, i.e., as naming the registers. Depending on the instruction, parts of an instruction may be fed to registers of the main memory, where they activate different registers depending on their value. These parts of the instruction are naturally interpreted as naming the registers within the main memory. Finally, part of an instruction may be fed to the datapath as input to an operation. This part is naturally interpreted as containing data.

Not all instructions have all of the above-mentioned parts, although in modern digital computers, all of them have the first part. Moreover, each (machine-language) instruction part of a given type must have a specific and fixed length and be placed in a fixed position within the instruction, so that the processor can react appropriately to each part of the instruction. Only when placed in an appropriate position within the rest of the instruction does a string of digits acquire its internal semantics. The same string of digits, placed in different parts of an instruction, or in the same position within different instructions, will represent different kinds of things (e.g., a memory address, a datum, or an operation). In this sense, internal semantics is not intrinsic to the "syntactic" type of the strings (that is, which digits constitute the string), but rather it is context sensitive.[8]

Because of how their parts are segmented and manipulated by the processor, arithmetic-logic instructions can be interpreted as telling which registers' contents must be manipulated in which ways, and which registers the results must be put in. Memory-reference instructions are naturally interpreted as telling which registers' contents must be shifted into which other registers. Branch instructions are naturally

[8] This is not only true for instruction parts. The same string of digits may constitute an instruction or a datum depending on which part of the memory it is placed in.

interpreted as telling how the program counter must be updated under which circumstances. Designers and programmers use this natural, internal semantics of instructions to interpret and write instructions and programs. Under this internal semantics, instructions and programs constitute a *code*, which is commonly referred to as the computer's *machine language*.

13. Conclusion

Complex non-computing components serve a variety of functions that are necessary to perform complex computations: taking inputs from the environment, holding internal states constant over time, determining which operation needs to be performed at a given time, delivering outputs to the environment, etc.

Complex computing components have the function of performing complex computations. Unlike the strings manipulated by primitive computing components, those manipulated by complex computing components are not atomic but have varying degrees of complexity. In addition, the rules for manipulating non-atomic strings have varying degrees of complexity, which require components of varying degrees of complexity to implement. As a result, complex computing components can be classified based on their computing power.

By combining computing and non-computing components in appropriate ways, we can construct various kinds of computing mechanisms, such as calculators and computers. Conversely, various kinds of computing mechanisms can be mechanistically explained in terms of the components described in this chapter. I offer such an analysis in the following three chapters.

10

Digital Calculators

1. Introduction

In the previous two chapters, I introduced both the primitive components of computing mechanisms and the main ways they can be put together to form complex components. We are now ready to see how all these components can be organized to form complete computing systems, which are computationally more powerful than any of their components taken in isolation. We begin with digital calculators in this chapter. In the next chapter we will look at digital computers and see why they are more powerful than calculators. After that we'll look at analog computers and neural networks.

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 machines*. Among computing machines, we single out a special class and call them *digital computers*. At the very least, digital computers are special because they are more versatile than other computing machines—or any other machine, for that matter. Digital 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. Digital computer versatility calls for an explanation, which I will sketch in the next chapter.

In contrast with the above intuitive picture, some authors suggest that there is nothing distinctive about digital 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 states of some other system, and the properties of the system support such an interpretation. . . . Conceivably, sieves and threshing machines could be construed as computers if anyone has reason to care about the specific function reflected in their input-output behavior (Churchland and Sejnowski 1992, 65–6).

If these authors are correct, then the distinctions between (i) systems that compute and systems that don't and (ii) digital computers and other computing machines, are

ill-conceived. For according to these authors, there is no fact of the matter whether something is a digital computer, some other computing machine, or something that performs no computations at all. Whether anything is said to be a computer, for these authors, depends merely on how we look at it.

This is an example of what in Chapter 4 I called interpretivist pancomputationalism: every physical system is a digital computer as long as someone interprets it as such. In the absence of a viable account of what is distinctive about digital computers, interpretivist pancomputationalism is tempting. But interpretivist pancomputationalists have a lot to explain away. They should explain away our intuition that there is something special about what we normally call *computers* and our consequent practice of applying the term 'computer' only to some machines—such as our desktop and laptop machines—and not others. They should explain why we think that the invention of computers was a major intellectual breakthrough and there are special sciences—computer science and computer engineering—devoted to studying the specific properties of computers. Finally, they should explain why the systems we ordinarily call computers, but not other computing machines that pre-date them, inspired the hypothesis that minds or brains are computers (von Neumann 1958; Fodor 1975).

Explaining all of this away is hopeless, for three reasons. First, as I argued in Chapter 4, pancomputationalism is primarily due to confusing computational modeling and computational explanation. Second, as I argued in Chapter 7, computing systems *in general* are a special class of physical systems: they are physical systems whose function is manipulating medium-independent vehicles according to appropriate rules. Third, there is a fact of the matter that is specific to digital computers and distinguishes them from other computing systems:

Digital Computer: A digital computing system of "large capacity."[1]

Digital Calculator: A digital computing system whose function is to perform a few computational operations on strings of digits of bounded but nontrivial size.

This chapter defends the second thesis by explicating digital calculators. The following chapter will defend the first thesis by explicating digital computers and the relevant notion of large capacity. Needless to say, the boundary between digital computers and digital calculators is a fuzzy one, and that's ok.

2. Digital Computing Mechanisms

As we saw in Chapter 7, computing mechanisms' capacities are explained in terms of their components, their components' functions, and the way their components and functions are organized. Digital computing mechanisms are distinguished from other

[1] I borrow the phrase 'large capacity' from John V. Atanasoff, who was among the first people to use the term 'computer' for a kind of machine (Atanasoff 1940, 1984).

mechanisms, including non-digital computing mechanisms, in that they perform a specific kind of process: digital computations. A *digital computation*, in turn, is the generation of output strings of digits from input strings of digits in accordance with a general rule that depends on the properties of the strings and (possibly) the internal state of the system. Finally, a *string of digits* is an ordered sequence of discrete elements of finitely many types, where each type is individuated by the type of effect it has on the mechanism that manipulates the strings.

Although strings of digits can be and usually are assigned semantic interpretations, the present notion of string is the one employed in computability theory and computer science, where semantic properties are not part of the identity conditions of strings (Chapter 3). This is a mechanistic, non-semantic notion of string. It is explicated in terms of how each type of digit, together with its position within a string, affects the computing components of a mechanism. Thus, this notion of string does not depend on semantic properties.

In analyzing calculators and computers, I will build on the account of computing and non-computing components presented in the previous chapter. Accordingly, I will distinguish between three classes of strings on the basis of the function they fulfill: (1) *data*, whose function is to be manipulated during a computation; (2) *results*, whose function is to be the final string of a computation; and (3) *instructions*, whose function is to cause appropriate computing mechanisms to perform specific operations on the data. Lists of instructions are called *programs*. I will also appeal to four kinds of large-scale components: (1) *processing units*, whose function is to execute any of a finite number of primitive operations on data; (2) *memory units*, whose function is to store data, intermediate results, final results, and possibly instructions; (3) *input devices*, whose function is to receive strings of digits from the environment and deliver them to memory units, and (4) *output devices*, whose function is to take strings of digits from memory units and deliver them to the external environment. Some processing units can be further analyzed into datapaths, whose function is to perform operations on data, and control units, whose function is to set up datapaths to perform the operation specified by any given instruction.

3. Calculators

Sometimes, the terms 'calculator' and 'computer' are used interchangeably. Following the more common practice, I reserve 'computer' for systems that are computationally more powerful than calculators. Given this restricted usage, a good way to introduce digital computers (in the next chapter) is to contrast them with their close relatives, ordinary digital calculators.

Few people use self-standing calculators anymore because they've been replaced by the *calculator function* on computers, tablets (which are portable computers that use an LCD screen as input device) and smartphones (which are small portable computers with a cellular phone function). But calculators used to be their own devices

and were ubiquitous until a few years ago. If you don't know what a self-standing calculator is, ask your parents.

A digital calculator is a computing machine made out of four main kinds of appropriately connected digital components: input devices, output devices, memory units, and processing units. I only discuss non-programmable calculators. So-called "programmable calculators," from a mechanistic perspective, are special purpose digital computers with small memory, and are subsumed within the next chapter.[2]

Input devices of digital calculators receive two kinds of input from the environment: (i) data and (ii) a command that causes the processing unit to perform a certain operation on the data. Commands are usually inserted in calculators by pressing appropriate buttons. An operation is a transformation of data into results. Calculators' memory units hold the data, and possibly intermediate results, until the operation is performed. Calculators' processing units perform one operation on the data. Which operation is performed depends only on which command was inserted through the input device. After the operation is performed, the output devices of a calculator return the results to the environment. The results returned by calculators are computable (recursive) functions of the data. Since digital calculators manipulate strings of digits according to a general rule defined over the strings, they are genuine digital computing mechanisms. The computing power of ordinary calculators is limited, however, by three important factors.

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

Second, an ordinary calculator performs only one operation on its data, after which it outputs the result and stops. It has no provision for performing several of its primitive operations in a specified order, so as to follow an algorithm automatically. Of course, each primitive operation of a calculator is performed by following a finite-domain algorithm (that is, an algorithm defined over finitely many inputs of bounded length). But a calculator—unlike a computer—cannot follow an ordinary algorithm or a finite-domain algorithm defined in terms of its primitive operations. In other words, a calculator has no control structure besides the insertion of

[2] For more details on programmable calculators, including a statement that they are a kind of computer, see Engelsohn 1978.

commands through the input device.[3] The operations that a calculator can compute on its data can be combined in sequences, but only by inserting successive commands through the input device after each operation is performed.

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

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

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

[3] A consequence of this is the often-remarked fact that calculators cannot perform branches; namely, they cannot choose among different operations depending on whether some condition obtains. (Branching is necessary to compute all computable functions.)

11

Digital Computers

1. Digital Computing Systems of Large Capacity

In the previous chapter, I argued that calculators are digital computing systems whose function is to perform a few computational operations on strings of digits of bounded but nontrivial size. In this chapter, I argue that digital computers are digital computing systems of large capacity. Computers' capacity is larger than the capacity of ordinary calculators in a number of ways that I'll articulate. Along the way, I develop a systematic taxonomy of computers based on their mechanistic properties, including hard-wired vs. programmable, general-purpose vs. special-purpose, analog vs. digital (Chapter 12), and serial vs. parallel (Chapter 13), 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 important theoretical issues in the history and philosophy of computing as well as the philosophy of cognitive science.

Digital computers are of philosophical interest for at least three reasons. First, digital computers are sufficiently important, both practically and conceptually, that understanding what they are is valuable in its own right. Second, a proper distinction between computers and other mechanisms and between different classes of computers is part of the foundations of computer science. In computer science there is no universally accepted notion of computer, and there have been controversies over whether certain machines count as computers of one kind or another. Some of those controversies have significant legal consequences (cf. Burks 2002). Resolving those controversies requires clear and cogent criteria for what counts as a computer of any significant kind. I provide such criteria in Sections 1–5. In Section 6, I will give an example of how my account illuminates a historical controversy. Third, as I will illustrate in Sections 7 and 8, a robust notion of digital computer gives substance to theories according to which the brain is a digital computer.

Like ordinary calculators, digital computers are made out of four main types of components: input devices, output devices, memory units, and processing units. The processing units of modern computers are called *processors* and can be analyzed as a combination of datapaths and control units (Chapter 9). A schematic representation of the functional organization of a modern computer is shown in Figure 11.1.

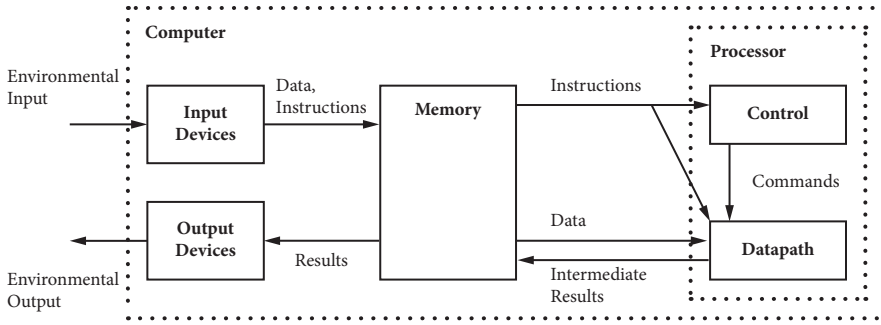


Figure 11.1 The main components of a computer and their functional relations.

The difference between (ordinary) calculators and digital computers lies in their mechanistic properties and organization. Computers' processors are capable of branching behavior and can be set up to perform any number of their primitive operations in any order (until they run out of memory or time). Computers' memory units are orders of magnitude larger than those of ordinary calculators, and often they can be increased in a modular 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 a relatively large size, whose bounds are not well-defined. Because of these characteristics, today's computers are called 'programmable', 'stored-program', and computationally 'universal'. (These terms are defined more explicitly in the next sections.)

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 machines 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 systems based on their relevant functional properties.

Until at least the 1940s, the term 'computer' was often 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 machine 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. Among computers, there is wide variation in how many operations their control unit can string together in a sequence and how complex an algorithm their control unit can follow (and consequently, how complex a problem a computer can solve). For instance, some machines that were built in the first half of the 20th century—such as the IBM 601—could string together a handful of arithmetical operations. They were barely more powerful than ordinary calculators, and a computing human could easily do anything that they did. Other machines—such as the Atanasoff-Berry Computer (ABC)—could perform long sequences of operations on their data; a computing human could not solve the problems that they solved without taking a prohibitively long amount of time.

The ABC, which was completed in 1942 and was designed to solve systems of up to 29 linear algebraic equations in 29 unknowns, appears to be the first machine that was called 'computer' by its inventor.[1] So, a good place to draw the boundary between calculators and computers might lie somewhere between machines that can be replaced by computing humans, such as ordinary calculators and the IBM 601, and machines that outperform computing humans at solving at least some problems, such as the ABC.[2] The exact boundary is best left vague. What matters is not how many machines we honor with the term 'computer,' but that we identify mechanistic properties that make a difference in computing power, and that whether a machine possesses any of these properties is a matter of fact, not interpretation. We've seen that one of those properties is the capacity to follow an algorithm defined in terms of the primitive operations that a machine's processing unit(s) can perform. Other important mechanistic properties of computers are discussed in the rest of this chapter.

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

2. Programmability

A computing human can do more than follow *one* algorithm to solve a problem. She can follow *any* (classical, i.e., non-quantum) algorithm, which is typically given to her

[1] Atanasoff 1940. On the ABC, see also Burks and Burks 1988; Burks 2002; and Gustafson 2000.
[2] A similar proposal is made by Burks and Burks 1988, chap. 5.

in the form of instructions, and thus she can compute any function for which she has an algorithm. More generally, a human being can be instructed to perform the same activity (e.g., knitting or playing the piano) in many different ways. Any machine that can be easily modified to yield different output patterns may be called 'programmable'. In other words, 'being programmable' means being modifiable so as to perform relatively long sequences of different operations in a different way depending on the modification.

Digital computers are not the only programmable mechanisms. This is especially relevant in light of the persistent tendency by some philosophers to identify computation with program execution (e.g., Cummins 1989, 91–2; Roth 2005, 458). On the contrary, some types of non-computing mechanisms, such as certain looms and music boxes, execute programs too. Computers' characteristic activity is computing, so a programmable digital computer is a digital computer that can be modified to compute in different ways. Furthermore, as will soon be clear, the kind of programmability that comes with the ability to execute computer programs is only one species of computer programmability among others, just as programmable computation is only one species of computation among others.

The simplest kind of programmability involves the performance of specified sequences of operations on an input, without the characteristics of the input making any difference on what operations must be performed. For instance, (*ceteris paribus*) a loom programmed to weave a certain pattern will weave that pattern regardless of what kinds of thread it is weaving. The properties of the threads make no difference to the pattern being woven. In other words, the weaving process is insensitive to the properties of the input. (Though, of course, whether the pattern is easy to observe and how it appears will depend on whether relevant threads have different colors, and what colors they have.)

This kind of programmability is insufficient for computing, because computing is sensitive to relevant differences in input. One reason for this is that a mathematical function typically yields different values given different arguments. Any mechanism that is computing that function must respond differentially to the different input arguments so as to generate the correct output values. Digital computing mechanisms do this by responding differentially to the different types of digit that make up the input and to the order in which they are concatenated into strings. Accordingly, programming a digital computer requires specifying how it must respond to different strings of digits by specifying how it must respond to different types of digit and different positions within a string.[3]

What counts as a legitimate modification for the purpose of programming a computer depends on the context of use and the means that are available to make

[3] The present distinction between programmability insensitive to the input and programmability sensitive to the input was partially inspired by a somewhat similar distinction, made by Brennecke 2000, 62–4, between executing a fixed sequence and using the output as input.

the modification. In principle, we could modify a calculator by taking it apart and rewiring its circuits or adding new parts, and then it would compute new functions. But this kind of modification would ordinarily be described as building a new machine rather than programming the old one. So we should relativize the notion of programmability to the way a machine is ordinarily used. This will not make a difference in the end, especially since the kind of programmability that matters for most purposes is soft-programmability (see below), where what counts as a relevant modification is determined unambiguously by the functional organization of the computer.

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

2.1 Hard programmability

Some early computers—including the celebrated ENIAC (Van der Spiegel et al. 2000)—had switches, plug-ins, and wires with plugs, which sent signals to and from their computing components. Flipping the switches or plugging the wires in different configurations had the effect of connecting the computing components of the machine in different ways, to the effect that the machine would perform different series of operations. I call any computer whose modification involves the mechanical modification of its functional organization *hard-programmable*. Hard programming requires that users change the way the components of a computer are spatially joined together, which changes the number of operations that are performed or the order in which they are performed, so that different functions are computed as a result. Hard programming is relatively slow and cumbersome. Programming is easier and faster if a machine is soft-programmable.

2.2 Soft programmability

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

Soft programmability allows us to use programs of quasi-unbounded size, which are given to the machine as part of its input or internal states. Since the whole program is written down as a list of instructions, in principle a soft-programmable machine is not bound to execute the instructions in the order in which they are written down in the program; it can execute arbitrary instructions in the list at any particular time. This introduces the possibility of conditional branch instructions, which require the machine to jump from one instruction to an arbitrary

instruction in the program based on whether a certain condition—which can be checked by the processing unit(s)—obtains. Conditional branch instructions are the most useful and versatile way of using intermediate results of a computation to influence control, which in turn is a necessary condition for computing all computable functions (cf. Brennecke 2000, 64–5). Since soft-programmable computers can execute programs of unbounded size and use intermediate results to influence control, they can be computationally universal (see Section 4).

There are two kinds of soft programmability: external and internal. A machine that is *externally* soft-programmable requires the programs to be inserted into the machine through an input device, and it does not have components that copy and store the program inside the machine. For example, universal Turing machines and some early punched cards computers—such as the famous Harvard Mark I (Cohen 1999)—are externally soft-programmable.

A machine that is *internally* soft-programmable contains components whose function is to copy and store programs inside the machine and to supply instructions to the machine's processing units. Internal soft programmability is by far the most flexible and efficient form of programmability. It even allows the computer to modify its own instructions based on its own processes, which introduces a final and most sophisticated level of computational control (cf. Brennecke 2000, 66).

Given the ability to modify their instructions, internally soft-programmable computers have a special, seldom-appreciated property. In principle, a soft-programmable computer may be able to increase its computational power by operations that are not computable by a Turing machine.[4] If an internally soft-programmable computer has the ability to modify its program(s) in a non-computable way (e.g., at random), then in principle it may modify its program so as to compute an increasingly larger number of functions. This increase in computational power need not be governed by any algorithm.

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

3. Stored-Program Computers

A stored-program computer has an internal memory that can store strings of digits. Programs are stored in memory as lists of instructions placed in appropriate memory

[4] Alan Turing is one of the few people who discuss this feature of internally soft-programmable computers; he uses it in his reply to the mathematical objection to the view that machines can think. For an extended discussion, see Piccinini 2003a.

registers. A stored-program computer also contains at least one processor. The processor contains a tracking mechanism (program counter) that allows the processor to retrieve instructions from a program in the appropriate order. The processor can extract strings of digits from memory and copy them into its internal registers (if they are data) or execute them on the data (if they are instructions). The same string can act as data or as an instruction in different occasions, depending on where it is located within a memory register and how the machine is functionally organized. After an instruction is executed, the tracking mechanism allows the control unit to retrieve the next instruction until all the relevant instructions are executed and the computation is completed.

A stored-program machine need not be programmable. Some kinds of memory unit are read-only, namely, they can communicate their content to other components but their content cannot change. Other kinds of memory unit are such that digits can be inserted into them. This opens up the possibility that a program be inserted from the outside of the computer (or from the inside, i.e., from the processor) into a memory unit, allowing a computer to be (soft) programmed and the existing programs to be modified. Once the program is in the appropriate part of the memory, the computer will execute that program on the data. A programmable stored-program computer can even be set up to modify its own program, changing what it computes over time. Stored-program computers are usually designed to be soft-programmable and to execute any list of instructions, including branch instructions, up to their memory limitations; if so, they are computationally universal (see Section 4). For this reason, the term 'stored-program computer' is often used as a synonym of 'universal computer' even though, strictly speaking, a stored-program computer may or may not be universal.

The idea to encode instructions as sequences of digits in the same form as the data and the idea of storing instructions inside the computer are the core of the notion of programmable, stored-program computer, which is perhaps the most fundamental aspect of modern artificial computing.[5] At this point, it should be clear that not every computing mechanism is a computer, and not every computer is a programmable, stored-program computer. In order to have a programmable, stored-program computer, we need to have a system of digits organized into strings, a scheme for encoding instructions, a re-writable memory to store the instructions, and a processor that is appropriately organized to execute those instructions. Several technical problems need to be solved, such as how to retrieve instructions from memory in the right order and how to handle malformed instructions. I have briefly described these properties of computers, the problems that need to be solved to design them, and the solutions to those problems, in terms of the components and their functional organization—that is, in mechanistic terms. I will soon describe other properties of

[5] In their authoritative textbook on computer organization, Patterson and Hennessy list these two ideas as the essence of modern computers (Patterson and Hennessy 1998, 121).

computers using the same strategy. This shows how the mechanistic account sheds light on the properties of computers.

4. Special-Purpose, General-Purpose, or Universal

Many old computers, which were not stored-program, were designed with specific applications in mind, such as solving certain classes of equations. Some of these machines, like the ABC, were hardwired to follow a specific algorithm. They are called *special-purpose* computers to distinguish them from their general-purpose successors. Special-purpose computers are still used for a number of applications; for instance, many computers installed in automobiles are special-purpose.

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

Assuming the Church-Turing thesis, namely the thesis that every effectively computable function is computable by a Turing machine, Alan Turing (1936–7) showed how to design universal computing mechanisms, i.e., computing mechanisms that would take any appropriately encoded program as part of their input and respond to that program so as to compute *any* computable function (see Appendix). Notice that Turing's universal computing mechanisms—universal Turing machines—are not stored-program (see Section 6 below for a defense of this claim).

We have seen that soft programmable computers can respond to programs of unbounded length and manipulate their inputs (of finite but unbounded length) according to the instructions encoded in the program. This does not immediately turn them into universal computers, because, for example, they may not have the ability to handle branch-instructions.[8] But no one builds computers like that. Ordinary soft-programmable computers, designed to execute all the relevant kinds of instruction, are *universal* computing mechanisms. For example, ordinary computers like our desktop and laptop machines are universal in this sense. Even Charles Babbage's legendary analytical engine, with minor modifications, would have been universal.[9] Universal computers can compute any Turing-computable function until they run out of time or memory. Because of this, the expressions 'general-purpose

[6] For an overview of early computers, see Rojas and Hashagen 2000.

[7] For a valuable attempt at distinguishing between degrees to which a computer is general-purpose, see Bromley 1983.

[8] Strictly speaking, branching is not necessary for computational universality, but the alternative is too impractical to be relevant (Rojas 1998).

[9] Thanks to Doron Swade for this point.

computer' and 'all-purpose computer' are sometimes loosely used as synonyms of 'universal computer'.

5. Functional Hierarchies

Above, I briefly described the functional organization that allows soft-programmable computers to perform a finite number of primitive operations in response to a finite number of the corresponding kinds of instruction. Computers with this ability are computationally universal up to their memory and time limitations. In order to get them to execute any given program operating on any given notation, all that is needed is to encode the given notation and program using the instructions of the computer's machine language.

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

When a processor executes instructions, memory registers for instructions and data are functionally identified by the addresses of the memory registers that contain the data or instructions. Each memory register has a fixed storage capacity, which depends on the number of memory cells it contains. *Virtual memory* is a way to functionally identify data and instructions by virtual addresses, which are independent of the physical location of the data and instructions, so that a programmer or user need not keep track of the physical location of the data and instructions. During the execution phase, the physical addresses of the relevant memory registers are automatically generated by the compiler on the basis of the virtual addresses. Since virtual memory is identified independently of its physical location, in principle it has unlimited storage capacity, although in practice the total number of physical digits that a computer can store is limited by the size of its physical memory.[10]

In analyzing how a processor executes instructions, all data and instructions must be digital strings corresponding to the physical signals traveling through the processor, and the functional significance of these strings is determined by their location

[10] For more details on virtual memory, including its many advantages, see Patterson and Hennessy 1998, 579–602.

within an instruction or data string. Moreover, all data and instruction strings have a fixed length, which corresponds to the length of the memory registers that contain them. *Complex notations*, instead, can contain any characters from any finite alphabet, such as the English alphabet. Programmers and users can use complex notations to form data structures that are natural and easy to interpret in a convenient way, similarly to natural languages. Thanks to virtual memory, these strings can be concatenated into strings of any length (up to the computer's memory limitations). During the execution phase, the encoding of data structures written in complex notations into data written in machine language is automatically taken care of by the functional hierarchy within the computer (operating system, compiler, and assembler).

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

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

The convenience of complex notations and the complex operations they represent, together with the natural tendency of programmers and users to assign them a semantics, makes it tempting to conclude that computers' inputs, outputs, and internal states are individuated by their content. For example, both computer scientists and philosophers sometimes say that computers *understand* their instructions. This analogy was briefly discussed in Chapter 7, Section 2.7. Now we can

return to that topic and clarify further in what sense computers do understand their instructions.

Computer instructions and data have functional significance, which depend on their role within the functional hierarchy of the computer. Before a program written in a complex notation is executed, its data and instructions are automatically encoded into machine language data and instructions. Then they can be executed. All the relevant elements of the process, including the programs written in complex notation, the programs written in machine language, and the programs constituting the functional hierarchy (operating system, compiler, and assembler), are strings of digits. All the operations performed by the processor in response to these instructions are operations on strings. The resulting kind of "computer understanding" is mechanistically explainable without ascribing *any* external semantics to the inputs, internal states, or outputs of the computer.

At the same time, data and instructions are inserted into computers and retrieved from them by programmers and users who have their own purposes. As long as the functional hierarchy is working properly, programmers and users are free to assign an external semantic interpretation to their data and instructions in any way that fits their purposes. This semantics applies naturally to the data and instructions written in the complex notation used by the programmers and users, although it may cease to be helpful when the complex code is compiled and assembled into machine language data and instructions. The semantics of a computer's inputs, outputs, and internal states is helpful in understanding how and why computers are used, but it is unnecessary to individuate computing mechanisms and the functions they compute.

6. Application 1: Are Turing Machines Computers?

Turing machines (TMs) can be seen and studied mathematically as lists of instructions or sequences of abstract strings, with no mechanism acting on the strings. They can also be seen as a type of computing mechanism, made out of a tape divided into squares and a unit that moves along the tape, acts on it, and is in one out of a finite number of internal states (see the Appendix and Davis et al. 1994 for more details). The tape and the active unit are the components of TMs. Their functions are, respectively, storing digits and performing operations on digits. The active unit of TMs is typically treated as a black box, though in principle it may be analyzed as a finite state automaton, which in turn may be analyzed as a Boolean circuit plus a memory register. When they are seen as mechanisms, TMs fall naturally under the present account of computation. They are uniquely defined by their mechanistic description, which includes their list of instructions.

There are two importantly different classes of TMs: ordinary TMs and universal TMs. Ordinary TMs are not programmable and *a fortiori* not universal. They are "hardwired" to compute one and only one function, as specified by the list of instructions that uniquely individuates each TM. Since TMs' instructions constitute

full-blown algorithms (i.e., algorithms defined over an infinite domain), by the light of the present account they are special-purpose computers.

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

This account contrasts with a common way of understanding TMs, according to which TM tapes are analogous to internal computer memories, but it is in line with Turing's original description of his machines:

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

Turing defines his machines by analogy with computing humans. As Eli Dresner (2003) points out, Turing does not even describe his machines as including the tape as a component, let alone as a memory component—the tape is "the analogue of paper" on which a human writes. For Turing, the analogue of human memory is not the machine's tape, but the machine's internal states ("*m*-configurations"). Only in 1945 and afterwards, when the design and construction of stored-program computers were under way, did Turing describe TM tapes as components of the machine and draw a functional analogy between TM tapes and computer memory (Turing 1945). Still, this analogy remains only partial, because computer memories properly so called are structurally and functionally distinct from input and output devices (Chapter 9). The distinction between memory, input devices, and output devices renders stored-program computers much more flexible in their use than literal TMs would be, making it possible to install and remove programs from memory, store multiple programs, switch between them, create a functional hierarchy, etc. These, in

turn, are some reasons why no one builds literal implementations of TMs for practical applications.[11]

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

7. Application 2: A Taxonomy of Computationalist Theses

Computing mechanisms have been employed in computational theories of cognition. These theories are sometimes said to be metaphors (Eliasmith 2003), but a careful reading of the relevant literature shows that computationalism is a literal mechanistic hypothesis (Piccinini 2003b, 2004a). The mechanistic account of computation can be used to add precision to computationalist theses about the brain by taxonomizing them in order of increasing strength. The theses range from the commonsensical and uncontroversial thesis that the brain processes information to the strong thesis that it is a programmable, stored-program, universal computer. Each thesis presupposes the truth of the preceding one and adds further assumptions to it, except for thesis (7), which presupposes (2) but rejects (3)–(6):

1) The brain is a collection of interconnected neurons that deal with information and control (in an intuitive, formally undefined sense).
2) The brain is a computing mechanism (generic connectionist or neurocomputational computationalism).
3) The brain is made out of Boolean circuits or finite state automata (McCulloch and Pitts 1943; Nelson 1982).
4) The brain is a programmable digital computer (Devitt and Sterelny 1999).
5) The brain is a programmable, stored-program digital computer (Fodor 1975).[12]
6) The brain is a universal digital computer (Newell and Simon 1976).
7) The brain is an analog computer (Rubel 1985).

Strictly speaking, (6) does not presuppose (5). For instance, universal TMs are not stored-program. In practice, however, all supporters of (6) also endorse (5), for the good reason that there is no evidence of a storage system in the environment—

[11] For an extended argument that computers have important features lacked by Turing machines, see Sloman 2002. Some of the features listed by Sloman are made possible, in part, by the distinction between memory, input devices, and output devices. For some friendly amendments to Sloman's argument, see Piccinini 2005.

[12] This thesis should not be confused with the uncontroversial fact that human beings are "programmable" in the sense of being able to follow different algorithms.

analogous to the tape of TMs—that would store the putative programs executed by brains.

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

Historically, thesis (3) is the first formulation of computationalism (Piccinini 2004a). Its weakening leads to (2), which includes modern connectionist and neurocomputational computationalism (Piccinini 2011; Piccinini and Shagrir 2014). Its strengthening leads to what is often called classical computationalism. Classical computationalism is usually identified with (5) or (6), but sometimes (4) and even (3) have been discussed as options. The above taxonomy allows us to see that different formulations of computationalism are not equivalent to one another, and that they vary in the strength of their assumptions about the mechanistic properties of neural mechanisms.

The above list includes only the theses that have figured prominently in the computationalist literature. It is by no means exhaustive of possible computationalist theses. First, notice that (3) can be divided into a relatively weak thesis, according to which the brain is a collection of Boolean circuits, and a much stronger one, according to which the brain is a collection of finite state automata. The mechanistic account of computation shows how to construct theses that are intermediate in strength between these two. For instance, we could hypothesize that the brain is a collection of components that can execute complex finite-domain algorithms (i.e., effective procedures defined only on finitely many inputs of bounded length), like the multiplication and division components of modern computers.

Another possible version of computationalism is the hypothesis that the brain is an ordinary calculator (as opposed to a digital computer, which is a special kind of calculator). This possibility is intriguing because I know of no one who has ever proposed it, even though calculators are *bona fide* computing machines. The mechanistic account sheds light on this fact. Among other limitations calculators have, their computational repertoire is fixed. There is no interesting sense in which they can learn to compute new things or acquire new computational capacities. One important factor that attracted people to computationalism is the flexibility and power of computers—flexibility and power that calculators lack. Because of this, it is unsurprising that no one has proposed that brains are calculators.

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

8. Application 3: Questions of Hardware

It is often said that even if the brain is a computing system, it need not have a von Neumann architecture (Pylyshyn 1984; Churchland and Sejnowski 1992). In these discussions, 'von Neumann architecture' is used as a generic term for the functional organization of ordinary digital computers. The claim that the brain need not have a von Neumann architecture is used to discount apparent dissimilarities between the functional organization of brains and that of ordinary digital computers as irrelevant to computationalism. The idea is that brains may compute in ways other than those exploited by modern digital computers.

It is true that computing systems need not have a von Neumann architecture. For example, Turing machines don't. But this does not eliminate the constraints that different versions of computationalism put on the functional organization of the brain, if the brain is to perform the relevant kinds of computation. During the current discussion, I intentionally avoided the term 'von Neumann architecture' because it is so vague that it obscures the many issues of functional organization that are relevant to the design and computing power of (digital) computing mechanisms. The present account allows us to increase the precision of our claims about computer and brain architectures, avoiding generic terms like 'von Neumann architecture' and focusing on various mechanistic properties of computing systems (and thus on what their computing power is).

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

1. What are the digits manipulated in the neural computation, and what are their types?
2. What are the elementary computational operations on neural digits, and what are the components that perform them?
3. How are the digits concatenated to one another, so that strings of them can be identified as inputs, internal states, and outputs of neural mechanisms and nontrivial computations from input strings to output strings can be ascribed to neural mechanisms?
4. What are the compositional rules between elementary operations, and the corresponding ways to connect the components, such that complex operations can be formed out of elementary ones and performed by the mechanism?

5. If the mechanism stores programs or even just data for the computations, what are the memory cells and registers and how do they work?
6. What are the control units that determine which operations are executed at any given time and how do they work? This question is particularly pressing if there has to be execution of programs, because the required kind of control unit is particularly sophisticated and needs to correctly coordinate its behavior with the components that store the programs.

When McCulloch and Pitts (1943) initially formulated computationalism, they had answers to the relevant versions of the above questions. In answer to (1), they argued that the presence of a neural spike and its absence are the two types of digit on which neural computations are defined. In answer to (2), they appealed to Boolean operations and claimed that they are performed by neurons. In answer to (3) and (4), they relied on a formalism they in part created and in part drew from Carnap, which is computationally equivalent to finite state automata. In answer to (5), they hypothesized that there are closed loops of neural activity, which act as memory cells. In answer to (6), they largely appealed to the innate wiring of the brain (Piccinini 2004a).

When von Neumann formulated his own version of computationalism (von Neumann 1958), he also tried to answer at least the first two of the above questions. In answer to (1), he maintained that the firing rates of neurons are the digit types. In answer to (2), he maintained that the elementary operations are arithmetical and logical operations on these firing rates. Although von Neumann's answers take into account the functional significance of neuronal spikes as it is understood by modern neurophysiologists, von Neumann did not have answers to questions 3 to 6, and he explicitly said that he did not know how the brain could possibly achieve the degree of computational precision that he thought it needed (Piccinini 2003b).

Today's digital computationalists no longer believe McCulloch's or von Neumann's versions of computationalism. But if digital computationalism is to remain a substantive, empirical hypothesis about the brain, these questions need to find convincing answers. If they don't, it may be time to abandon digital computationalism, at least as a general thesis about the brain, in favor of other mechanistic versions of computationalism (Piccinini and Bahar 2013).

9. Conclusion

Contrary to what some authors maintain (e.g., Churchland and Sejnowski 1992; Searle 1992), whether something is a digital computer, and what kind of digital computer it is, is an objective feature of a system. Digital computers are digital computing systems of large capacity, whose function is to perform computations that involve long sequences of primitive operations on strings of digits, operations that can be performed automatically by the computers' processors. Digital calculators are a

less powerful kind of digital computing machine. Digital computing machines, in turn, are distinct from other systems in that they manipulate strings of digits according to rules defined over the digits.

Different classes of digital computers can be programmed in different ways or compute different classes of functions. These and other useful distinctions between classes of digital computers can be drawn by looking at computers' mechanistic properties, and they can be profitably used in historical and philosophical discussions pertaining to computers and other computing mechanisms.

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

Explanations by program execution are invoked in the philosophy of mind literature (cf. Piccinini 2004b). Given the mechanistic account of digital computers, explanations by program execution are a special kind of mechanistic explanation that applies to soft-programmable computers. Soft-programmable computers are digital computers with processors that respond differentially to different strings of digits, to the effect that different operations are performed on data. Within a digital computer, program execution is a process by which (a stable state of) a certain part of the mechanism, the program, affects another part of the mechanism, the processor, so that the processor performs appropriate operations on (a stable state of yet) another part of the mechanism, the data. Only mechanisms with the relevant mechanistic properties are subject to explanation by program execution. By identifying more precisely the class of computers that support explanation by program execution and how they do so, the mechanistic account makes explicit the commitments of those who appeal to explanation by program execution in the philosophy of mind.

Finally, the present account of digital computers can be used to formulate a rigorous taxonomy of computationalist theses about the brain, which makes explicit their empirical commitments to specific functional and structural properties of brains, and to compare the strength of the different empirical commitments of different computationalist theses. This makes it ideal to ground discussions of computational theories of cognition.

My taxonomy of computationalist theses ended with the thesis that the brain is an *analog* computer. Analog computers preceded digital computers and competed with them for a few decades. Although they are no longer in widespread use, they remain an important class of computing systems. The next chapter takes a closer look.

12

Analog Computers

1. Disambiguating 'Analog'

In the previous chapter, I explicated digital computers. There are also analog computers. The distinction between analog and digital computers has generated confusion. For example, it is easy to find claims to the effect that analog computers (or even analog systems in general) can be approximated to any desired degree of accuracy by digital computers, countered by arguments to the effect that some analog systems are computationally more powerful than Turing machines (Siegelmann 1999). In this section, I will draw some of the distinctions that are lumped together under the analog-digital banner. In the next section, I will sketch a mechanistic account of analog computers properly so called.

First, analog *computation* properly so called should be distinguished from analog *modeling*. Sometimes, certain kinds of models or modeling technologies, such as wind tunnels and certain electrical circuits, are called 'analog computers' (e.g., Hughes 1999, 138) or 'direct analog computers' (e.g., Care 2010, 18 who uses 'indirect analog computers' for what I call analog computers properly so called). And sometimes analog models are used as the basis for the claim that the brain is a computer—according to this view, the brain is an analog computer because it contains analog models of its environment (Shagrir 2010a, b).

This use of the term 'analog computer' for analog models appears due to the following. Analog *models* in the class that is relevant here represent target systems, and can be used to draw inferences about target systems, because there is an "analogy," or similarity, between the model and the target system. Typically, both the model and the target system satisfy the same system of differential equations. The system of differential equations is the "analogy" between the model and the target system. Analog *computers* properly so called are often used to model target systems by implementing mathematical methods that solve systems of differential equations satisfied by the target system. Again, the system of differential equations is the "analogy" between the model and the target system. There is an important difference: the methods used by analog computers properly so called rely on integration, which is performed by integrators, which are absent in analog models. But, since both analog models and analog computers properly so called are used for modeling, and

both rely on an "analogy" with a target system, in some contexts it seems natural to call both of them 'analog computers'.

An immediate concern with this broad use of the term 'analog computer' is that lots of things can be said to be analogous to something else in some respect, perhaps captured by a system of differential equations satisfied by both systems, and used to draw inferences about it. This turns lots of things into an analog computer in this sense, which waters down the notion of analog computer.

This trivialization concern can be alleviated by considering only mechanisms whose *function* is bearing certain analogies to certain other systems. This functional restriction on analog models is similar to the functional restriction on computing mechanisms that grounds the present mechanistic account of computation (Chapters 6, 7). Still, the resulting class of systems remains too broad for present purposes; it includes systems that do not rely on any mathematical method for solving the equations that model their target system—rather, they model by simply satisfying the same system of equations.

More importantly, the notion of analog modeling is a semantic notion—it defines 'analog' in terms of a representational relation between the model and the target system. True, analog computers (like other analog models) are typically used to model other systems. Also true, the history of analog modeling is intertwined with the history of analog computing properly so called (Care 2010). But, as we saw in Chapter 3, computation does not require representation.

The notion of an analog model is legitimate and useful, but it is orthogonal to the notion of analog computation. Some analog computations are models, while other analog computations may be done for purely mathematical purposes or purely for fun without modeling anything at all. Conversely, some analog models do their modeling job by being implemented in an analog computer, whereas other analog models may be implemented in digital computers or simply by standing in certain stable relations to the system they model. Since analog modeling is orthogonal to analog computation, I will leave the notion of analog modeling aside.

A second issue concerns whether a system is continuous or discrete. Analog systems are often said to be continuous, whereas digital systems are said to be discrete. When some computationalists claim that neural systems are analog, their motivation seems to be that some of the variables representing neural networks can take a continuous range of values.[1] One problem with grounding the analog-digital distinction in the continuous-discrete distinction alone is that a system can only be said to be continuous or discrete under a given mathematical description, which applies to the system at a certain level of analysis. Thus, the

[1] Cf.: "The input to a neuron is analog (continuous values between 0 and 1)" (Churchland and Sejnowski 1992, 51).

continuous-discrete dichotomy, although relevant, seems insufficient to distinguish between analog and digital computers other than relative to a level.[2]

The only way to establish whether physical systems are ultimately continuous or discrete depends on fundamental physics. On one hand, some authors speculate that, at the most fundamental level, everything will turn out to be discrete (e.g., Toffoli 1984; Wolfram 2002; cf. Chapter 4, Section 4). If this were true, under this usage there would be no analog computers at the fundamental physical level. On the other hand, the physics and engineering of middle-sized objects are still overwhelmingly done using differential equations, which presuppose that physical systems as well as spacetime are continuous. This means that at the level of middle-sized objects, there should be no digital computers. But the notions of digital and analog computers have a well-established usage in computer science and engineering, which seems independent of the ultimate shape of physical theory. It is this usage that originally motivated analogies between brains and computers. Therefore, the continuous-discrete distinction alone is not enough to draw the distinction between analog and digital computers that is of interest to computer scientists, engineers, and theoretical neuroscientists.

Previous philosophical treatments of the digital-analog distinction have addressed a generic, intuitive distinction, with emphasis on modes of representation, and do not take into account the functional properties of different classes of *computers* (Goodman 1968; Lewis 1971; Haugeland 1981; Blanchowicz 1997; Katz 2008; Maley 2011; Schonbein 2014). Those treatments do not serve our present purposes, for two reasons. First, our current goal is to understand computers qua computers and not qua representational systems. In other words, we should stay neutral on whether computers' inputs, outputs, or internal states are representations, and if they are, on how they get to represent. Second, we are working within the mechanistic account of computation, according to which computing mechanisms should be understood in terms of their (non-semantic) mechanistic properties.

2. Processing Continuous Variables

Analog and digital computers are best distinguished by their mechanistic properties.[3] Like digital computers, analog computers are made of (appropriately connected) input devices, output devices, and processing units (and in some cases, memory units). Like digital computers, analog computers manipulate medium-independent vehicles and have the function of generating outputs in accordance with a general rule whose application depends on their input and (possibly) internal states. Aside

[2] By contrast, when I characterized digits as discrete, I implicitly assumed that the relevant level(s) of analysis was (were) determined by the system's functions.

[3] Authoritative works on analog computers, which I have used as sources, include Jackson 1960; Johnson 1963; Korn and Korn 1972; and Wilkins 1970.

from these broad similarities, however, analog computers are mechanistically very different from digital ones.

The most fundamental difference is in the vehicles they manipulate. Whereas the inputs and outputs of digital computers and their components are strings of digits, the inputs and outputs of analog computers and their components are continuous or *real variables* (Pour-El 1974). From a mechanistic perspective, real variables are physical magnitudes that (i) *vary* over time, (ii) (are assumed to) take a *continuous range of values* within certain bounds, and (iii) (are assumed to) vary *continuously* over time. Examples of real variables include the rate of rotation of a mechanical shaft and the voltage level in an electrical wire.

The operations performed by computers are defined over their inputs and outputs. Whereas digital computers and their components perform operations defined over strings of digits, analog computers and their components perform operations on portions of real variables. Specifically, analog computers and their processing units have the function of transforming an input real variable received during a certain time interval into an output real variable that stands in a specified functional relation to the input. The discrete nature of strings makes it so that digital computers perform discrete operations on them (that is, they update their states only once every clock cycle), whereas the continuous change of a real variable over time makes it so that analog computers must operate continuously over time. By the same token, the rule that specifies the functional relation between the inputs and outputs of a digital computer is an effective procedure, i.e., a sequence of instructions, defined over strings from a finite alphabet, which applies uniformly to all relevant strings, whereas the rule that represents the functional relation between the inputs and outputs of an analog computer is a system of differential equations.

Due to the nature of their inputs, outputs, and corresponding operations, analog computers are intrinsically less precise than digital computers, for two reasons. First, analog inputs and outputs can be distinguished from one another, by either a machine or an external observer, only up to a bounded degree of precision, which depends on the precision of the preparation and measuring processes. By contrast, by design digital inputs and outputs can always be unambiguously distinguished from one another. Second, analog operations are affected by the interference of an indefinite number of physical conditions within the mechanism, which are usually called "noise," to the effect that their output is usually a worse approximation to the desired output than the input is to the desired input. These effects of noise may accumulate during an analog computation, making it difficult to maintain a high level of computational precision. Digital operations, by contrast, are unaffected by this kind of noise—either they are performed correctly, regardless of noise, or else they return incorrect results, in which case the system miscomputes.

Another limitation of analog computers, which does not affect their digital counterparts, is inherited from the limitations of any physical device. In principle, a real variable can take any real number as a value. In practice, a physical magnitude within

a device can only take physical values within bounds set by the physical limits of the device. Physical components malfunction or break down if some of their relevant physical magnitudes, such as voltage, take values beyond certain bounds. Therefore, the values of the inputs and outputs of analog computers and their components must fall within certain bounds; for example, ± 100 volts. Given this limitation, using analog computers requires that the problems being solved be appropriately scaled so that they do not require the real variables being manipulated by the computer to exceed the proper bounds of the computer's components. This is an important reason why the solutions generated by analog computers need to be checked for possible errors by employing appropriate techniques (which often involve the use of digital computers).

Analog computers are designed and built primarily to solve systems of differential equations. The most effective general technique for this purpose involves successive integrations of real variables. Because of this reliance on integration, the crucial components of analog computers are *integrators*, whose function is to output a real variable that is the integral of their input real variable. The most general kinds of analog computer that were traditionally built—*general-purpose analog computers*—contain a number of integrators combined with at least four other kinds of processing unit, which are defined by the operations they have the function to perform on their input. *Constant multipliers* have the function of generating an output real variable that is the product of an input real variable multiplied by a real constant. *Adders* have the function of generating an output real variable that is the sum of two input real variables. *Variable multipliers* have the function of generating an output real variable that is the product of two input real variables. Finally, *constant function generators* have the function of generating an output whose value is constant. Many analog computers also include special components that generate real variables with special functional properties, such as sine waves. By connecting integrators and other components in appropriate ways, which may include feedback (i.e., recurrent) connections between the components, analog computers can be used to solve certain classes of differential equations.

Pure analog computers can be set up to perform different sequences of primitive operations, and in this sense, they are programmable. This notion is similar to that of hard programmability for digital computers but, unlike hard programmability, analog programmability is not a precursor to soft programmability, because it does not involve programs. Programs, which are the basis for soft programming digital computers, are sequences of instructions defined over strings. They are not defined over the real variables on which analog computers operate. Furthermore, programs cannot be effectively encoded as values of real variables. This is because for a program to be effectively encoded, the device that is responding to it must be able to unambiguously distinguish it from other programs. This can be done only if a program is encoded as a string. Effectively encoding programs as values of real

variables would require unbounded precision in storing and measuring a real variable, which is beyond the limits of current (and foreseeable) analog technology.

Analog computers can be divided into special-purpose computers, whose function is to solve limited classes of differential equations, and general-purpose computers, whose function is to solve larger classes of differential equations. Insofar as the distinction between special- and general-purpose analog computers has to do with flexibility in their application, it is analogous to the distinction between special- and general-purpose digital computers. But there are important disanalogies: these two distinctions rely on different functional properties of the relevant classes of devices, and the notion of general-purpose analog computer, unlike its digital counterpart, is not an approximation of Turing's notion of computational universality (see Chapter 10, Section 4). Computational universality is a notion defined in terms of computation over strings, so analog computers—which do not operate on strings—are not devices for which it makes sense to ask whether they are computationally universal. Moreover, computationally universal mechanisms are computing mechanisms that are capable of responding to any program (written in an appropriate language). We have already seen that pure analog computers are not in the business of executing programs; this is another reason why analog computers are not in the business of being computationally universal.

It should also be noted that general-purpose analog computers are not maximal kinds of computer in the sense in which standard general purpose digital computers are. At most, a digital computer is capable of computing the class of Turing-computable functions.[4] By contrast, it may be possible to extend the general-purpose analog computer by adding components that perform different operations on real variables, and the result may be a more powerful analog computer.[5]

Since analog computers do not operate on strings, we cannot apply Turing's notion of computable functions over strings directly to measure the power of analog computers. Instead, we can measure the power of analog computers by employing the notion of function of a real variable. Refining work by Shannon (1941), Pour-El identified precisely the class of functions of a real variable that can be generated by general-purpose analog computers. They are the differentially algebraic functions, namely, functions that arise as solutions to algebraic differential equations (Pour-El 1974; see also Lipshitz and Rubel 1987; and Rubel and Singer 1985). Algebraic differential equations are equations of the form $P(y, y', y'', \ldots y^{(n)}) = 0$, where P is a polynomial function with integer coefficients and y is a function of x. It has also been shown that there are algebraic differential equations that are "universal" in the sense that any continuous function of a real variable can be approximated with arbitrary accuracy over the whole positive time axis $0 \leq t < \infty$ by a solution of the

[4] I am here ignoring the possibility, for now rather speculative, of building hypercomputers (cf. Chapter 16).

[5] For a step in this direction, see Rubel 1993.

equation. Corresponding to such universal equations, there are general-purpose analog computers with as little as four integrators whose outputs can, in principle, approximate any continuous function of a real variable arbitrarily well (Duffin 1981; Boshernitzan 1986).

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

Some limitations of analog computers can be overcome by adding digital components to them and by employing a mixture of analog and digital processes. In fact, the last generation of analog computers to be widely used were analog-digital hybrids, which contained digital memory units as well as digital processing units capable of being soft-programmable (Korn and Korn, 1972). In order to build a stored-program or soft-programmable analog computer, one needs digital components, and the result is a computer that owes the interesting computational properties that it shares with digital computers (such as being stored-program and soft-programmable) to its digital properties.

3. Conclusion

Given how little (pure) analog computers have in common with digital computers, calculators, and other computing mechanisms, and given that analog computers do not even perform computations in the sense defined by the mathematical theory of computation, one may wonder why both classes of devices are called 'computers'.

Part of the answer lies in the history of these devices. As I mentioned, the term 'computer' was apparently first used for a digital computer in something close to the contemporary sense by John Atanasoff in the early 1940s. At that time, what we now call analog computers were called *differential analyzers*. Digital machines operating on strings of digits, such as Atanasoff 's ABC, were often designed to solve problems similar to those solved by differential analyzers—namely, solving systems of differential equations—by the manipulation of strings. Since the new digital machines operated on digits and could replace computing humans at solving complicated problems by following algorithms, they were dubbed 'computers'. The differential analyzers soon came to be re-named 'analog computers', perhaps because both classes of machines were initially designed for similar practical purposes, and for a few decades they competed with each other. These historical factors should not blind

us to the fact that analog computers manipulate vehicles and perform operations that are radically different from those of digital computers.

The rest of the answer is provided by the mechanistic account of computation. Both analog and digital computers are devices whose function is manipulating vehicles according to rules that are sensitive to differences between different portions of the vehicles—in other words, both analog and digital computers manipulate medium-independent vehicles. In the case of digital computers, their vehicles are strings of digits and their rules are instructions defined over the strings. In the case of analog computers, their vehicles are real variables and their rules are systems of differential equations. Since both digital and analog computers manipulate vehicles in accordance with appropriate rules, they count as systems that perform (digital and analog, respectively) computations.

Having accounted for both analog and digital computers, there remains an important class of computing devices that are often invoked both in computer science, cognitive science, and elsewhere: neural networks. The next chapter is all about them.

13

Parallel Computers and Neural Networks

1. Neural Networks

In the previous two chapters, we looked at digital and analog computers. Neural networks are another important class of systems that are often mentioned in discussions of computation. There are long-standing disputes about whether neural networks perform computations and which kind of computation they perform.

This chapter applies the mechanistic account to neural networks and resolves these disputes. I distinguish several notions of parallel computing, introduce neural networks, identify the sense in which neural networks are parallel computing systems, and distinguish several classes of neural networks based on the kind of computation they perform. I explicate and defend the following theses: (1) Many neural networks compute—they perform computations. (2) Some neural networks compute in a classical way. Ordinary digital computers, which are very large networks of logic gates, belong in this class of neural networks. (3) Other neural networks perform digital computations in a non-classical way. (4) Yet other neural networks perform non-digital computations. Brains may well fall into this last class.

Neural networks are sets of connected signal-processing units. Typically, they have units that receive inputs from the environment (input units), units that yield outputs to the environment (output units), and units that communicate only with other units in the system (hidden units). Each unit receives input signals and delivers output signals as a function of its input and current state. As a result of their units' activities and organization, neural networks turn the input received by their input units into the output produced by their output units.

A neural network may be either a concrete physical system or a mathematically defined system, which in turn can be used as a description of a physical system (cf. Chapter 1). A mathematically defined neural network may be used to model another system to some degree of approximation. The modeled system may be either a concrete neural network or something else; e.g., an industrial process.

Psychologists and neuroscientists of a connectionist or neurocomputational persuasion use mathematically defined neural networks to model cognitive and neural systems. They often propose their theories as alternatives to classical, or "symbolic,"

computational theories of cognition. According to classical theories, the brain is analogous to a digital computer (Newell and Simon 1976; Fodor and Pylyshyn 1988; Pinker 1997; Rey 1997; Gallistel and Gibbon 2002). According to connectionist and neurocomputational theories, the brain is a (collection of) neural network(s).

Given the standard way neural networks are defined, classical, connectionist, and neurocomputational theories are not necessarily in conflict. Nothing in the definition of 'neural network' prevents the brain, a concrete neural network par excellence, from being a (classical) digital computer.

The term 'connectionist system' is more or less synonymous with 'neural network'.[1] Brains, of course, are neural networks. More precisely, there is overwhelming evidence that nervous systems carry out their information processing, cognitive, and control functions primarily in virtue of the activities of the neural networks they contain. In this sense, it should be uncontroversial that brains are concrete connectionist systems and cognition is explained by connectionist processes. Both connectionists and classicists should agree on this much.

To bring out the contrast between the two theories, we need a more qualified statement: according to paradigmatic connectionist and neurocomputational theories, the brain is a (collection of) non-classical neural network(s). This statement is informative only insofar as there is a nontrivial distinction between classical and non-classical systems. This is not the same as the distinction between systems that compute and systems that don't. We should be able to ask whether any, some, or all non-classical neural networks are computational. A clear answer to this question is needed to resolve the dispute between classicists and anti-classicists about the nature of cognition.

Many mainstream connectionist and neurocomputational theorists agree with classicists that brains perform computations and neural computations explain cognition (Marr and Poggio 1976; Feldman and Ballard 1982; Hopfield 1982; Rumelhart and McClelland 1986; Schwartz 1988; Churchland 1989; Cummins and Schwarz 1991; Churchland and Sejnowski 1992; Koch 1999; Bechtel and Abrahamsen 2002; Eliasmith 2003; Roth 2005; O'Brien and Opie 2006; Shagrir 2006b,; Smolensky and Legendre 2006). The claim that distinguishes such connectionist or neurocomputational computationalists from classicists is that according to connectionist/neurocomputational computationalism, non-classical neural networks are a better model of the brain than classical computing systems. In reply, some classicists argue that (non-classical) connectionist systems do not perform computations at all (Fodor 1975; Pylyshyn 1984; Gallistel and Gibbon 2002). According to such classicists, only

[1] Models in the connectionist tradition tend to be constrained solely or primarily by behavioral evidence, whereas models in the computational neuroscience tradition tend to be constrained both by behavioral evidence and neurophysiological and neuroanatomical evidence. This important difference between the two traditions does not affect that when the two classes of models are mathematically defined, they are defined more or less in the same way—roughly, as sets of connected signal-processing units.

classical systems perform genuine computations. This may not bother a different group of connectionist theorists, who reject or downplay the claim that brains compute (Perkel 1990; Edelman 1992; Globus 1992; Horgan and Tienson 1996; Freeman 2001).

Who's right? Do brains compute? Do (non-classical) neural networks compute? Which kind of system—classical or non-classical, computational or non-computational—is the best model for the brain?

Making progress on these debates—between classicists and anti-classicists as well as between computationalists and anti-computationalists—requires independently motivated distinctions between, on the one hand, systems that compute and systems that don't, and on the other hand, classical and non-classical systems. By applying such distinctions to neural networks, we can find out which neural networks, if any, do or do not perform computations, and which, if any, are classical or non-classical. Yet it has proven difficult to draw such distinctions in a satisfactory way.

The same problem may be framed in terms of theories of cognition. Is cognition explained by non-classical, neural computations? The answer depends on both what the brain does and where we draw two lines: (i) the line between neural computation and other kinds of neural processes and (ii) the line between classical computation and non-classical computation. Drawing these lines in a satisfactory way is a contribution to several projects: a satisfactory account of computation, a correct understanding of the relationship between classical and connectionist theories of cognition, and an improved understanding of cognition and the brain.

2. Do Neural Networks Compute?

Accounts of the nature of computation have been hindered by the widespread view that computing requires executing programs. Several authors embrace such a view (Fodor 1975; Pylyshyn 1984). Some authors endorse the stronger view that computing *is* program execution: "To compute function g is to execute a program that gives o as its output on input i just in case $g(i) = o$. Computing reduces to program execution" (Cummins 1989, 91; Roth 2005). The weaker view—namely, that program execution is a necessary condition for genuine computing—is strong enough for our purposes. Such a view is plausible when we restrict our attention to at least some classical systems. The same view gives rise to paradoxical results when we consider non-classical systems.

The view that computation requires program execution leads to a dilemma: either neural networks execute programs or they don't compute. Different people have embraced different horns of this dilemma.

A computationalist who is opposed to (paradigmatic) connectionist theories might wish to deny that neural networks—or at least, paradigmatic examples of neural networks—perform computations. Here is something close to an outright denial: "so long as we view cognition as computing in any sense, we must view it as *computing*

over symbols. No connectionist device, however complex, will do" (Pylyshyn 1984, 74, italics original). A denial that neural networks compute is also behind the view that connectionism is not a computationalist framework, but rather, say, an associationist framework, as if the two were mutually exclusive (Gallistel and Gibbon 2002).

In light of the thesis that computing requires executing programs, rejecting the idea that neural networks perform computations may sound like a reasonable position. Unfortunately, this position doesn't fit with the observation that the input-output mappings produced by many paradigmatic neural networks may be characterized by the same formalisms employed by computability theorists to characterize classical computing systems.

It is difficult to deny that many paradigmatic examples of neural networks perform computations in the same sense in which Turing machines and digital computers do. The first neural network theorist to call his theory 'connectionist' appears to be Frank Rosenblatt (1958). Rosenblatt's Perceptron networks and subsequent extensions and refinements thereof can be studied by the same formalisms and techniques that are employed to study other paradigmatic computing systems; their computing power can be defined and evaluated by the same measures (e.g., Minsky and Papert 1988).

Nowadays, the term 'connectionist system' is used to encompass more than Perceptrons. It usually encompasses all neural networks, as in my definition above. Rosenblatt himself was building on previous neural network models. He openly acknowledged that "the neuron model employed [by Rosenblatt] is a direct descendant of that originally proposed by McCulloch and Pitts" (Rosenblatt 1962, 5).

It just so happens that, roughly speaking, a McCulloch and Pitts neuron is functionally equivalent to a logic gate. A collection of (connected, synchronous, ordered) logic gates with input and output lines and without recurrent connections is a Boolean circuit. Boolean circuits are the components of ordinary (classical) digital computers (Chapter 8). Because of this, there is a straightforward sense in which digital computers are neural networks, aka connectionist systems. Classicism is a form of connectionism!

Of course, as I discussed in Chapter 11, digital computers are computationally more powerful than Boolean circuits, because they have recurrent connections between their components and because of their specific functional organization. In terms of their computation power, digital computers are finite state automata. Under the idealization of taking their memory to be unbounded, digital computers are computationally equivalent to universal Turing machines.

That digital computers are just one (quite special) kind of neural network is underappreciated by those who debate the merits of connectionist versus classical computational theories of cognition (cf. Macdonald and Macdonald 1995). In that debate, connectionist systems (neural networks) are often pitched without further qualifications against classical systems such as digital computers. But unless Boolean circuits and collections thereof are excluded from consideration—which there is no

principled reason for doing—denying that neural networks perform computations is tantamount to denying that computers compute.

Digital computers execute programs, and executing programs is an important aspect of the way they compute. Insofar as digital computers qualify as neural networks, at least some neural networks execute programs. But as we have seen, many non-classical neural networks, such as systems in Rosenblatt's tradition, perform computations too. Whether they compute by executing programs will be discussed in the next section.

In summary, the view that computation requires program execution generates a dilemma: either neural networks execute programs or they don't compute. We have now ruled out the second horn of the dilemma. To put the point bluntly, the computational properties of many neural networks are studied by one branch of computability and computational complexity theory among others.

3. Do Neural Networks Execute Programs?

Having ruled out one horn of our dilemma, let's consider the other. Perhaps (non-classical) neural networks execute programs, after all. Except for one small detail: where are such programs? Did we somehow miss the memory components in which (non-classical) neural networks store their programs? Of course not. The view that neural networks execute programs requires weakening the notion of program execution to the point that, contrary to what you might have supposed, executing programs does not require writing, storing, or physically manipulating a concrete program in any way—at least in the sense in which programs are manipulated by ordinary computers.

Here is an example: "programs . . . are just specifications of functional dependencies, and . . . a system executes a program if the system preserves such dependencies in the course of its state changes" (Roth 2005, 465).

This is not just an ad hoc proposal. Something like this weak notion of program execution predates the rise of neo-connectionism in the mid-1980s, which brought the question of whether connectionist systems (i.e., neural networks) perform computations to the attention of philosophers. A similar notion was often employed during discussions of classical computing systems: "programs aren't causes but abstract objects or play-by-play accounts" (Cummins 1983, 34; see also Cummins 1977, 1989; Cummins and Schwarz 1991).

This proposal may be put as follows: all that program execution requires is acting in accordance with a program. Acting in accordance with a program, in turn, may be explicated in at least two ways.

In the ordinary sense, 'acting in accordance with a program' means performing the operations specified by a program in the specified order. For instance, a program for multiplication might require the computation of partial products followed by their addition. A system that acts in accordance with such a program must generate

intermediate results corresponding to partial products and then generate a final result corresponding to their sum.

This notion of acting in accordance with a program does not solve the present problem. Typical non-classical neural networks do not generate successive outputs corresponding to separate computational operations defined over their inputs. Rather, they turn their inputs into their outputs in one step, as it were. Thus, they do not act in accordance with a program in the ordinary sense.

More recently, Martin Roth (2005) proposed a novel construal of what I'm calling 'acting in accordance with a program'. Under his construal, acting in accordance with a program does not require the temporal ordering of separate operations. Instead, it requires that the weights of a neural network be defined in a way that (i) satisfies the logical relations between the program's operations and (ii) results in a system input/output equivalent to the program.

Consider again a neural network that performs multiplications. Under Roth's proposal, the system acts in accordance with a partial products program if and only if the system's connection weights are derived from a partial products program, even though the system produces its output in one step. By contrast, if the connection weights are derived from a different multiplication program—such as a program for multiplying by successive additions—then the system doesn't act in accordance with a partial products program (but rather, a successive additions program).

Roth's notion of acting in accordance with a program appears to provide an ingenious solution to the present conundrum. If we accept that executing a program amounts to acting in accordance with one, we can now use Roth's notion of acting in accordance with a program to conclude that at least those neural networks that act in accordance with a program in Roth's sense do execute programs. This suggestion is not as helpful as it seems.

First, how can we derive connection weights from programs? Roth refers to a technical report by P. Smolensky, G. Legendre, and Y. Miyata, which was later expanded into a book (Smolensky and Legendre 2006). Smolensky and colleagues describe a technique for defining the weights of certain neural networks from certain computer programs. The resulting neural networks are input/output equivalent to the programs.

This technique applies only to a special class of neural networks. What about the others? Roth's notion of program execution does not appear to apply to them. If we wish to say that they compute, as per Section 2, we need another account of neural network computation.

It also remains to be seen whether, after the connection weights are defined using Smolensky et al.'s technique, the best thing to say is that—as Roth puts it—the neural network executes the program. It seems more appropriate to say that the neural network computes the function defined by the program without executing the program. The latter, sensibly enough, is Smolensky and Legendre's view.

Finally, suppose you want to say that a neural network executes a program based on Roth's proposal. Which programming language is it written in? In ordinary computers, this question has a well-defined answer. But it appears that the neural networks described by Roth act in accordance with any program, written in any programming language, which specifies the relevant operations in the relevant order. This makes it difficult to pick one of these programs as the one the system is executing. Or does it execute them all?

We need not press these questions further. For even if we grant Roth his notion of acting in accordance with a program, we should resist his conclusion that the neural networks he describes execute programs in the relevant sense. If acting in accordance with a program is sufficient for executing programs, then at least the systems defined using Smolensky et al.'s technique may well execute programs. This might work as a new meaning of 'program execution'. But in the sense of 'program execution' employed in computer science, acting in accordance with a program is hardly sufficient for program execution.

As we saw in Chapter 11, a program is a list of instructions implemented by a concrete string of digits; 'executing a program' means responding to each instruction by performing the relevant operation on the relevant data. As I'm using the term, digits are discrete stable states that can be stored in memory components and transmitted from component to component.

The modern notion of program execution originates as a way to characterize a special property of modern program-controlled computers (and some other machines; more on this soon). Program-controlled computers do much more than act in accordance with a program. They can act in accordance with any number of programs. This is because they can store physical instantiations of the programs, and it is those physical instantiations that, together with input data, drive computer processes.

Programs in this sense are much more than "specifications of functional dependencies." They are strings of digits that program-controlled computers can respond to and manipulate in the same way that they respond to and manipulate data. Programs can be written, tested, debugged, downloaded, installed, and, of course, executed. It is the execution of programs in this sense that explains the behavior of ordinary computers. This is also the notion of program execution that contributed to inspire the analogy between minds and computers, on which many computational theories of cognition are based.

If we want to honestly assess whether neural networks execute programs and use this assessment to compare different computational theories of cognition, we should use the standard notion of program execution that is used in computer science. And in this sense of 'program execution', paradigmatic non-classical neural networks do *not* execute programs.

4. Neural Networks and the Mechanistic Account of Computation

Our hands are tied. We have found that neural networks (or connectionist systems) perform computations even though they don't execute programs. We must conclude that computation does not require program execution. This is a good outcome. Several independent considerations point to the same conclusion.

First, nothing in the notion of computation studied by computer scientists and computability theorists entails that computation must be performed by executing a program. The original mathematical notion of computation is that of a process that accords with the steps of an algorithm or effective procedure—there is no further requirement that the algorithm be implemented by a program or that the program be executed.

Second, the notion of program-controlled machines, of which program-controlled computers are a species, did not even originate in the field of computing. It originated in the textile industry. The first technology for programming machines—punched cards and the mechanisms for executing them—was developed in the 18th Century to control the weaving of patterns in mechanical looms. In 1801, Joseph Marie Jacquard developed an improved version of this technology for his Jacquard Loom. Only later did Charles Babbage borrow Jacquard's idea to control his Analytical Engine. The Analytical Engine, which Babbage never managed to construct, was the first (hypothetical) program-controlled computer.

Third, in computer science there is a useful distinction between computing systems that execute programs and computing systems that don't. Computation, including mechanical computation, is much older than program-controlled computers. Ordinary calculators (Chapter 10) and many other computing devices don't execute programs in the sense in which full-blown computers do. Aside from neural networks, there are plenty of systems that compute without executing programs.

Finally, the standard explanation of how computers execute programs appeals to components that compute without executing programs (Chapter 9). Computers execute programs in virtue of possessing the relevant kind of processors. Such processors contain a control unit and a datapath. When an instruction reaches the processor, the control and the datapath perform two different functions. The control receives one part of the instruction—the part that encodes a command (e.g., sum, multiply, etc.). Then, the control unit determines which operation must be performed on the data and sends an appropriate signal to the datapath. The datapath receives the command from the control unit plus the remaining part of each instruction—the part that encodes the data. After that, the datapath performs the relevant operation on the data and sends the result out.

The control and the datapath are computing components—in the language of computability theory, they are finite state automata. But neither of them alone executes any program—it is only their organized effort, in cooperation with memory

components, which allows the computer to execute programs. Thus, in ordinary digital computers, program execution itself requires components that compute without executing programs.

For all these reasons, we need an account of computation that accommodates both computing systems that execute programs and computing systems that don't. Luckily, I articulated and defended such an account in previous chapters.

In my terminology, a functional mechanism is a system of organized components, each of which has functions to perform (Chapter 6). When appropriate components and their functions are appropriately organized and functioning properly, their combined activities constitute the capacities of the functional mechanism. Conversely, when we look for an explanation of the capacities of a functional mechanism, we decompose the mechanism into its components and look for their functions and organization. The result is a mechanistic explanation of the system's capacities.

Computing systems are a special class of functional mechanisms. They are distinguished from other mechanisms by the peculiar capacity they have. Their function is to manipulate medium-independent vehicles so as to generate output vehicles from input vehicles and (possibly) internal states in accordance with a general rule that applies to all relevant vehicles and depends on the inputs and (possibly) internal states for its application.

Classical computing mechanisms are computing systems that manipulate strings of digits one discrete step at a time (Chapters 7–11). Each step occurs during a functionally well-defined time interval. During each time interval, the mechanism produces an output string of digits. Thus, the mechanism's processes are *in accordance with* a program that defines a classical computation. (This is not yet program *execution*, which requires a program to be physically instantiated and manipulated.)

Classical computing mechanisms are computationally decomposable in the following sense. As a whole, a classical computing mechanism performs computations in accordance with a general rule defined over the strings of digits that are its vehicles. For instance, it computes the square root of its input. The performance of such a computation may be explained in terms of the mechanisms' components and the computations they perform. For instance, a square root computation is explained by the combined action of a memory component storing a square root program and a processor executing the program.

This explanatory strategy can be iterated. The processor's capacity to execute programs is explained by the computations performed by the control and datapath that constitute it and the way they are wired together (as mentioned above). The computations performed by the control and datapath are explained by the computations performed by the circuits that constitute them and the way those circuits are organized. Finally, the computations performed by circuits are explained by the operations performed by the logic gates that constitute the circuits and the way those gates are connected. Since the operations performed by logic gates are computationally primitive, computational decomposition stops with them. The capacities

of logic gates can still be mechanistically explained in terms of the organized functions performed by their components (resistors, capacitors, or whatever). But such mechanistic explanations no longer appeal to computations performed by their components.

To recapitulate where we've gotten so far, the idea that executing programs is necessary for computing leads to the dilemma that either neural networks execute programs or they don't compute. On the first horn, we must disconnect the notion of program execution from the special type of mechanism that gave rise to the idea of (computational) program execution in the first place. On the second horn, we must say that systems that compute by the lights of computability theory do not, in fact, compute. Fortunately, we need not impale ourselves on either horn. It is much better to embrace a broader, mechanistic view of computation, according to which computation doesn't require program execution. This is a good result, with plenty of independent motivation. After abandoning the mistaken notion that computation requires program execution, we can gain a better understanding of neural network computation.

5. Serial vs. Parallel Computation

Neural networks are parallel systems par excellence. But the distinction between serial and parallel computers has generated some confusion. A common claim is that the brain cannot be a "serial computer" like our ordinary digital computers, because it is "parallel" (e.g., Churchland et al. 1990, 47; Dennett 1991, 214; Churchland and Sejnowski 1992, 7). In evaluating this claim, we should keep in mind that digital computers can be parallel too, in several senses of the term. There are several distinctions to be made, and our understanding of computation can only improve if we make at least the main ones explicit.

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

A separate question is whether a computing mechanism can perform only one or more than one computational operation during any relevant time interval. Analog computers are parallel in this sense. This appears to be the sense that connectionist computationalists appeal to when they say that the brain is "massively parallel." But again, most complex computing mechanisms, such as Boolean circuits, are parallel in this sense. Since most (digital) computers are made out of Boolean circuits and other computing components that are parallel in the same sense, they are parallel in this

sense too. In this respect, then, there is no principled difference between ordinary computers and (other, non-classical) neural networks.

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

In the connectionist and neural network literature, there is a tendency to call neurons 'processors' (e.g., Siegelmann 2003). In many cases, this language implicitly or explicitly suggests an analogy between a brain and a parallel computer that contains many processors, so that every neuron corresponds to one processor. This is misleading, because there is no useful sense in which a neuron can be said to execute instructions in the way that a computer processor can. In terms of their functional role within a computing mechanism, neurons are more similar to logic gates than to the processors of digital computers. In fact, modeling neurons as logic gates was the basis for the first formulation of a computational theory of cognition (McCulloch and Pitts 1943). Nevertheless, it may be worth noticing that if the comparison between neurons and processors is taken seriously, then in this sense—the most important for computability theory—neurons are serial processors, because they perform only one functionally relevant activity (i.e., they fire at a certain rate) during any relevant time interval. Even if we consider an entire neural network as a processor, we obtain the same result; namely, that the network is a serial processor (it turns one input string into one output string). However, neither neurons nor ordinary neural networks are really comparable to computer processors in terms of their organization and function—they certainly don't execute

instructions. So, to assimilate neural networks to computer processors in this respect is inappropriate.

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

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

6. Digital Neural Network Computation

Many neural networks perform digital computations—they manipulate strings of digits in accordance with an appropriate rule.

When the properties of paradigmatic neural networks are characterized computationally, the inputs to their input units and the outputs from their output units are relevant only when they are in one of a finite number of (equivalence classes of) states. Therefore, these neural networks' inputs and outputs—though not necessarily the inputs and outputs of their hidden units—are digits in the present sense.

Even when the values of a neural network's inputs or outputs are assumed to vary continuously, in many cases there are discontinuities in the way the different values are classified. Typically, only values in certain neighborhoods (e.g., two neighborhoods labeled '0' and '1') are counted as determining what the whole system's input or output is; there may be gaps between neighborhoods; and all values within each neighborhood are counted as functionally equivalent. Therefore, the systems' inputs and outputs still constitute digits.

Furthermore, depending on the kind of system, either the spatial ordering of the input units or the temporal sequence of the input units' inputs constitute a string of digits, and either the spatial ordering of the output units or the temporal sequence of the output units' outputs constitute a string of digits. Such strings are the entities in terms of which the computation power of the system is defined.

Finally, the way the system's outputs are (or are supposed to be) functionally related to its inputs (plus, perhaps, its internal states) constitute a rule defined over the digits. Such a rule defines the computation performed (or approximated) by the system.

The first authors who defined a class of neural networks and ascribed computations to them were Warren McCulloch and Walter Pitts (McCulloch and Pitts 1943). The above account applies straightforwardly to McCulloch-Pitts networks. Each unit receives and returns only two values, typically labeled '0' and '1'. These are the digits. Each unit returns an output that stands in a definite logical relation to its inputs (e.g., AND, OR). The units have a discrete dynamics and are synchronous, so that each unit processes its inputs and returns its output during the same time intervals. Input and output units are arranged in well-defined layers; units in each layer can be ordered from first to last. So during any time interval, the inputs going into the input units and the outputs coming from the output units constitute well-defined strings of digits. Finally, the structure of the network doesn't change over time.

Rosenblatt's Perceptrons (of any number of layers) are nothing but McCulloch-Pitts networks in which the last condition is dropped. Instead of having a fixed structure, Perceptrons can be trained by varying their connection weights during a special period of time, called the "training period." The same is true of Adalines, another early and influential type of neural network (Widrow and Hoff 1960).

Being trainable makes no difference to whether a system is computational. After a Perceptron or Adaline is trained, the system maps its input string of digits onto its output string of digits according to a fixed rule. An elementary example of such a rule is EXCLUSIVE OR (either x or y, but not both). EXCLUSIVE OR is notorious in connectionist circles because its computation requires Perceptrons or Adalines with hidden units; until the 1970s, some authors doubted that multi-layer networks could be trained reliably (Minsky and Papert 1988). The discovery of training methods for such networks (Werbos 1974) proved such doubts to be unfounded.

Thus, Perceptrons and Adalines are nothing but trainable McCulloch-Pitts networks (cf. Cowan 1990). After they are trained, they compute the same function computed by the corresponding McCulloch-Pitts network. (Of course, trainable neural networks may be trained more than once; after each training period, they may compute a different function.) Other classes of neural networks can be defined by relaxing more of McCulloch and Pitts's assumptions.

We may allow units to take any real-valued quantity as input, internal state, or output (instead of only finitely many values, such as '0' and '1'), and we may let the input-output function of the processing units be nonlinear. To be physically realistic, the range of values that units can take and process must be restricted to a finite interval. And in many paradigmatic applications, the inputs to and outputs from the network are defined so that all values within certain intervals are taken to be functionally equivalent. For instance, the real-valued quantities may be restricted to the interval [0, 1]; all values near 0 may be labeled '0'; all values near 1 may be labeled '1'.

Thus, all values within the right neighborhoods count as digits of the same type, which may be ordered into strings according to the ordering of the network's input and output units. When such a convention is in place, the power of such networks can be defined in terms of classical computability theory.

We may also let the units be asynchronous, let the network's dynamics be continuous in time, or both. When this is done, there may no longer be any way to individuate input and output strings of digits, because there may no longer be a well-defined way of classifying inputs or outputs into discrete equivalence classes (digits) and ordering different digits into strings. Unless, that is, the network receives its inputs all at once and there are conventions for grouping outputs into equivalence classes and ordering them into strings. A standard way of doing so is to consider network dynamics that stabilize on a stable state, from which networks produce constant outputs. Under such circumstances, inputs and outputs still constitute strings of digits. Unsurprisingly, these conditions are in place whenever the power of such networks is analyzed in terms of classical computability theory. I will discuss networks that fail to satisfy these conditions in the next section.

One moral of the above discussion is that digital neural network computation can be either classical or non-classical. For instance, McCulloch-Pitts nets are perfectly classical. They are so classical that digital computers are essentially made out of them. Now it's time to say more explicitly what's peculiar about non-classical (but still digital) neural network computation.

Unlike classical computing systems, (paradigmatic) neural networks may be trained. Training is the adjustment of the connections between the units to fit a desired input-output rule. Thus, the dynamics of neural networks may change over time independently of which inputs they get. In other words, a neural network may evolve so as to yield different outputs in response to the same inputs at different times. Understanding the causal mechanism behind the system's input-output rule and its evolution over time requires understanding the dynamical relations between the system's units and the way they can change.

Besides trainability, there is a deeper difference between classical and non-classical systems. Unlike the former, the latter do not act in accordance with a step-by-step procedure, or program. Classical computation proceeds by performing one operation at a time, where an operation is a change of one string of digits into another. Since strings are (sequences of) discrete entities, changing one string into another is a discrete operation. Thus, a classical computational dynamics—involving many transformations of one string into another—is by definition discrete. By contrast, many neural networks proceed by letting their units affect each other's activation values according to dynamical relations that vary in continuous time. A neural network's dynamics is the analog of an ordinary computer's digital logic. Whereas digital logic determines discrete dynamics, the typical dynamic of a neural network is continuous.

Because of this, many neural networks are not computationally decomposable like classical computing systems. That is, such neural networks have computational capacities, but these capacities cannot be explained in terms of digital computational steps performed by their components together with the way their components are organized. The reason is that in the case of an irreducibly continuous dynamics, there is no well-defined way of breaking down the process by which a system generates its outputs from its inputs into intermediate, discrete computational steps, equivalent to the transformation of one input string into an output string.

It doesn't follow that there is no mechanistic explanation of neural network computations. The activities of neural networks are still constituted by the activities of their components and the way the components are organized and they may even be constituted by simpler computations, provided that those are not digital computational steps. It's just that to explain how typical neural networks work, we need mathematical tools different from digital logic and computability theory—tools like cluster analysis, nonlinear dynamics, and statistical mechanics.

In brief, there are two dimensions along which classical and non-classical computing systems differ. First, they may or may not be trainable. Second, they may compute either by acting in accordance with a program (or algorithm) or by following a dynamics that cannot be decomposed into intermediate digital computational steps. Purely classical systems (e.g., digital computers, McCulloch-Pitts nets) have the first two characteristics (fixed structure and acting in accordance with a program). Many non-classical systems have the second two characteristics (trainability and continuous dynamics), but there are also systems that are non-classical only in one way (e.g., the original Perceptrons and Adalines, which are trainable but have discrete dynamics).

If this is right, why do many connectionists speak of connectionist algorithms, as they often do, even when talking about systems with continuous dynamics? The term 'connectionist algorithm' is used in several ways. The present account allows us to distinguish and elucidate all of them.

Sometimes, 'connectionist algorithm' is used for the procedure that trains a neural network, that is, the procedure for adjusting the dynamical relations between units. This is typically an algorithm in the classical sense—a list of instructions for manipulating strings of digits. Training algorithms, of course, are not what actually drives any given neural network computation: neural network computations are driven by the state of and dynamical relations between a system's units. Thus, from the fact that a system is trained using an algorithm, it doesn't follow that the system itself computes by following an algorithm.

But 'connectionist algorithm' is also often used for the process by which a neural network produces its output. Based on what we have seen, typically this is not an algorithm in the classical sense. Many neural networks do not have the kind of discrete dynamics defined over strings of digits that can be accurately described by algorithms in the classical sense. Therefore, this second usage of 'connectionist

algorithm' is equivocal. But given how common it has become, it may be considered a new or extended sense of the term 'algorithm'—not to be confused with the classical sense.

Finally, the process by which a series of neural networks, each one feeding into the next, generates a final output through intermediate processes is sometimes called a 'connectionist algorithm'. This may well be an algorithm in the classical sense—or something close—provided that each intermediate process is definable as a transformation of strings of digits. Or else, it is an algorithm in the extended sense explicated in the previous paragraph.

7. Non-digital Neural Network Computations

In previous sections, I gave an account of what it takes for a neural network to perform digital computations: its inputs and outputs must be strings of digits, and its input-output relationship must accord with a rule defined over the digits (and possibly internal states). Implicitly, such an account also specifies conditions under which a neural network performs non-digital computations or does not perform computations—all it takes is for a system to fail to satisfy the conditions under which a system performs digital computations. This happens when its input-output relationship does not accord with an appropriate rule, its inputs and outputs are not strings of digits, or both.

The best example of a system that does not act in accordance with a rule at all is a system that generates random outputs. Since its outputs are random, there is no rule that relates the outputs to the inputs. There are also systems that act in accordance with a rule, but the rule is not defined over medium-independent vehicles (Chapter 7). An example is a mechanical loom, which performs the same actions regardless of what its inputs are like (or even whether it receives any inputs at all). Neither random systems nor mechanical looms are computing systems, because they don't act in accordance with the right kind of rule.

It is certainly possible to define neural networks with random outputs or outputs that are unrelated to properties of the inputs. They would be atypical. Typical neural networks (have the function to) respond to the properties of their inputs in accordance with a rule defined over the inputs in a medium-independent way. Therefore, typical neural networks qualify as computing systems under the mechanistic account. But there are many neural networks whose inputs and outputs, for one reason or another, are not strings of digits.

For instance, some neural networks manipulate continuous variables (Chen and Chen 1993). Understanding the processing of continuous variables requires specialized mathematical tools, which differ from the discrete mathematics normally employed by computability theorists and computer scientists. At the very least, there is no straightforward way to assign computation power, in the standard sense defined over effectively denumerable domains, to systems that manipulate

continuous variables. Because of this, it seems appropriate to conclude that neural networks that process continuous variables do something other than digital computing. They are much closer to analog computers (Chapter 12).

More generally, any neural network whose inputs and outputs violate the conditions sketched in the previous section, conditions which allow the inputs and outputs to be characterized as strings of digits, performs non-digital computations (if it performs computations at all). This point has special relevance to neuroscience and psychology, where the question of whether the brain computes is central to many debates.

In the case of artificial neural networks, it is somewhat open to stipulation which of their properties count as inputs and outputs. We may consider only networks with properties that are conducive to classifying their inputs and outputs as strings of digits. We may choose to send inputs to our network all at once at well-defined times, consider only network dynamics that stabilize on stable states, group inputs and outputs into finitely many equivalence classes, and impose a well-defined order on the input and output units, so as to characterize inputs and outputs as strings of digits.

When we switch to studying natural systems such as brains, we must not import such stipulations into our theories without justification. Brain theories must be based on empirical evidence about which properties of neural activity are functionally significant, not on which putative properties can be conveniently characterized in terms of computability theory.

Earlier in Section 2, I pointed out that Rosenblatt, who first characterized his neural network theory as connectionist, was building on McCulloch and Pitts's work. McCulloch and Pitts, in turn, were building on work by the mathematical biophysics group led by Nicholas Rashevsky (cf. Piccinini 2004a).

Rashevsky and his collaborators, young Walter Pitts among them, studied the properties of certain idealized neural systems mathematically. They defined and studied neural systems using differential and integral equations that represented, among other variables, the frequency of neuronal pulses travelling through idealized neuronal fibers. They analyzed systems with different structural and functional properties, characterizing their behavior and offering explanations of phenomena such as discrimination, perception, reflexes, learning, and thinking (Rashevsky 1938, 1940; Householder and Landahl 1945). From a modern perspective, they should count as pioneers of computational neuroscience. Yet they never even claimed that their networks compute, much less that they perform digital computations.

The present account of computation makes sense of this fact. The mathematical tools Rashevsky and his collaborators used to analyze their networks didn't include logic or computability theory. Furthermore, the main variable in terms of which they characterized neuronal activity was the frequency of neuronal pulses in continuous time—roughly corresponding to what today is called 'firing rate'. Since Rashevsky's time, theoretical neuroscientists have learned to study firing rates using more

sophisticated mathematical tools. Now as then, firing rates appear to be the most important neurophysiological variable. Now as then, theoretical and computational neuroscientists who study spike trains and firing rates (Dayan and Abbott 2001; Ermentrout and Terman 2010) make no significant use of logic or computability theory. This is not an accident: as in the case of continuous variables, there doesn't seem to be any clear or theoretically useful way to characterize firing rates as strings of digits. Nor is there any clear or theoretically useful way to characterize firing rates varying in continuous time as continuous variables like those manipulated by analog computers. If this is correct, then neural computations are neither digital nor analog computations; neural computations are *sui generis* (Piccinini and Bahar 2013).

If it turns out that neural processes are *sui generis*, some theories of cognition will have to change. Many theorists assume that cognition is digital computation (in the sense here explicated). This includes classicists, whose view is that cognition is classical computation, as well as many connectionists, whose view is that cognition is non-classical (but still digital) computation. Their theories postulate some digital computation or other to explain cognitive phenomena, but they usually gloss over the issue of how such computations are implemented in the brain. This may be ok as long as neural processes are (or "implement") digital computations: somehow, neural computations must implement the digital computations postulated by cognitive theorists.

But if neural computations are *sui generis*, theorists of cognition must learn how nervous systems work and formulate theories in terms that can be implemented by actual neural processes. There are those who have been doing this all along, and there are those who are moving in that direction. It may be time for the rest of the community to join them.

8. Conclusion

Nowadays, many authors use 'computation' loosely. They use it for any processing of signals, for "information processing," for whatever the brain does, or even more broadly. In these loose senses, all neural networks—brains and computers included—compute. But this is rather uninformative: it doesn't lead to applying the mathematical theory of computation to analyze neural networks, their power, and their limitations. They should not be confused with the interesting proposal first made by McCulloch and Pitts.

When McCulloch and Pitts claimed that their networks compute, their claim was strong and informative. They claimed that what we now call 'computability theory' would allow us to analyze and evaluate the power of neural networks. This is the kind of claim that I have explicated in this chapter.

If the above considerations are on the right track, several conclusions can be drawn. First, many neural networks perform *digital* computations: they manipulate strings of digits in accordance with a rule defined over the inputs (and possibly

internal state). This allows us to apply the mathematical theory of computation to them and evaluate their computation power. Second, some neural networks (such as McCulloch-Pitts networks) perform digital computations in a classical way: they compute by operating in accordance with a program (or algorithm) for generating successive strings of digits, one step at a time. Digital computers, which are very large networks of logic gates, belong in this class of neural networks. Third, other neural networks perform *digital* computations but in a *non-classical* way: they are trainable, turn their input into their output in virtue of their continuous dynamics (which cannot be broken down into intermediate computational steps), or both. Fourth, yet other neural networks (such as Rashevsky's networks) perform *non-digital computations*: their inputs and outputs are not strings of digits. Brains may well fall into this last class.

This chapter concludes my examination of some important classes of computing systems. The next three chapters will address two other important topics pertaining to concrete computation. In this and previous chapters, periodically we encountered the notions of information and information processing, which are often equated to computation. The next chapter argues that information processing entails computation but not vice versa. We also encountered the question whether some physical systems are computationally more powerful than Turing machines. I address that question in the last two chapters.

14

Information Processing

1. Computation = Information Processing?

Computation is typically used to process information, so much so that the notions of computation and information processing are often used interchangeably. This is built into some versions of the semantic account (Chapter 3), according to which there is no computation without representation. Here is a representative example: "I… describe the principles of operation of the human mind, considered as an *information-processing, or computational*, system" (Edelman 2008, 10, emphasis added). This statement presupposes that computation is the same as information processing. Is this right?

There are two immediate complications. First, as I argued beginning with Chapter 3, computation does not, in fact, require representation—computation can occur in the complete absence of representation. Second, there are several notions of both computations and information. As we have seen in previous chapters, computation comprises importantly different notions: computation in the generic sense, digital computation, analog computation, etc. The same is true of information. This chapter is devoted to distinguishing different notions of information and assessing the assimilation of computation to information processing.[1]

Information plays a central role in many disciplines. In the sciences of mind, information is invoked to explain cognition and behaviour (e.g., Miller 1951; Minsky 1968). In communication engineering, information is central to the design of efficient communication systems such as television, radio, telephone networks, and the internet (e.g., Shannon 1948; Fano 1961; Pierce 1980). A number of biologists have suggested explaining genetic inheritance in terms of the information carried by DNA sequences (e.g., Smith 2000; see also Godfrey-Smith 2000; Griffiths 2001). Animal communication theorists routinely characterize non-linguistic communication in terms of shared information (Bradbury and Vehrencamp 2000). Some philosophers maintain that information can provide a naturalistic grounding for the intentionality of mental states, namely, their being *about* states of affairs (Dretske 1981; Millikan

[1] This chapter derives mostly from Piccinini and Scarantino 2011, so Andrea Scarantino deserves partial credit for most of what is correct here.

2004). Finally, information plays important roles in several other disciplines, such as computer science, physics, and statistics.

To account for the different roles information plays in all these fields, more than one notion of information is required. I begin this chapter by distinguishing between three main notions of information: Shannon's non-semantic notion plus two notions of semantic information. After that, I'll go back to the relation between information processing and computation.

2. Shannon information

I use 'Shannon information' to designate the notion of information initially formalized by Claude Shannon (Shannon 1948).[2] I distinguish between information theory, which provides the formal definition of information, and communication theory, which applies the formal notion of information to engineering problems of communication. I begin by introducing two measures of information from Shannon's information theory. Later I will discuss how to apply them to the physical world.

Let X and Y be two discrete random variables taking values in $A_X=\{a_1,\ldots,a_n\}$ and $A_Y=\{b_1,\ldots,b_r\}$, respectively, with probabilities $p(a_1),\ldots,p(a_n)$ and $p(b_1),\ldots,p(b_r)$. We assume that $p(a_i)>0$ for all $i=1,\ldots,n$, $p(b_j)>0$ for all $j=1,\ldots,r$, $\sum_{i=1}^{n}p(a_i)=1$ and $\sum_{j=1}^{r}p(b_j)=1$.

Shannon's key measures of information are the following:

$$H(X)=-\sum_{i=1}^{n}p(a_i)\log_2 p(a_i)$$

$$I(X;Y)=\sum_{i=1,j=1}^{n,r}p(a_i,b_j)\log_2\frac{p(a_i\mid b_j)}{p(a_i)}$$

H(X) is called 'entropy', because it's the same as the formula for measuring thermodynamic entropy; here, it measures the *average information* produced by the selection of values in $A_X=\{a_1,\ldots,a_n\}$.[3] We can think of H(X) as a weighted sum of n expressions of the form $I(a_i)=-\log_2 p(a_i)$. $I(a_i)$ is sometimes called the 'self-information' produced when X takes the value a_i. Shannon obtained the formula for *entropy* by setting a number of mathematical desiderata that any satisfactory measure of uncertainty

[2] Shannon was building on important work by Boltzmann, Szilard, Hartley, and others (Pierce 1980).

[3] The logarithm to the base b of a variable x—expressed as $\log_b x$—is defined as the power to which b must be raised to get x. In other words, $\log_b x=y$ if and only if $b^y=x$. The expression $0\log_b 0$ in any of the addenda of H(X) is stipulated to be equal to 0. Shannon (1948, 379) pointed out that in choosing a logarithmic function he was following Hartley (Hartley 1928) and added that logarithmic functions have nice mathematical properties, are more useful practically because a number of engineering parameters "vary linearly with the logarithm of the number of possibilities," and are "nearer to our intuitive feeling as to the proper measure" of information.

should satisfy, and showing that the desiderata could only be satisfied by the formula given above.[4]

Shannon information may be measured with different units. The most common unit is the "bit."[5] A bit is the information generated by a variable, such as X, taking a value a_i that has a 50 percent probability of occurring. Any outcome a_i with less than 50 percent probability will generate *more* than 1 bit; any outcome a_i with more than 50 percent probability will generate *less* than 1 bit. The choice of unit corresponds to the base of the logarithm in the definition of entropy. Thus, the information generated by X taking value a_i is equal to 1 bit when $I(a_i)=-\log_2 p_i=1$; that is, when $p_i=0.5$.[6]

Entropy has a number of important properties. First, it equals zero when X takes some value a_i with probability 1. This tells us that information as entropy presupposes uncertainty: if there is no uncertainty as to which value a variable X takes, the selection of that value generates no Shannon information.

Second, the closer the probabilities $p_1, \ldots, p_n$ are to having the same value, the higher the entropy. The more uncertainty there is as to which value will be selected, the more information is generated by the selection of a specific value.

Third, entropy is highest and equal to $\log_2 n$ when X takes every value with the same probability.

I(X;Y) is called 'mutual information'; it is the difference between the entropy characterizing X, on average, *before* and *after* Y takes values in A_Y. Shannon proved that X carries mutual information about Y whenever X and Y are statistically dependent, i.e., whenever it is not the case that $p(a_i, b_j)=p(a_i)p(b_j)$ for all i and j. This is to say that the transfer of mutual information between two sets of uncertain outcomes $A_X=\{a_1,\ldots,a_n\}$ and $A_Y=\{b_1,\ldots,b_r\}$ amounts to the *statistical dependency* between the occurrence of outcomes in A_X and A_Y. Information is *mutual* because statistical dependencies are symmetrical.

Shannon's measures of information have many applications. The best-known application is Shannon's own: communication theory. Shannon was looking for an optimal solution to what he called the "fundamental problem of communication" (Shannon 1948, 379), that is, the reproduction of messages from an information source to a destination. In Shannon's sense, any device that produces messages in a stochastic manner can count as an information source or destination.

[4] The three mathematical desiderata are the following: (i) The entropy H should be continuous in the probabilities p_i, (ii) The entropy H should be a monotonic increasing function of n when $p_i=1/n$, and (iii) If $n=b_1+\ldots+b_k$ with b_i positive integer, then $H(1/n,\ldots,1/n)=H(b_1/n,\ldots,b_k/n)+\sum_{i=1}^{k} b_i/n\, H(1/b_i,\ldots,1/b_i)$. Shannon further supported his interpretation of H as the proper measure of information by demonstrating that the channel capacity required for most efficient coding is determined by the entropy (Shannon 1948; see Theorem 9 in Section 9).

[5] Shannon credits John Tukey, a computer scientist at Bell Telephone Laboratories, with introducing the term in a 1947 working paper.

[6] $-\log_2 0.5=1$.

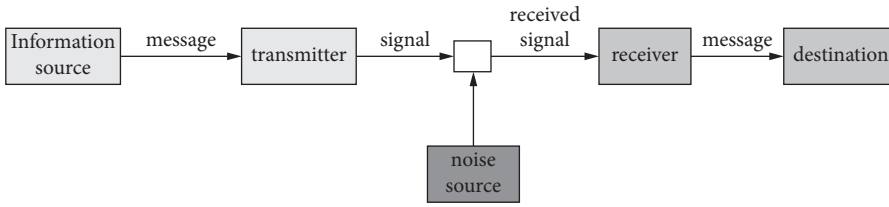


Figure 14.1 Shannon's depiction of a *communication system* (Shannon 1948).

Shannon distinguished three phases in the communication of messages: (a) the *encoding* of the messages produced by an *information source* into channel inputs, (b) the *transmission* of signals across a channel with random disturbances generated by a *noise source*, (c) the *decoding* of channel outputs back into the original messages at the *destination*.

The process is summarized by the picture of a *communication system* shown in Figure 14.1.

Having distinguished the operations of *coding, transmission* and *decoding*, Shannon developed mathematical techniques to study how to perform them efficiently. Shannon's primary tool was the concept of *information*, as measured by entropy and mutual information.

A common misunderstanding of Shannon information is that it is a semantic notion. This misunderstanding is promoted by the fact that Shannon liberally referred to *messages* as carriers of Shannon information. But Shannon's notion of message is not the usual one. In the standard sense, a message has semantic content or meaning—there is something it stands for. By contrast, Shannon's messages need not have semantic content at all—they need not stand for anything.[7]

I will continue to discuss Shannon's theory using the term 'message,' but you should keep in mind that the usual commitments associated with the notion of message do not apply. The identity of a communication-theoretic message is fully described by two features: it is a physical structure distinguishable from a set of alternative physical structures, and it belongs to an exhaustive set of mutually exclusive physical structures selectable with well-defined probabilities.

Under these premises, to *communicate* a message produced at a source amounts to generating a second message at a destination that replicates the original message so as to satisfy a number of desiderata (accuracy, speed, cost, etc.). The sense in which the semantic aspects are irrelevant is precisely that messages carry information merely qua selectable, physically distinguishable structures associated with given probabilities. A seemingly paradoxical corollary of this notion of communication is that a

[7] Furthermore, Shannon's messages don't even have to be strings of digits of finitely many types; on the contrary, they may be continuous variables. I defined entropy and mutual information for discrete variables, but the definitions can be modified to suit continuous variables (Shannon 1948, Part III).

nonsense message such as "#r %h@" could in principle generate *more* Shannon information than a meaningful message such as "avocado".

To understand why, consider an experiment described by a random variable X taking values a_1 and a_2 with probabilities $p(a_1) = 0.9999$ and $p(a_2) = 0.0001$ respectively. Before the experiment takes place, outcome a_1 is almost certain to occur, and outcome a_2 almost certain not to occur. The occurrence of both outcomes generates information, in the sense that it resolves the uncertainty characterizing the situation before the experiment takes place.

The occurrence of a_2 generates *more information* than the occurrence of a_1, because it is *less expectable*, or *more surprising*, in light of the prior probability distribution. This is reflected by the measure of self-information we introduced earlier: $I(a_i) = -\log_2 p(a_i)$. Given this measure, the higher $p(a_i)$ is, the lower $I(a_i) = -\log_2 p(a_i)$ is, and the lower $p(a_i)$ is, the higher $I(a_i) = -\log_2 p(a_i)$ is. Therefore, if "#r %h@" is a more surprising message than "avocado", it carries more Shannon information, despite its meaninglessness.

Armed with his measures of non-semantic information, Shannon proved a number of seminal theorems that had a profound impact on the field of communication engineering. The "fundamental theorem for a noisy channel" established that it was theoretically possible to make the error rate in the transmission of information across a randomly disturbed channel as low as desired up until the point in which the source information rate in bits per unit of time becomes larger than the channel capacity, which is defined as the mutual information maximized over all input source probability distributions.[8] Notably, this is true regardless of the physical nature of the channel (e.g., true for electrical channels, wireless channels, water channels, etc.).

Communication theory has found many applications, most prominently in communication engineering, computer science, neuroscience, and psychology. For instance, Shannon information is commonly used by neuroscientists to measure the quantity of information carried by neural signals about a stimulus and estimate the efficiency of coding (what forms of neural responses are optimal for carrying information about stimuli) (Dayan and Abbott 2001, chap. 4; Baddeley et al. 2000). I now move on from Shannon information, which is non-semantic, to semantic information.

3. Semantic Information

Suppose that a certain process leads to the selection of signals. Think, for instance, of how you produce words when speaking. There are two dimensions of this process that are relevant for the purposes of information transmission. One is the uncertainty

[8] Shannon defined the channel capacity C as follows: $Max_{P(a)}\, I(X;Y) = Max_{P(a)}[H(X)–H(X|Y)]$. The conditional entropy is calculated as follows: $H(X|Y) = \sum_{i=1,j=1}^{n,r} p(a_i, b_j)\log_2 \frac{1}{p(a_i|b_j)}$.

that characterizes the speaker's word selection process as a whole—the probability that each word be selected. This is the non-semantic dimension that would interest a communication engineer and is captured by Shannon's theory. A second dimension of the word selection process concerns what the selection of each word *means* (in the context of a sentence). This would be the dimension of interest, for example, to a linguist trying to decipher a tribal language.

Broadly understood, semantic notions of information pertain to what a specific signal broadcast by an information source means. To address the semantics of a signal, it is neither necessary nor sufficient to know which other signals might have been selected instead and with what probabilities. Whereas the selection of, say, any of twenty-five equiprobable but distinct words will generate the same non-semantic information, the selection of each individual word will generate different semantic information depending on what that particular word means.

Semantic and non-semantic notions of information are both connected with the reduction of uncertainty. In the case of non-semantic information, the uncertainty has to do with which among many possible signals is selected. In the case of semantic information, the uncertainty has to do with which among many possible states of affairs is the case.

To tackle semantic information, let's begin with Grice's distinction between two kinds of meaning that are sometimes conflated, namely, natural and non-natural meaning (Grice 1957). Natural meaning is exemplified by a sentence such as 'those spots mean measles', which is true—Grice claimed—just in case the patient has measles. Non-natural meaning is exemplified by a sentence such as 'those three rings on the bell (of the bus) mean that the bus is full' (Grice 1957, 85), which is true even if the bus is not full.

I extend the distinction between Grice's two types of *meaning* to a distinction between two types of *semantic information*: *natural information* and *non-natural information*.[9] Spots carry natural information about measles because there is a reliable physical correlation between measles and spots. By contrast, the three rings on the bell of the bus carry non-natural information about the bus being full by virtue of a convention.[10] I will now consider each notion of semantic information in more detail.

3.1 Natural (Semantic) Information

When smoke carries information about fire, the basis for this informational link is the causal relation between fire and smoke. By the same token, when spots carry

[9] By calling this kind of information *non-natural*, I am not taking a stance on whether it can be naturalized, that is, reduced to some more fundamental natural process. I am simply using Grice's terminology to distinguish between two importantly different notions of semantic information.

[10] This is not to say that conventions are the only possible source of non-natural meaning. For further discussion, see (Grice, 1957).

information about measles, the basis for this informational link is the causal relation between measles and spots. Both are examples of natural semantic information.

The most basic task of an account of natural semantic information is to specify the relation that has to obtain between a source (e.g., fire) and a signal (e.g., smoke) for the signal to carry natural information about the source. Following Dretske (1981), I discuss natural information in terms of correlations between event types. On the view I propose, an event token *a* of type *A* carries natural information about an event token *b* of type *B* just in case *A reliably correlates* with *B*.

Reliable correlations are the sorts of correlations information users can count on to hold in some range of future and counterfactual circumstances. For instance, smoke-type events reliably correlate with fire-type events, spot-type events reliably correlate with measles-type events, and ringing-doorbell-type events reliably correlate with visitors-at-the-door-type events. It is by virtue of these correlations that we can dependably infer a fire-token from a smoke-token, a measles-token from a spots-token, and a visitors-token from a ringing-doorbell-token.

Yet correlations are rarely perfect. Smoke is occasionally produced by smoke machines, spots are occasionally produced by mumps, and doorbell rings are occasionally produced by naughty kids who immediately run away. Dretske (1981) and most subsequent theorists of natural information have disregarded imperfect correlations, either because they believe that they carry no natural information (Cohen and Meskin 2006) or because they believe that they do not carry the sort of natural information necessary for knowledge (Dretske 1981). Dretske focused only on cases in which signals and the events they are about are related by nomically underwritten perfect correlations.

This is unfortunate because, although not necessarily knowledge-causing, the transmission of natural information by means of imperfect yet reliable correlations is what underlies the central role natural information plays in the descriptive and explanatory efforts of many sciences. In other words, most of the natural information signals carry in real world environments is *probabilistic*: signals carry natural information to the effect that *o* is *probably G*, rather than natural information to the effect that, with nomic certainty, *o* is *G*. In the present account, all-or-nothing natural information—the natural information that *o* is *G* with certainty—is a special, limiting case of probabilistic natural information.

Unlike the traditional (all-or-nothing) notion of natural information, this probabilistic notion of natural information is applicable to the sorts of signals studied by many empirical sciences, including the science of cognition. Organisms survive and reproduce by tuning themselves to reliable but imperfect correlations between internal variables and environmental stimuli, as well as between environmental stimuli and threats and opportunities. In comes the sound of a predator, out comes running. In comes the redness of a ripe apple, out comes approaching.

But organisms do make mistakes, after all. Some mistakes are due to the reception of probabilistic information about events that fail to obtain.[11] For instance, sometimes non-predators are mistaken for predators because they sound like predators; or predators are mistaken for non-predators because they look like non-predators.

Consider a paradigmatic example from ethology. Vervet monkeys' alarm calls appear to be qualitatively different for three classes of predators: leopards, eagles, and snakes (Struhsaker 1967; Seyfarth and Cheney 1992). As a result, vervets respond to eagle calls by hiding behind bushes, to leopard calls by climbing trees, and to snake calls by standing tall.

A theory of natural information with explanatory power should be able to make sense of the connection between the reception of different calls and different behaviors. To make sense of this connection, we need a notion of natural information according to which types of alarm call carry information about types of predators. But any theory of natural information that demands perfect correlations between event types will not do, because as a matter of empirical fact when the alarm calls are issued the relevant predators are not always present.

The present notion of probabilistic natural information can capture the vervet monkey case with ease, because it only demands that informative signals *change the probability* of what they are about. This is precisely what happens in the vervet alarm call case, in the sense that the probability that, say, an eagle/leopard/snake is present is significantly higher given the eagle/leopard/snake call than in its absence.

Notably, this notion of natural information is graded: signals can carry more or less natural information about a certain event. More precisely, the higher the difference between the probability that the eagle/leopard/snake is present given the call and the probability that it is present without the call, the higher the amount of natural information carried by an alarm call about the presence of an eagle/leopard/snake.

This approach to natural information allows us to give an information-based explanation of why vervets take refuge under bushes when they hear eagle calls (*mutatis mutandis* for leopard and snake calls). They do because they have received an ecologically significant amount of probabilistic information that an eagle is present.

To sum up, an event's failure to obtain is compatible with the reception of natural information about its obtaining, just like the claim that the probability that o is G is high is compatible with the claim that o is not G. No valid inference rules take us from claims about the transmission of probabilistic information to claims about how things turn out to be.[12]

[11] This is not to say that natural information is enough to explain why acting on the basis of received natural information sometimes constitutes a mistake and sometimes it does not.

[12] A consequence of this point is that the transmission of natural information entails nothing more than the truth of a probabilistic claim (Scarantino and Piccinini 2010). It follows that the present distinction

3.2 Non-natural (Semantic) Information

Cognitive scientists routinely say that cognition involves the processing of information. Sometimes they mean that cognition involves the processing of natural information. At other times, they mean that cognition involves the processing of non-natural information. This second notion of information is best understood as the notion of *representation*, where a (descriptive) representation is by definition something that can get things wrong.[13]

Some cognitive scientists simply assimilate representation with *natural* semantic information, assuming in effect that what a signal represents is what it reliably correlates with. This is a weaker notion of representation than the one I endorse. Following the predominant usage in the philosophical literature, I reserve the term 'representation' for states that can get things wrong or misrepresent (e.g., Millikan 2004).

Bearers of natural information, I said, "mean" states of affairs in virtue of being physically connected to them. I provided a working account of the required connection: there must be a reliable correlation between a signal type and its source type. Whatever the right account may be, one thing is clear: in the absence of the appropriate physical connection, no natural information is carried.

Bearers of non-natural information, by contrast, need not be physically connected to what they are about in any direct way. Thus, there must be an alternative process by which bearers of non-natural information come to bear non-natural information about things they may not reliably correlate with. A convention, as in the case of the three rings on the bell of the bus, is a clear example of what may establish a non-natural informational link. Once the convention is established, error (misrepresentation) becomes possible.

The convention may either be explicitly stipulated, as in the rings case, or emerge spontaneously, as in the case of the non-natural information attached to words in natural languages. But non-natural information need not be based on convention (cf. Grice 1957). There may be other mechanisms, such as learning or biological evolution, through which non-natural informational links may be established. What matters for something to bear non-natural information is that, somehow, it stands for something else relative to a signal recipient.

An important implication of the present account is that semantic information of the non-natural variety does not entail truth. On the present account, false non-

between natural and non-natural meaning/information differs from Grice's original distinction in one important respect. On the present view, there is nothing objectionable in holding that *those spots carry natural information about measles, but he doesn't have measles*, provided measles are more likely given those spots than in the absence of those spots.

[13] There are also imperative representations, such as desires. And there are representations that combine descriptive and imperative functions, such as honeybee dances and rabbit thumps (cf. Millikan 2004). For simplicity, I focus on descriptive representations.

natural information is a genuine kind of information, even though it is epistemically inferior to true information. The statement 'water is not transparent' gets things wrong with respect to the properties of water, but it does not fail to represent that water is not transparent. By the same token, the statement 'water is not transparent' contains false non-natural information to the effect that water is not transparent.

Most theorists of information have instead followed Dretske (Dretske 1981; Millikan 2004; Floridi 2005) in holding that false information, or misinformation, is not really information. This is because they draw a sharp distinction between information, understood along the lines of Grice's natural meaning, and representation, understood along the lines of Grice's non-natural meaning.

The reason for drawing this distinction is that they want to use natural information to provide a naturalistic reduction of representation (or intentionality). For instance, according to some teleosemantic theories the only kind of information carried by signals is of the natural variety, but signals come to *represent* what they have the function of carrying natural information about (e.g., Millikan 1984). According to theories of this sort, what accounts for how representations can get things wrong is the notion of biological function: representations get things wrong whenever they fail to fulfill their biological function.

But my present goal is not to naturalize intentionality. Rather, my goal is to understand the central role played by information and computation in computer science and cognitive science. If scientists used 'information' only to mean natural information, I would happily follow the tradition and speak of information exclusively in its natural sense. The problem is that the notion of information as used in the special sciences often presupposes representational content.

Instead of distinguishing sharply between information and meaning, I distinguish between natural information, understood roughly along the lines of Grice's natural meaning (cf. Section 3.1), and non-natural information, understood along the lines of Grice's non-natural meaning. We lose no conceptual distinction, while we gain an accurate characterization of how the concept of information is used in the sciences. This is a good bargain.

We are now in a position to refine the sense in which Shannon information is non-semantic. Shannon information is non-semantic in the sense that Shannon's measures of information are not measures of semantic content or meaning, whether natural or non-natural. Shannon information is also non-semantic in the sense that, as Shannon himself emphasized, the signals studied by communication theory need not carry any non-natural information.

But the signals studied by communication theory always carry natural information about their source when there is mutual information in Shannon's sense between source and receiver. When a signal carries Shannon information, there is by definition a reliable correlation between source events and receiver events. For any receiver event, there is a nonzero probability of the corresponding source event. And as I defined natural information, *A* carries natural information about *B* just in

case *A* reliably correlates with *B*—thus, receiver events carry natural information about source events whenever there is mutual information between the two. In addition, in virtually all practical applications of communication theory (as opposed to information theory more generally), the probabilities connecting source and receiver events are underwritten by a causal process.

For instance, ordinary communication channels such as cable and satellite television rely on signals that causally propagate from sources to receivers. Because of this, the signals studied by communication theory, in addition to producing a certain amount of mutual information in Shannon's sense based on the probability of their selection, also carry a certain amount of natural semantic information about their source (depending on how reliable their correlation with the source is). Of course, in many cases, signals also carry non-natural information about things other than their sources—but that is a contingent matter, to be determined case by case.

4. Other Notions of Information

There are other notions of information, such as *Fisher information* (Fisher 1935) and information in the sense of *algorithmic information theory* (Li and Vitányi 1997). There is even an all-encompassing notion of physical information. Physical information may be generalized to the point that every state of a physical system is defined as an information-bearing state. Given this all-encompassing notion, allegedly the physical world is, at its most fundamental level, constituted by information (Wolfram 2002; Lloyd 2006). If computation is defined as information processing in this sense, then this is just a variant formulation of ontic pancomputationalism, which I rejected in Chapter 4, Section 4.

In one or more of these senses of the term, computing entails processing information (cf. Milkowski 2013, 42–51). But these notions of information are not directly relevant to our present concerns, so I set them aside.

Nir Fresco and Marty Wolf have recently introduced an interesting notion of *instructional information* (Fresco 2013, Chapter 6; Fresco and Wolf 2014). Fresco and Wolf 's formulation relies on notions of data and meaning that I don't fully understand, so I'll offer an alternative account that captures the gist of what they are after. Let *s* be a control signal affecting system *C*:

> *s* carries *instructional information e* within system *C* just in case when *C* acts in accordance with *s, C* generates action *e*.

Notice that *s* may be but *doesn't have to be* a string of digits that acts as an instruction in the sense of Chapter 9. Signal *s* is *any* signal external to *C* that activates a capacity of *C*. For example, consider a Boolean circuit that can perform two different operations depending on a single control digit. That single control digit counts as carrying instructional information. In any case, acting in accordance with *s* means doing what *s* causes within the system (i.e., action *e*), which may be seen as what *s*

says to do. Thus, Fresco and Wolf 's notion of instructional information gives rise to a more inclusive notion of instruction than the one discussed in Chapter 9. Fresco and Wolf maintain that "nontrivial" digital computation is the processing of discrete data in accordance with instructions in their sense. Since discrete data in Fresco and Wolf 's sense are strings of digits and Fresco and Wolf 's instructions are a kind of rule, Fresco and Wolf 's account of "nontrivial" digital computation appears to be a digital version of the causal account of computation (Chapter 2) restricted to systems that respond to control signals. If we formulate Fresco and Wolf 's account in terms of functional mechanisms (Chapter 6), then their account is equivalent to the mechanistic account of *digital* computations that respond to control signals (Chapter 7).

In what follows, I focus on the relations between computation and the following three notions of information: Shannon information (information according to Shannon's theory), natural information (truth-entailing semantic information based on a physical connection between signal and source), and non-natural information (non-truth-entailing semantic information).

The differences between these three notions can be exemplified as follows. Consider an utterance of the sentence 'I have a toothache'. It carries Shannon information just in case the production of the utterance is a stochastic process that generates words with certain probabilities. The same utterance carries natural information about having a toothache just in case utterances of that type reliably correlate with toothaches. Carrying natural information about having a toothache entails that a toothache is more likely given the signal than in the absence of the signal. Finally, the same utterance carries non-natural information just in case the utterance has non-natural meaning in a natural or artificial language. Carrying non-natural information about having a toothache need not raise the probability of having a toothache.

5. How Computation and Information Processing Fit Together

Do the vehicles of computation necessarily bear information? Is information processing necessarily carried out by means of computation? What notion of information is relevant to the claim that a computational system is an information processor? Answering these questions requires combining computation and information in several of the senses I have discussed.

I will focus on two main notions of computation, digital and generic, and three main notions of information processing: processing Shannon information, processing natural semantic information and processing non-natural semantic information. It's time to see how they fit together.

I will argue that the vehicles over which computations are performed may or may not carry information. Yet, as a matter of contingent fact, the vehicles over which

most computations are performed generally do carry information, in several senses of the term. I will also argue that information processing must be carried out by means of computation in the generic sense, although it need not be carried out by computation in any more specific sense.

Before I begin, I should emphasize that there is a trivial sense in which computation entails the processing of Shannon information. I mention it now but will immediately set it aside because it is not especially relevant to computer science and cognitive science.

Consider an electronic digital computer. Each of its memory cells is designed to stabilize on one of two thermodynamically macroscopic states, called '0' and '1'. Suppose that, as soon as the computer is switched on, each memory cell randomly goes into one of those states. A nonzero Shannon entropy is associated with this state initialization process, since it is a random process, and manipulation of randomly generated initial states by the computer amounts to the processing of Shannon information in this trivial sense.

Note, however, that the digits created by random initialization of the memory cells carry no non-natural semantic information, since the digits in question do not have any conventional semantic content. They carry no functionally relevant natural semantic information either, since they do not reliably correlate with any source other than the thermal component of the electrons' dynamics in the memory cell at power up. Random noise is not the kind of source of natural information that is relevant to the way computers are used or to the explanation of cognition. Thus, the digits do not carry any natural semantic information beyond the theoretically unimportant natural information they carry about what caused them—whatever that may be.

Since a computer manipulates its digits and, in the above sense, a nonzero Shannon information is associated with randomly generated digits, it follows that a computer processing randomly generated digits is processing information-bearing states. In this trivial sense, such a computer is necessarily a processor of Shannon information. But here I am not interested in whether computation can be regarded as processing information in this trivial sense. I am interested in how computation fits with the processing of Shannon information, natural semantic information, and non-natural semantic information *about sources in the system's distal environment*—the sources the system needs to respond to. I will take up each of these in turn.

5.1 Computation and the Processing of Shannon Information

The notion of *processing* Shannon information is not entirely clear. Shannon information is not a property of individual signals, which can be manipulated as the signal is manipulated. Instead, Shannon entropy and mutual information quantify statistical properties of a selection process and a communication channel as a whole.

The first thing to note is that, strictly speaking, computational vehicles need not *carry* Shannon information at all. This is because Shannon information

requires random variables—deterministic variables carry no Shannon information whatsoever—whereas computation is compatible with deterministic variables. For instance, consider again a digital computer. For its inputs to be regarded as carriers of Shannon information, they have to be selected in a stochastic way. But the computer works just the same whether its inputs are selected deterministically or probabilistically. The same point can be made about other computing devices, such as analog computers and neural networks. Thus, computation does not entail the processing of Shannon information.

Another relevant observation is that computation need not *create* Shannon information. This is because computation may be either probabilistic or deterministic—in most real-world applications, computation is deterministic. Deterministic computation cannot create Shannon information—it can only conserve or destroy it. More precisely, Shannon information is conserved in logically reversible computations (which are necessarily deterministic), is reduced in logically irreversible deterministic computations, and may be created, conserved, or destroyed in noisy computations.

These points should not obscure the fundamental importance of communication theory to the understanding of computing systems. Even though, strictly speaking, computing systems need not process Shannon information, it is useful to assume that they do. That is, it is useful to characterize the inputs, internal states, and outputs of computing systems and their components stochastically and analyze system performance and resource requirements using information-theoretic measures and techniques. On this view, the components of computing systems may be treated as sources and receivers in the sense of communication theory (Winograd and Cowan 1963). This allows engineers to design efficient computer codes and effective communication channels within computing systems. Similar techniques may be used to analyze neural codes and signal transmission within the nervous system.

So far, I have discussed whether computation is the processing of Shannon information. What about the converse? Processing Shannon information may or may not be done by means of digital computation. First, for Shannon information processing to be done by digital computing, the information must be produced and carried by strings of digits. But Shannon information can also be produced and carried by continuous signals, which may be processed by analog means. Second, even when the bearers of Shannon information are digits, there exist forms of processing of such digits other than digital computation. Just to give an example, a digital-to-analog converter transforms digits into analog signals.

Shannon information processing is, however, a form of computation in the generic sense. As I defined it, generic computation is the functional manipulation of any medium-independent vehicle. Shannon information is a medium-independent notion, in the sense that whether Shannon information can be associated with a given vehicle does not depend on its specific physical properties, but rather on its probability of occurrence relative to its physically distinguishable alternatives. Thus, generic computation is broad enough to encompass the processing of Shannon

information. In other words, if a vehicle carries Shannon information, its processing is a computation in the generic sense.

The bottom line is this: although, strictly speaking, computation need not process vehicles that carry Shannon information, it usually does and, at any rate, it is useful to assume that it does. Conversely, the processing of Shannon information amounts to some form of generic computation. Communication theory places constraints on any kind of signal transmission (and hence on the processing of any information carried by the transmitted signals) within computing systems, including cognitive ones. This does not yet tell us what we are most interested in: how does computation relate to semantic information?

5.2 Computation and the Processing of Natural Information

Many people use 'computation' and 'information processing' interchangeably. What they often mean by 'information processing' is the processing of natural semantic information carried by the computation vehicles. This use of 'information processing' is common in the behavioral and brain sciences. It is thus important to examine whether and in what sense computation is the processing of natural information.

The notion of digital computation does not require that the computation vehicles carry natural information—or more precisely, digits are not required to carry natural information about the computing system's *distal* environment.[14] In this section, I will focus on whether digital computation entails the processing of natural information about the system's distal environment.

Granted, natural information is virtually ubiquitous—it is easy enough to find reliable correlations between physical variables. This being the case, the digits within a digital computer may well carry natural information about cognitively relevant sources, such as environmental stimuli that the computer responds to. Consider a car's computer, which receives and responds to natural information about the state of the car. The computer uses feedback from the car to regulate fuel injection, ignition timing, speed, etc. In such a case, a digital computation will be a case of natural information processing.

But digits—the vehicles of digital computation—need not correlate reliably with anything (in the distal environment) in order for a digital computation to be performed over them. Thus, a digital computation may or may not constitute the processing of natural information (about the distal environment). If the computation vehicles *do* carry natural information (about the distal environment), this may be quite important. In our example, that certain digits carry natural information about

[14] At least for deterministic computation, there is always a reliable causal correlation between later states and earlier states of the system as well as between initial states and whatever caused those initial states—even if what caused them are just the thermal properties of the immediate surroundings. In this sense, (deterministic) digital computation entails natural information processing. But the natural information that is always "processed" is not about the distal environment, which is what theorists of cognition are generally interested in.

the state of the car explains why the car's computer can regulate, say, fuel injection successfully. My point is simply that digital computation does not entail the processing of natural information about the distal environment.

This point is largely independent of the distinction between semantically interpreted and non-semantically-interpreted digital computation. A semantically interpreted digital computation is generally defined over the *non*-natural information carried by the digits—independently of whether they carry natural information (about the distal environment) or what specific natural information they carry. For instance, ordinary mathematical calculations carried out on a computer are defined over numbers, regardless of what the digits being manipulated by the computer reliably correlate with. Furthermore, it is possible for a digital computation to yield an incorrect output, which misrepresents the outcome of the operation performed. As we have seen, natural information cannot misrepresent. Thus, even semantically interpreted digital computation does not entail the processing of natural information (about the distal environment). (More on the relation between natural and non-natural information in the next sections.)

This point generalizes to generic computation, which includes, in addition to digital computation, at least analog computation. Nothing in the notion of analog computation mandates that the vehicles being manipulated carry natural information (about the distal environment). More generally, nothing in the notion of manipulating medium-independent vehicles mandates that such vehicles carry natural information (about the distal environment). Thus, computation may or may not be the processing of natural information.

What about the converse claim: that the processing of natural information must be carried out by means of computation? The answer is that yes, it must, although the computation need not be digital or of any specific kind. Natural information may be encoded by continuous variables, which cannot be processed by digital computers unless they are first encoded into strings of digits. Or it may be encoded by strings of digits, whose processing consists in converting them to analog signals. In both of these cases, natural information is processed, although not by means of digital computation.

But if natural information is processed, some kind or other of generic computation must be taking place. Natural information, like Shannon information, is a medium-independent notion. As long as a variable reliably correlates with another variable, it carries natural information about it. Any other physical properties of the variables are immaterial to whether they carry natural information. Since generic computation is just the processing of medium-independent vehicles, any processing of natural information amounts to a computation in the generic sense.

In sum, computation vehicles need not carry natural information, but natural information processing is necessarily some kind of generic computation—though not necessarily either digital or analog computation.

5.3 *Computation and the Processing of Non-natural Information*

Non-natural information is a central notion in our discourse about minds and computers. We attribute representational contents to each other's minds. We do the same with the digits manipulated by computers. I will now examine whether and in what sense computation is the processing of non-natural information.

Digital computation may or may not be the processing of non-natural information, and vice versa. This case is analogous to that of natural information, with one difference. In the case of *semantically interpreted* digital computation—digital computation that has semantic content by definition—most notions of semantic content turn digital computation into non-natural information processing. (The primary exception are notions of semantic content based solely on natural information; but since natural information is not representational, such notions would be inadequate to capture the main notion of semantic content—representational semantic content.) As a matter of fact, almost all (digital) computing conducted in artifacts *is* information processing, at least in the ordinary sense of *non-natural* information.

But the main question I am trying to answer is whether *necessarily*, digital computation is the processing of non-natural information. It *is* if we define a notion of digital computation that entails that the digits are representations. But as I have repeatedly pointed out, nothing in the notion of digital computation used by computer scientists and computability theorists mandates such a definition. The theoretically most important sense of digital computation is the one implicitly defined by the practices of computability theory and computer science, and which provides a mechanistic explanation for the behavior of digital computers. In this theoretically most important sense of digital computation, digital computation need not be the processing of non-natural information.

Conversely, the processing of non-natural information need not be carried out by means of digital computation. Whether it is depends on whether the non-natural information is digitally encoded and how it is processed. It may be encoded either digitally, by continuous variables, or by other vehicles. Even when encoded digitally, non-natural information may be manipulated by digital computation or by other processes, as in digital-to-analog conversion.

Generic computation may or may not constitute the processing of non-natural information, for similar reasons. Nothing in the notion of analog computation mandates that the vehicles being manipulated carry non-natural information. They could be entirely meaningless and the computations would proceed just the same. More generally, nothing in the notion of manipulating medium-independent vehicles mandates that such vehicles carry non-natural information.

As in the case of natural information, the converse does not hold. Instead, non-natural semantic information processing is necessarily done by means of computation in the generic sense. For non-natural semantic information is medium-independent—it's defined in terms of the association between an information-carrying

vehicle and what it stands for, without reference to any physical properties of the vehicles. Since generic computation is defined as the processing of medium-independent vehicles, the processing of non-natural semantic information (i.e., the processing of vehicles carrying non-natural semantic information) is computation in the generic sense.

Our work is not done yet. I defined non-natural information in terms of the association between bearers of non-natural information and what they stand for. This representation relation requires explication.

In recent decades, philosophers have devoted a large literature to representation. They have articulated several theoretical options. One option is anti-naturalism: representation cannot and need not be explained or reduced in naturalistic or mechanistic terms—the representational character of cognition simply shows that psychology and perhaps neuroscience are *sui generis* sciences. A second option is anti-realism: representation is not a real (i.e., causally efficacious and explanatory) feature of cognition, although ascriptions of non-natural information-bearing states may be heuristically useful (Egan 2010). Finally, naturalists look for explanations of representation in non-intentional terms (Rupert 2008).

A prominent family of naturalistic theories attempts to reduce non-natural information to natural information—plus, perhaps, some other naturalistic ingredients. As I mentioned earlier, several efforts have been made to this effect (Dretske 1981; Millikan 2004; Dretske 1988; Barwise and Seligman 1997; Fodor 1990). According to these theories, non-natural information is, at least in part, natural information. But theories according to which non-natural information reduces to natural information are only one family of naturalistic theories among others. According to other naturalistic theories, non-natural information reduces to things other than natural information (e.g., Millikan 1984; Harman 1987; Papineau 1987; Grush 2004; Ryder 2004).

Summing up the contents of this section, I have reached two main conclusions. First, computation may or may not be the processing of non-natural information—with the exception of semantically interpreted computation, which is *not* the core notion of computation. Second, the processing of non-natural information is a kind of generic computation.

6. Computation is Not Information Processing

In this chapter, I have charted some conceptual relations between computation and information processing. 'Computation' and 'information processing' are commonly used interchangeably, which presupposes that they are roughly synonymous terms. If that were true, computation would entail information processing, and information processing would entail computation. Alas, things are not so simple. Each of the two assimilated notions has more than one meaning, and the entailment relations do

not hold on many legitimate ways of understanding computation and information processing.

Here are my main conclusions:

Question: Is digital computation the same as information processing?

Thesis 1: Digital Computation vs. Processing Shannon Information
(a) Digital computation does not entail the processing of Shannon information;
(b) The processing of Shannon information does not entail digital computation.

Thesis 2: Digital Computation vs. Processing Natural Information
(a) Digital computation does not entail the processing of natural information (about the distal environment);
(b) The processing of natural information does not entail digital computation.

Thesis 3: Digital Computation vs. Processing Non-natural Information
(a) Non-semantically interpreted digital computation does not entail the processing of non-natural information;
(b) Semantically interpreted digital computation entails the processing of non-natural information;
(c) The processing of non-natural information does not entail digital computation.

Question: Is generic computation the same as information processing?

Thesis 4: Generic Computation vs. Processing Shannon Information
(a) Generic computation does not entail the processing of Shannon information;
(b) The processing of Shannon information entails generic computation.

Thesis 5: Generic Computation vs. Processing Natural Information
(a) Generic computation does not entail the processing of natural information (about the distal environment);
(b) The processing of natural information entails (semantically interpreted) generic computation.

Thesis 6: Generic Computation vs. Processing Non-natural Information
(a) Non-semantically interpreted generic computation does not entail the processing of non-natural information;
(b) Semantically interpreted generic computation entails the processing of non-natural information;
(c) The processing of non-natural information entails (semantically interpreted) generic computation.

These six theses summarize some of the relations of conceptual entailment, or lack thereof, between 'computation' and 'information processing', depending on what each notion is taken to mean.

Having dealt with the relation between computation and information processing, the final two chapters will address another pervasive and controversial topic: the computational power of physical systems.

15

The Bold Physical Church-Turing Thesis

One of the outstanding and most fascinating puzzles about physical computation is how powerful it is. In previous chapters, we occasionally encountered the suggestion that some physical systems might be computationally *more* powerful than Turing machines. This suggestion appears to violate the Church-Turing thesis, according to which Turing machines capture the limits of what is computable. This chapter and the next address the computational limitations of physical systems.

The Church-Turing thesis (CT) may be stated as follows:

> **CT**: Any function that is intuitively computable is computable by some Turing machine (Turing-computable for short). (Figure 15.1)

Or in Alan Turing's terms, CT pertains to functions that may be "naturally regarded as computable" (Turing 1936–7, 135).[1]

The *converse* of CT is the thesis that any Turing-computable function is intuitively computable. This is true in the sense that any competent human being (or digital computer) can carry out the computation in question as long as she has resources such as time, writing material, etc. She can do so because she can execute the instructions that make up the program that defines the Turing machine. It is irrelevant that for most inputs and programs, she lacks sufficient resources to complete the computation (or even to read the whole input and program, if they are long enough).

While the converse of CT is relatively easy to establish, CT itself is more difficult to assess. For starters, how should we explicate "intuitively" or "naturally regarded as"? In recent decades, the literature on CT has mushroomed. While some aspects of CT have become clearer, little consensus has emerged on how to formulate and evaluate CT. This Chapter and the next offer some suggestions on how to make progress.

Following an established recent trend, I distinguish between Mathematical CT, which is the thesis supported by the original arguments for CT, and Physical CT, which pertains to the computational limitations of physical processes. In

[1] Turing talked about computable numbers rather than functions, but that makes no difference for present purposes.

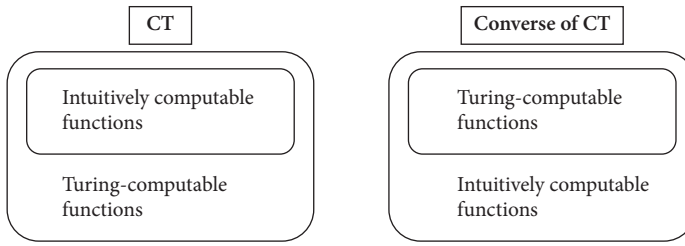


Figure 15.1 Venn diagrams representing The Church-Turing thesis and its converse.

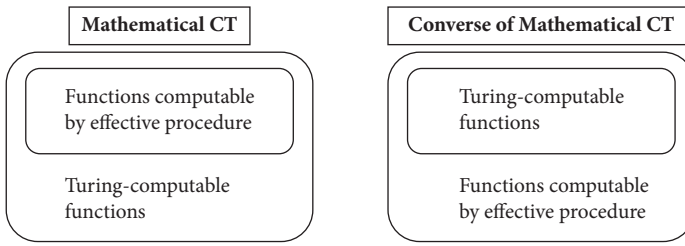


Figure 15.2 The Mathematical Church-Turing thesis and its converse.

addition, I distinguish between *bold* formulations of Physical CT, according to which any physical process—anything doable by a physical system—is computable by a Turing machine, and *modest* formulations, according to which any function that is *computable* by a physical system is computable by a Turing machine.

Mathematical CT: Any function that is computable by following an effective procedure is Turing-computable. (Figure 15.2)

Bold Physical CT: Any physical process is Turing-computable. (Figure 15.3)

Modest Physical CT: Any function that is physically computable is Turing-computable. (Figure 15.4)

The converses of these theses are easily established on the same grounds as the converse of (generic) CT. The converse of Mathematical CT is that any Turing-computable function is computable by following an effective procedure. The converse of Bold Physical CT is that any Turing-computable process can be physically realized. Finally, the converse of Modest Physical CT is that any Turing-computable function can be physically computed. All of these are true for the same reason that the converse of generic CT is true. The computational behavior of any Turing machine is exhaustively specified by an effective procedure consisting of Turing machine instructions. Any competent human being or digital computer can follow those instructions and carry out the computations as long as they have appropriate resources. While performing the computation, either a computing human or a digital computer physically realize the relevant Turing-computable

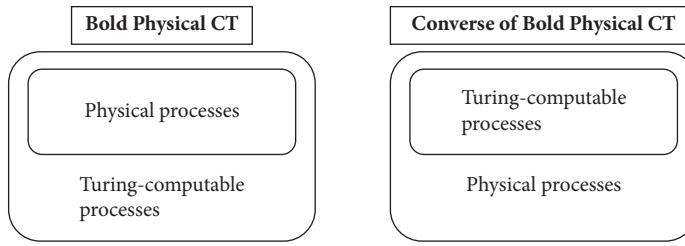


Figure 15.3 The Bold Physical Church-Turing thesis and its converse.

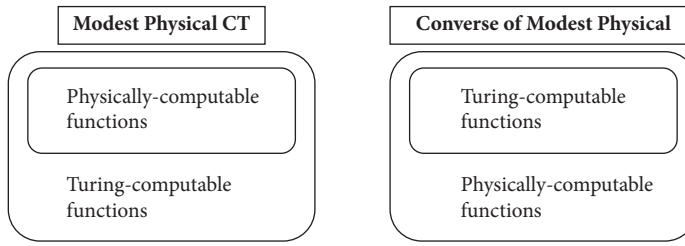


Figure 15.4 The Modest Physical Church-Turing thesis and its converse.

process. Although such physical realizations may be unable to *complete* most Turing-computable processes for lack of sufficient resources, this is beside the point.

Once again, however, the specialized versions of CT—especially the physical versions—are more difficult to assess. It's not even clear which version of CT is the correct physical analogue of Mathematical CT. Accordingly, this chapter and the next have two main purposes: to determine which version of Physical CT is the physical analogue of Mathematical CT and to determine whether any version of Physical CT is plausible.

In Section 1, a few remarks about Mathematical CT will establish that the notion of computability that grounds CT is an epistemological notion. In the case of Mathematical CT, it is the notion of what can be obtained by a finite observer exploiting procedures defined over a denumerable domain that are executable, automatic, uniform, and reliable. More precisely, a function defined over a denumerable domain (such as the natural numbers) is *computable* just in case a finite observer can find the function's values by a procedure that is executable, automatic, uniform, and reliable.

In Section 2, I will introduce and articulate a usability constraint on physical computation: for a physical process to count as a computation (and thus for it to be relevant to Physical CT properly so called), it must be usable by a finite observer to obtain the desired values of a function defined over a denumerable domain using a process that is executable, automatic, uniform, and reliable.

In Section 3, I will argue that Bold Physical CT is both implausible and irrelevant to the epistemological concerns that motivate CT properly so called—Bold Physical

CT is not about what a finite observer can discover about the desired values of a function. Thus, contrary to what some authors assume, Bold Physical CT is not suitable as a physical analogue of Mathematical CT.

1. The Mathematical Church-Turing Thesis

Many authors assume that the intuitive sense of 'computable' has to do with what can be computed by physical systems. This is too simplistic without some qualifications. CT was originally proposed and evaluated by a group of mathematical logicians, who were investigating the foundations of mathematics not the general properties of physical systems. They were concerned with what can be established by following certain procedures, namely, effective procedures for generating values of functions defined over effectively denumerable domains, such as the natural numbers or strings of letters from a finite alphabet. This notion of effective procedure was closely related to the notion of proof within a formal logical system (as explicated by, e.g., Church 1956, §7).

An effective procedure for evaluating a given function may be informally characterized as follows:

- *Executability*: The procedure consists of a *finite* number of *deterministic instructions* (i.e., instructions determining a unique next step in the procedure), which have *finite* and *unambiguous* specifications commanding the execution of a *finite* number of *primitive operations*.
- *Automaticity*: The procedure requires *no intuitions* about the domain (e.g., intuitions about numbers), no ingenuity, no invention, and no guesses.
- *Uniformity*: The procedure is the same for each possible argument of the function.[2]
- *Reliability*: If the procedure terminates, the procedure generates the *correct* value of the function for each argument after a *finite* number of primitive operations are performed.

To determine what can be accomplished by following effective procedures, different authors proposed and studied different formalisms, such as Turing machines, general recursive functions, and λ-definable functions. They suggested that these formalisms capture in a precise way the informal notion of what can be calculated by effective procedures. Since Alonzo Church and Alan Turing were the first to make this suggestion, Stephen Kleene dubbed it the *Church-Turing thesis*.[3]

[2] This condition rules out Kálmar's "arbitrary correct methods", which may change from one argument of a function to another (Kálmar 1959).

[3] For more on the history of computability theory, including more details on the epistemological concerns that motivated it, see Davis 1982; Shapiro 1983; Gandy 1988; Sieg 1994, 1997, 2005, 2006a, Soare 1999; Piccinini 2003a, Copeland 2006; Hodges 2006; Shagrir 2006a, and Copeland, Posy, and Shagrir 2013.

The logicians who initially proposed and endorsed CT did not base their defense of CT on the limitations of physical systems. Their main reasons were the following:

1) Lack of counterexamples. Every function known to be effectively calculable, and every operation for defining a function effectively from other functions, has been shown to be Turing-computable.
2) Failure of diagonalization. Diagonalization over Turing machines, which might be expected to yield a function that is not Turing-computable, does not lead outside the class of Turing-computable functions.
3) Confluence. A number of authors, working with different primitive notions, defined different mathematical formalisms and proposed that the functions definable within their formalism be identified as the class of effectively calculable functions. Some of the earliest formalisms are: general recursiveness (Gödel 1934), λ-definability (Church 1932; Kleene 1935), Turing-computability (Turing 1936–7), and reckonability (Gödel 1936). These notions have been proven to be extensionally equivalent to one another in the sense that any function that falls within one of these formally defined classes falls within all of them. Since the different formalisms have little in common and yet they all define the same class of functions, it is likely that the class of functions so defined is the intended one.[4]
4) Turing's argument. A Turing machine seems capable of reproducing any operation that a human being can perform while following an effective procedure (Turing's main argument for CT in his 1936–7, 135–6).

I will call these the *original arguments* for CT.[5]

It doesn't take a detailed analysis to see that the original arguments have little to do with the general properties of physical systems. The view that CT pertains directly to what can be computed by physical systems makes no sense of the fact that most logicians and theoretical computer scientists accept CT on the grounds of one or more of the original arguments. Such a view implies that logicians are confused as to the proper justification for CT. Of course, they might be confused in many ways, but it is implausible that they are confused that badly. It is more charitable to distinguish two versions of CT. One, *Mathematical CT*, is the version supported by the original arguments.

Mathematical CT: Any function that is computable by following an effective procedure is Turing-computable.

[4] The argument from confluence is the most popular in computer science, where CT is often formulated as the thesis that any new formalization of the notion of effectively calculable functions will yield the same old class—that of the Turing-computable functions (e.g., Newell 1980).

[5] The earliest systematic exposition of the original arguments is in Kleene 1952, § 62, § 67.

The other is the version that pertains to what can be done or computed by physical systems. It may be called *Physical CT* (following Pitowsky 1990).

During the last two decades or so, the distinction between Mathematical and Physical CT has become fairly well established, on grounds similar to those I discussed.[6] The distinction between Mathematical and Physical CT is an important clarification, but it by no means completes the task of understanding CT. If anything, it doubles the task. We now have to understand two theses instead of just one.

There is much to say about Mathematical CT. The two largest areas of disagreement are how to further explicate Mathematical CT and whether it is rigorously provable.[7] I will not address those debates, because I wish to focus on Physical CT. I will conclude this brief discussion of Mathematical CT by reiterating a few points that *should*[8] be uncontroversial. First, Mathematical CT—however it ought to be further explicated—is supported by the original arguments. Second, one or more of the original arguments are good enough to establish that Mathematical CT is true. Third, Mathematical CT is formulated in terms of functions defined over effectively denumerable domains. Therefore, it does not apply directly to functions defined over domains of larger cardinality. Fourth, the notion of computation in whose terms Mathematical CT is formulated is an epistemological notion: it is the notion of what can be established by following effective procedures defined over effectively denumerable domains. Fifth, Mathematical CT does not pertain directly to what can be computed by physical systems in general.

Well, then, what should we say about what can be computed by physical systems in general?

2. A Usability Constraint on Physical Computation

If Physical CT is to be analogous to Mathematical CT, it must be formulated in terms of a suitable notion of physical computation. The above considerations suggest that a physical process should not count as a computation unless a finite observer can use it to generate the desired values of a given function. This principle can be formulated as an explicit constraint on physical computation:

[6] E.g., Gandy 1980; Earman 1986; Odifreddi 1989; Pitowsky 1990; Mundici and Sieg 1995; Sieg 2002; Shagrir 1997; Copeland 2002b; Shagrir and Pitowsky 2003; Fitz 2006; Button 2009. For examples of people who conflate or fail to distinguish explicitly between Mathematical and Physical CT, see Churchland and Churchland 1990; Cleland 1993; Hogarth 1994; Siegelmann 1999; Smith 2006a.

[7] On how to explicate Mathematical CT, see Kleene 1952, 1987b; Gandy 1980; Shapiro 1981, 1993, 2013; Shagrir 2002; Sieg 2002, 2008, Copeland 2002b; Rescorla 2007; and Arkoudas 2008. On whether it's provable, see Gandy 1980; Mendelson 1990; Shapiro 1993, 2013; Sieg 1994, 1997; Folina 1998; Black 2000; Smith 2007; Dershowitz and Gurevich 2008; and Kripke 2013.

[8] For a recent though unsuccessful attempt to challenge the notion of effective procedure that grounds Mathematical CT, see Hogarth 2004. For a rebuttal, see Button 2009.

Usability Constraint: If a physical process is a computation, it can be used by a finite observer to obtain the desired values of a function.

The usability constraint is an epistemic constraint.[9] It can be made more precise by specifying what it means to be usable by a finite observer and who counts as a finite observer.

As to the latter question, one possibility is to include among finite observers human beings and any other intelligent beings of similarly bounded capacities. Under this notion of finite observer, the usability constraint is relevant to computer scientists, engineers, entrepreneurs, and consumers, whose aims are designing, building, selling, and using computers now or in the future. If we include among observers real or hypothetical creatures at regions of spacetime that are physically inaccessible to us, the usability constraint is also relevant to physicists whose aim is investigating the computational power of physical systems anywhere in spacetime.

Another possibility is to construe 'finite observer' more broadly, to include any functionally organized system whose behavior is influenced by a computation in an appropriate way. If nervous systems are computing systems, then observers in this extended sense are the bodies of organisms whose behavior is influenced by their own neural computations. Under this broader notion of finite observer, the usability constraint is relevant to scientists interested in the computational explanation of cognition.

The constraints I will propose may be applied to observers in either sense. Given that the special relationship between organisms and their nervous systems may require additional caveats, however, I will focus the discussion on finite observers understood under the former, more restricted construal.

As to what it means for a physical process to be usable by a finite observer, we need a physical counterpart to the notion of effective procedure that grounds Mathematical CT. As per Section 1 of this chapter, a finite observer can follow an effective procedure because an effective procedure is executable, automatic, uniform, and reliable. By analogy, in order to be usable by a finite observer, a physical process must be executable, automatic, uniform, and reliable. These requirements ought to be qualified to take into account the differences between effective procedures and physical processes more generally.

- An *executable* physical process is one that a finite observer can set in motion to generate the values of a desired function until it generates a readable result. This requires that the observer can discover which function is being computed, that the process's inputs and outputs[10] be readable by the observer, and that the finite

[9] The usability constraint is weaker than the verifiability constraint discussed, and rightly rejected, by Shagrir and Pitowsky 2003, 90–1.

[10] Do computations have to have inputs and outputs? The mathematical resources of computability theory can be used to define "computations" that lack inputs, outputs, or both. But the computations that

observer be able to construct the system that exhibits the process. An executable physical process is also a process that, like an effective procedure, in principle can be repeated if a finite observer wishes to run it again. Each of these aspects of executability will be spelled out as a usability sub-constraint.

- An *automatic* physical process is one that runs without requiring intuitions, ingenuity, invention, or guesses.
- A *uniform* physical process is one that doesn't need to be redesigned or modified for different inputs.
- Finally, a *reliable* physical process is one that generates results at least some of the time and, when it does so, its results are correct. Reliability will be spelled out further shortly.

These principles follow from the usability constraint by analogy with effective procedures. While automaticity and uniformity are essentially the same as their analogues for effective procedures, executability and reliability deserve to be explicated further. I suggest six sub-constraints: 1) Readable inputs and outputs; 2) process-independent rule: 3) repeatability; 4) settability; 5) physical constructibility; 6) reliability. The first five sub-constraints cash out executability; the sixth explicates reliability.

2.1. Readable inputs and outputs

The inputs and outputs of a computation must be readable. A quantity readable in this sense is one that can be measured to the desired degree of approximation, so as to be used as output, and either prepared or discovered by a finite observer to the desired degree of approximation, so as to be used as input. One important reason why computing technology is paradigmatically digital is that strings of digits are the only known inputs and outputs to be reliably readable without error.[11,12]

For an output to be readable in the intended sense, the computing system must have a recognizable halting state. As a counterexample, consider a machine that purportedly computes characteristic function *f*. Suppose that for any argument of *f*, the machine is guaranteed to output its correct value at some future time. The machine gives its output as a one or a zero on a display. Suppose further that at any time, the value on the display may change from a one to a zero or vice versa and

are generally relevant for applications are computations with both inputs and outputs. Here I focus on the ordinary case.

[11] This point is discussed in more detail in Chapter 7, especially Section 3.4. Terminological note: I use 'digit' to denote a concrete instantiation of a letter from a finite alphabet.

[12] Someone might ask, what about analog computers? Don't they have real-valued quantities as inputs and outputs? Actually, analog computers (in the sense of Pour-el 1974) are designed to yield the values of functions of real variables, which is not quite the same as functions of real-valued quantities. What analog computers do is useful for solving certain classes of equations. Their output is still a string of digits representing a real number to some degree of approximation. Analog computers and their relation to digital computers were discussed in Chapter 12.

there is no known time after which the result of the computation is definitive. It may seem that the output is readable, because one can see whether it's a zero or a one. But it is not readable in the relevant sense, because the user never knows whether she is reading the desired value of *f*.

2.2. Process-independent rule

In a genuine computation, the problem being solved (or equivalently, the function being computed) must be definable independently of the process of solving it (computing it). Computation is interesting precisely because it finds the values of functions defined independently of the processes that compute them. If we can't find out what function *f* is before we run process *P*, we are not going to learn anything interesting about *f*, such as the values we desire to know for selected arguments, by running *P*. Hence, we should not count *P* as a physical computation.

A classic example of Turing-uncomputable function is the halting function for Turing machines. It is an explicitly defined function from strings of digits to strings of digits: the function that returns '1' if Turing machine t halts on input x, '0' otherwise. Any putative machine that solves the halting problem by reliably generating (measurable) values of the function for any argument (of reasonable size) satisfies this second sub-constraint (as well as the first sub-constraint, readable inputs and outputs).[13]

2.3. Repeatability

For a physical process to count as a genuine computation, it must be in principle repeatable by any competent observer who wishes to obtain its results.[14] Of course, the very same sequence of particular concrete events cannot occur twice. But physical processes may be repeatable in the sense that the same sequence of states (as defined by, say, a set of dynamical equations) can occur either within the same physical system at different times or within two relevantly similar physical systems (e.g., two systems satisfying the same equations). So for a physical process to count as a computation, any competent observer must be able to set up a physical system that undergoes the relevant sequence of state transitions.

Unrepeatable processes can only be observed by the lucky few who happen to be in their presence. They cannot be used by others who might be interested in learning from them. If no one is present, no one can learn from an unrepeatable process. If someone is present but makes a mistake in recording a result, that mistake can never be corrected. But again, computation matters to us (and to the logicians who created

[13] The process-independent rule sub-constraint entails that modeling one physical system using another is not enough for computing. For example, testing a model of an airplane in a wind tunnel does not constitute a computation because there is no way to define the function being putatively computed apart from performing the modeling experiments.

[14] I owe this observation to Michael Rabin. I don't know whether he agrees with the rationale I offer for it.

computability theory) because we can learn from it. If we can't repeat a process when we need it, it is not computation, and hence it is irrelevant to an adequate formulation of Physical CT.

Like the other sub-constraints, repeatability must be taken with a grain of salt. A computing system may be able to perform only one computation (or part thereof) during its lifetime, but it should be possible to repeat the same computation on another system. Repeatability applies during ordinary conditions of use, whatever they may be. It does not entail that a computing system must work under all physical conditions, or that computation must be instantaneous to allow for cases in which users want a computation repeated immediately after it starts. Within ordinary conditions of use, it must be possible to repeat a process, or else the process does not count as a computation.[15]

2.4. *Settability*

An ordinary computing system is something that, within its limits, can compute any value of one or more functions. Universal computers can compute any value of any Turing-computable function until they run out of memory or time. For this to be the case, normally a user sets the system to its initial state and feeds it different arguments of the function being computed. When a new computation is needed, the system is *re*set to its initial state, where it may be given either the same or a different input. If the system is given the same input, it should yield—barring malfunctions—the same output. Thus, resettability plus repeatability of the input entails repeatability of the output. But resettability is not required for computation; settability is enough for present purposes. If a system is not even settable, so that a user cannot choose which value of the function is going to be generated in a given case, then that system does not count as computing.

2.5. *Physical constructibility*

If a system cannot be physically constructed, it may count as performing notional computations, but it is irrelevant to Physical CT. A system is physically constructible in the present sense just in case the relevant physical materials can be arranged to exhibit the relevant properties. The materials may well include entities that are very large, small, or distant, so long as they can be made to exhibit the relevant properties.

Suppose that firing n pulses of energy into a sunspot reliably and repeatedly causes $f(n)$ pulses of radiation to be emitted, for some nontrivial f. The system that includes the sunspots as well as the apparatus for emitting and receiving energy pulses counts

[15] Repeatability may also be needed to establish reliability (see Section 2.6). One standard way to check that a computer is functioning properly is to run the same computation more than once, making sure that each time the results are the same. (Of course, repeatability does not rule out all malfunctions, since the results of repeated computations could be the same but still wrong.)

as physically constructible in the present sense. By the same token, every computing technology exploits natural properties of existing materials.

Developing computing technology involves many engineering challenges. That is why we don't yet have (computationally nontrivial) DNA or quantum computers, in spite of their apparent possibility in principle. The practical obstacles facing these technologies may be surmountable. But suppose that a putative computing technology involved obstacles that are practically insurmountable, for principled physical reasons. For instance, a putative computer might require physically unattainable speeds, more matter-energy than the universe contains, or parts smaller than the smallest physical particles. Such a technology would not be relevant to our practical interests. What can't be physically constructed can't refute Physical CT.

Someone might object as follows. Consider Turing machines, with their tape of unbounded length. If the universe contains only a finite amount of matter-energy, then tapes of unbounded length might not be physically constructible. Furthermore, it seems unlikely that a finite user would ever be able to harness an infinite amount of matter-energy. Most likely, the best we can do is to construct finite approximations of Turing machines, such as our ordinary digital computers. But Turing machines are the very machines in terms of which CT is formulated. If Turing's idealization is legitimate, someone might conclude, other idealizations are legitimate too. Since we accept Turing machines as relevant to CT, we should also accept other machines, regardless of whether they are physically constructible.

Arguments along these lines may be found in the literature on physical computability.[16] They miss the way in which Turing machines are relevant to CT. All that CT says is that any function that is intuitively computable is computable by some Turing machine. Computability by Turing machines is the *upper bound* on what CT deems intuitively computable. And acting as an upper bound does not require being physically constructible.

Perhaps the objection against physical constructibility derives some allure from the frequent practice of lumping CT and its converse together and calling their conjunction "the Church-Turing thesis". Such a lumping may be innocuous in other contexts, but it is misleading here. If CT and its converse are lumped together, Physical CT is taken to be the thesis that the physically computable functions are the same as the Turing-computable ones. If we take this to be Physical CT and notice that Turing machines may not be physically constructible, we might find nothing wrong in considering whether other physically non-constructible machines extend the

[16] See Button (2009, 775–8) for a recent discussion. According to Button, the argument just rehearsed may be resisted if there is an appropriate notion of physical possibility according to which, roughly, for any computation by a Turing machine, there is a possible world containing enough physical resources to complete the computation. Although I side with Button against the argument in question, Button's request for a specialized notion of physical possibility, postulating enough physical resources for any Turing machine computation, is both ontologically onerous and ad hoc. The considerations to follow in the main text provide a simpler and more principled reply.

range of the physically computable, thereby refuting Physical CT. This would be a mistake. To see that it is, it helps to keep CT and its converse separate.

When we focus on what can be physically computed, we know that any machine with a finite memory, such as our digital computers, computes less than Turing machines (for lack of a literally unbounded tape). We may then ask whether there are physical means to compute as much as or more than what Turing machines can compute. But to surpass the power of Turing machines and falsify Physical CT, it is not enough to show that some *hypothetical* machine is more powerful than Turing machines. That's easy and yet says nothing about what can be *physically computed*. What we need to show is that some machine is both more powerful than Turing machines and physically *constructible*. Given that Turing machines themselves are unlikely to be physically constructible, this is quite a tall order.

In conclusion, neither CT nor its converse entail that Turing machines are physically constructible, and this is as it should be. But anyone who maintains that some physically computable functions are not Turing computable ought to show that the systems that compute such functions are physically constructible.

2.6. Reliability

Demanding machines that never break down would be unrealistic, but machines that never complete their computation successfully are not worth building. To be useful, a computing system must operate correctly long enough to yield (correct) results at least some of the time. For this to happen, the system's components must not break too often. In addition, the system's design must be such that noise and other external disturbances are generally insufficient to interfere with the results. The history of electronic computation illustrates this point nicely.

When large electronic computers were initially proposed, some skeptics argued that their vacuum tubes would break too often for the computers to be useful. The skeptics were proven wrong, but this was no small feat. Even so, early electronic computers broke often enough that they required full-time technicians devoted to fixing them. Since then, computers have become remarkably reliable; this contributes to making them useful and accessible to a large public. The case of early computers shows that yielding a correct result even a low percentage of the time is compatible with usefulness. But we should be wary of computer designs that are guaranteed to break, self-destroy, or destroy the user (see next chapter) before the results of the computation can be known. Those early skeptics had a point: if a machine is unreliable as a matter of principle, it is not worth building.

2.7. Conclusions

This list of constraints should be enough to illustrate the kind of property that is required for something to be useful for physical computing, and hence to be relevant to Physical CT. The constraints are not meant to be exhaustive; the list is open ended. For what makes a system useful for computing depends in part on how the physical

world is, and that is not a matter for philosophers to settle. The important point is that we are interested in computation because of what we (finite observers) can learn from it. If we can't learn anything from a putative computation, then that process is not relevant to Physical CT.

Another important point is that there is a close connection between the usability constraints and the mechanistic account of computation (Chapter 7). According to the mechanistic account, a computation is a process performed by a functional mechanism that manipulates appropriate vehicles in accordance with a rule defined over the vehicles (and possibly internal states). The processes performed by computing mechanisms satisfy the first four usability sub-constraints. For computing mechanisms follow a rule defined over the vehicles they manipulate; that means there is a process-independent rule (Sub-constraint 2). They must also have readable inputs and outputs (Sub-constraint 1) and their processes must be repeatable (Sub-constraint 3) and settable (Sub-constraint 4), or else they would fail to fulfill their function. In the other direction, any system whose processes satisfy the first four usability sub-constraints is a notional computing mechanism or at least may function as such. That is, it is a notional system that may be used to perform a computational function.

If a functional mechanism is also physically constructible (Sub-constraint 5) and reliable (Sub-constraint 6), then it is usable in practice to perform computations. And if a system is usable in practice to perform computations, it is definitely either a computing mechanism or something that functions as such.

3. The Bold Physical Church-Turing Thesis

To obtain a physical version of CT that we can evaluate, we need to do two things. First, we need to replace Mathematical CT's appeal to effective procedures with an appeal to physical processes. Second, we need this appeal to physical processes to be relevant to the epistemological notion of computation—the generation of values of functions over strings, the solution of mathematical problems, etc.—that motivates CT in the first place.

The simplest starting point is the following:

Bold Physical CT: Any physical process is Turing-computable.

This is not very precise, but we need to start here because in the literature, Physical CT is often formulated in roughly these terms. Bold Physical CT ranges over all physical processes—implicitly, it counts all physical processes as *computations*, or at least as something that can be described exactly (not just approximately) by a computation. This is reminiscent of pancomputationalism—the view that every physical system is computational. In Chapter 4, I argued that pancomputationalism is mostly due to a conflation between computational modeling and computational

explanation. I will now argue that Bold Physical CT suffers from closely related shortcomings.

Sometimes, versions of (Bold) Physical CT are formulated that are more clear and precise, and hence more clearly evaluable. The following is a list of representative formulations:

(A) Any physical process can be simulated by some Turing machine (e.g., Deutsch 1985; Wolfram 1985; Pitowsky 2002).

(B) Any function over denumerable domains (such as the natural numbers) that is computable by an idealized "computing machine" that manipulates arbitrary real-valued quantities (as defined by Blum et al. 1998) is Turing-computable.[17]

(C) Any system of equations describing a physical system gives rise to computable solutions (cf. Earman 1986; Pour-El 1999). A solution is said to be computable just in case given computable real numbers as initial conditions, it returns computable real numbers as values. A real number is said to be computable just in case there is a Turing machine whose output effectively approximates it.

(D) For any physical system S and observable W, there is a Turing-computable function f: $\mathbf{N} \rightarrow \mathbf{N}$ such that for all $t \in \mathbf{N}$, $f(t)=W(t)$ (Pitowsky 1990).

Each of (A)–(D) deserves to be discussed in detail. Here I will limit my discussion to a few general points to the effect that none of these formulations are adequate physical analogues of Mathematical CT. I will argue that Bold Physical CT is both too broad in its scope and too indifferent to epistemological considerations to be relevant to the notion of computability that motivates CT. In other words, Bold Physical CT fails to satisfy the usability constraint defended in Section 2. As a consequence, Bold Physical CT needs to be replaced by a more modest formulation of Physical CT, which pertains to what can be computed by physical means in a restricted sense that encompasses more than what can be accomplished by following effective procedures but does not include all physical processes.

[17] Thesis (B) is a special case. Unlike (A), (C), and (D), it does not range over all physical processes but is restricted to a certain specific "computations". Blum, Cucker, Shub and Smale (1998) set up a mathematical theory of "computation" over real-valued quantities, which they see as a fruitful extension of ordinary computability theory. They define idealized "computing machines" that perform addition, subtraction, multiplication, division, and equality testing as primitive operations on arbitrary real-valued quantities. They easily prove that such "machines" can "compute" all sets defined over denumerable domains by encoding their characteristic function as a real-valued constant (Blum et al. 1998, 405). Of course, most such sets are not Turing-computable; thus, (B) is far from true. Blum et al. do not discuss this result as a refutation of Physical CT. Nevertheless, Blum et al.'s result is often mentioned by critics of Physical CT, even though, as I will argue, Blum et al.'s "computations" are not computations in the epistemological sense that motivates CT and so should not be seen as counterexamples to Physical CT properly so called. Because of this, it is appropriate to include (B) as a version of Bold Physical CT. More on this later in this chapter.

3.1. *Lack of confluence*

One of the original arguments for Mathematical CT is that seemingly disparate notions (Turing machines, general recursive functions, etc.) turn out to be extensionally equivalent. This confluence is a major strength of Mathematical CT.

By contrast, different formulations of Bold Physical CT appear to be logically independent of one another. They appeal to disparate notions whose mutual connections are less than transparent. This state of affairs demonstrates, at the very least, that clarity and consensus on Physical CT are lacking. It also raises doubts on the widespread practice of formulating putative physical formulations of CT without introducing constraints on what counts as a relevant physical process. Should we accept any of these formulations as our physical formulation of CT? Which one? On what basis should we choose one over the others? If we are to choose one, presumably we need arguments to rule out the others as irrelevant (or less likely, to show that the others reduce to our preferred one). This already shows that before we settle on a formulation of Physical CT, there is work to do.

3.2. *Unconstrained appeals to real-valued quantities*

Many current physical theories assume that nature contains real-valued quantities that vary along a continuum. These may include the velocity of objects, the coordinates that define the position of their center of gravity in the spacetime manifold, and more. If these physical theories are correct, then many properties of many entities take arbitrary real numbers as their values. Therefore, systems of physical equations, whose simulations, solutions, or observables are appealed to, respectively, by (A), (C), and (D), involve arbitrary real numbers. In addition, (B) explicitly involves arbitrary real-valued quantities. So typical versions of Bold Physical CT involve, explicitly or implicitly, arbitrary real numbers.

But most real numbers in any continuous interval are Turing-uncomputable. In fact, there are only countably many Turing-computable numbers, while any continuous interval contains uncountably many real numbers. Thus, the probability that a randomly selected real-valued quantity is Turing-computable is zero. Therefore, if our physical theories are correct, most transformations of the relevant physical properties are transformations of Turing-uncomputable quantities into one another.

For instance, an object's change of speed, or even its simple change of spatial location may be transformations of one Turing-uncomputable real-valued quantity into another. A transformation of one Turing-uncomputable value into another Turing-uncomputable value is certainly a Turing-uncomputable operation, for the simple reason that there is not even a Turing-computable way of writing down the first value.[18] Therefore, it would seem that given many of our physical theories,

[18] Unless the first uncomputable value is already given and there is a computable operation returning the second value from the first. For example, if x is a Turing-uncomputable numeral written on an infinite tape, a Turing machine can easily calculate, say, $1+x$.

the physical world is chock-full of operations that outstrip the power of Turing machines. If this is correct, it falsifies Bold Physical CT.

But there is no reason to believe that a finite observer can use the Turing-uncomputable operations just mentioned to compute in the epistemological sense that motivates CT in the first place—to solve mathematical problems, to generate the values of desired functions for desired arguments. While Blum et al. (1998) may find it useful to define notional "machines" that manipulate arbitrary real-valued quantities, this kind of manipulation cannot be exploited by a finite observer.

In order to measure a real-valued quantity exactly at an arbitrary time, or even in order to measure the exact value of an arbitrary digit in the expansion of a real-valued quantity, an observer needs unbounded precision in measurement. Since the value of the variable may change over time, the observer also needs unbounded precision in the timing of the measurement. There is no reason to believe that such unbounded precision is available to a finite observer. Furthermore, there is no reason to believe that a finite observer can discover the exact value of a real-valued quantity, either computable or not, by means other than measurement. Finally, preparing a real-valued quantity with an arbitrary exact value also requires unbounded precision. Again, there is no reason to believe that unbounded precision in preparation is available to a finite observer.

As far as we know, finite observers are limited to preparing and measuring physical quantities to a finite degree of approximation. Thus, manipulations of arbitrary real-valued quantities should not count as computation—at least until someone shows how a finite observer can exploit them in practice. Bold Physical CT, which is falsified by such manipulations, is not interestingly analogous to Mathematical CT.[19]

3.3. *Falsification by Irrelevant Counterexamples*

Bold Physical CT encompasses all physical processes regardless of whether they can be used by a finite observer to generate the values of a function. As a consequence, Bold Physical CT is liable to be refuted by irrelevant counterexamples. By an 'irrelevant counterexample', I mean a process that is either obviously not physically implementable or useless for generating the values of a desired function. I will now argue that by encompassing all physical processes and thereby exposing itself to irrelevant counterexamples, Bold Physical CT abandons the epistemological notion of computation that motivated CT in the first place. Thus, it is not a good physical analogue of Mathematical CT.

[19] The point just made does not impugn analog computation in the standard sense of Pour-El (1974). Analog computation does not manipulate exact values of arbitrary real-valued quantities but rather continuous variables. Although a continuous variable may be assumed to take any real value within a relevant interval, a successful concrete analog computation requires only the measurement of real variables to some degree of approximation. No exact values of arbitrary real-valued quantities are exploited by an analog computation, so analog computation does not falsify Bold Physical CT (cf. Rubel 1989).

Specifically, formulations (A)-(D) would be falsified by a sequence generated by a random (i.e., nondeterministic) physical process. Consider a discrete random process, such as the decay of atoms from a radioactive sample, over an infinite period of time. Its output at regular time intervals is a string of digits—'0' if no atoms decay during a time interval, '1' if one or more atoms decay during a time interval. A simple cardinality consideration shows that, with probability one, the sequence produced by our random process is not Turing-computable.

There are uncountably many infinite strings of digits. (Even more strongly, there are uncountably many infinite strings of digits with any given limiting frequency of '0's and '1's.) But there are only countably many Turing-computable infinite strings. Therefore, assuming that each infinite string (or each infinite string with a certain limiting frequency) has the same probability of occurring as a result of a random process, the probability that a random process would generate a Turing-computable string of digits is zero, whereas the probability that the string of digits is not Turing-computable is one.[20] Thus, simply by using a random process to generate a string, there is a sense in which we would have physical means that go beyond what is Turing-computable. As Alan Turing (1950, 438–9) pointed out, a machine with a "random element" can do "more" than a Turing machine.

Someone might be tempted to bite the bullet and conclude that if there are genuinely random processes, then Physical CT is false (e.g., Bowie 1973, 74; Calude 2005, 10). But there are several reasons to deny that a genuine random process P should count as a computation.

First, there is no process-independent rule (sub-constraint 2). There is no way to define the function f: $\mathbf{N} \rightarrow \{0,1\}$ whose values are being generated by P without reference to P itself. Of course, f exists in the set-theoretic sense of a set of pairs whose output values P happens to emit. But in this context, defining f means actually specifying the relationship that obtains between the arguments and values of f, as we do when we define, for instance, the halting function for Turing machines. If P is genuinely random, there is no way to specify f without generating the values of f by running P. Therefore, we should not count Bold Physical CT, which is falsified by P's existence, as a genuine version of CT.[21]

[20] Abbott et al. (2012) prove a stronger result: there are quantum random "number" generators that generate binary sequences that are not Turing-computable and, even more strongly, bi-immune, where a sequence is bi-immune just in case none of its infinite subsequences is Turing-computable.

[21] Determining which f is such that its values are generated by a random process P is not only impossible to do ahead of running P. It also contains two other elements of arbitrariness that constitute disanalogies with genuine computation. First, in the case of P, we can define indefinitely many f's depending merely on which time intervals we consider when observing P's outputs. We could count P's outputs per second, or millisecond, or year. Each temporal criterion will give rise to a different f. Second, in the case of P, we can define indefinitely many f's depending merely on which time interval we decide to count as the first one. By contrast, genuine computing systems come with a univocal criterion for what counts as the function being computed and the beginning of the computation. For instance, in the case of Turing machines, the process begins with the machine acting on a designated first square of the tape while

A second reason against counting random processes as computations is that they are not repeatable, whereas computations must be repeatable (sub-constraint 3).

A third reason is that a random process is not settable. Ordinary computing systems can be set so as to compute the value of a function for a desired argument (sub-constraint 4). But a "user" of a random process *P* cannot select the value of *f* she wishes *P* to generate. If she wishes to obtain the n^{th} value of *f*, and assuming the *P* has not generated such a value yet, all she can do is let *P* take its course and wait until the n^{th} value is generated. If this won't happen until a million years from now, that's too bad.

Someone might object that the case of a random process that takes too long to be useful is analogous to ordinary unfeasible computation. But there is an important disanalogy. The reason that ordinary computations are unfeasible is that they take too many steps for our current technology. Although some computations require so many steps that they will always be unfeasible, that is beside the point. For in general, as long as we shorten each computational step, more and more (ordinary) computations become feasible. Not so in the case of random processes. Since there is no way to repeat them, a fortiori there is no way to repeat them using faster technology. If the function *f* whose values are produced by our random process is defined to take one year per output digit, it will take one thousand years to obtain the thousandth value of *f*. We can't do anything to obtain faster results.

In fact, the objection from lack of settability does not depend on how long it takes to generate a future value. Suppose a user wishes to obtain a value that *P* produced 24 hours ago and no one recorded. There is just no way she can set *P* (or any other process, for that matter) to generate that value.

The above remarks converge on the conclusion that genuine random processes are *not* computations. Unlike computations properly so called, random processes cannot be used to generate the desired values of a function or solve the desired instances of a general problem. Random processes can be *exploited* by a computation, of course—there are important computational techniques that rely on random or pseudo-random choices at some stages of a computation.[22] But no computational technique can amount to a mere sequence of random choices. So any thesis, such as Bold Physical CT, that would be falsified by a sequence generated by a random process is too broad in scope to capture the notion of *physical* computability—the physical analogue of computability by effective procedure. Contrary to what some authors seem to assume, Bold Physical CT is not the physical analogue of Mathematical CT.

in a designated initial state; the function being computed is the one whose arguments and values are written on the tape, respectively, at the beginning and at the end of the process.

[22] If some quantum random sequences were random in the sense of algorithmic information theory, they may even raise the probability of obtaining correct solutions from computational techniques that rely on random choices (Calude 2005, 10).

4. From the Bold to the Modest Physical Church-Turing Thesis

In order to put the debate over Physical CT on more fertile ground, we need to distinguish the issue of physical computability proper—the issue that pertains to the physical analogue of Mathematical CT—from other issues that connect computability and physics. Many questions about the relationship between physical processes and computability deserve to be asked.

What can be computationally approximated to what degree under what circumstances (cf. Chapter 4)? What can be accomplished by performing certain operations over arbitrary real-valued quantities? Which systems of equations describing a physical system give rise to computable solutions? This is presumably what (A), (B), and (C), respectively, are after. These questions are interesting and deserve to be investigated. (I don't see that (D) is formulated well enough to address any interesting question.) Nevertheless, these questions do not properly belong in discussions of CT, because they are different from the question of what can be physically computed, which is the question that motivates CT in the first place.

In the end, so-called Bold Physical CT is a cluster of more or less interesting theses relating computation and physics, none of which is a version of CT properly so called. To find the physical analogue of Mathematical CT, we need to moderate our ambition. That's what the next chapter is about.

16

The Modest Physical Church-Turing Thesis

In the previous chapter, I introduced the Church-Turing thesis (CT) and distinguished between its Mathematical version, which is supported by the original arguments or CT, and its Physical version, which is about physical processes in general. I also introduced a usability constraint on physical computation: in order to count as a computation, a physical process must be usable by a finite observer to generate the values of a function defined over a denumerable domain. Finally, I argued that Bold Physical CT, which says that any physical process is Turing-computable, is inadequate as a physical analogue of Mathematical CT, precisely because it encompasses processes that fail to satisfy the usability constraint.

In this chapter, I will argue that the correct physical analogue of Mathematical CT is Modest Physical CT, which is grounded in a notion of physical computation based on the usability constraint I introduced in the previous chapter. In other words, genuine physical computation must be *usable*. I will also argue that current proposals for machines that falsify Physical CT are still far from doing so because they have not been shown to satisfy this usability constraint. I will conclude that Modest Physical CT is the correct physical formulation of CT and is plausible in light of current evidence.

Of course, many others have previously suggested or assumed that for a physical process to count as a genuine computation, it must be *usable*. But previous efforts in this direction have not clarified what 'usable' means and have failed to persuade the many authors who remain largely indifferent to considerations of usability. In light of this, here I build on the notion of usability, articulated in terms of explicit sub-constraints, that I defended in the previous chapter. These sub-constraints delimit the scope of the version of Physical CT that deserves its name, i.e., the modest version.

1. The Modest Physical Church-Turing Thesis

If Physical CT is to be grounded on the epistemological notion of computation, thereby avoiding the pitfalls of Bold Physical CT, it needs to cover less than what can be *physically done*. What Physical CT needs to cover is any process of genuine

physical computation, where physical computation is explicated in terms of our usability constraint and its sub-constraints (Chapter 15, Section 2).

This proposal turns Physical CT into the following:

> **Modest Physical CT**: Any function that is physically computable is Turing-computable.

Modest Physical CT asserts that every function defined over a denumerable domain that can be physically computed—every usable transformation of input strings into output strings in accordance with a process-independent rule defined over the strings—is Turing-computable.

Modest Physical CT has two desirable features. First, it is relevantly analogous to Mathematical CT, because it is faithful to the epistemological concerns that motivated CT in the first place. In other words, Modest Physical CT has to do with physical processes that can be used by a finite observer to generate the values of a desired function. The second desirable feature is that Modest Physical CT is open to empirical refutation.[1]

Prototypical examples of physical computation are the processes of ordinary digital computers and their components, including digital circuits. Such processes can be exhaustively described by effective procedures, which are already covered by Mathematical CT. Mathematical CT says that any function computable by an effective procedure is Turing-computable. Thus, any physical process such as an ordinary digital computation, which follows an effective procedure, is Turing-computable.

But physical computation in the present sense is a broader notion than computation by effective procedure. A process may count as a physical computation even if there is no effective procedure for describing the process, perhaps because there are no finite instantaneous descriptions of the internal states that constitute the process or no way to finitely and exactly specify the transition from one instantaneous description to the next. Many neural networks are like that (Chapter 13). Thus, Modest Physical CT, which is formulated in terms of physical computability, is

[1] By contrast, consider the alternative approach to Physical CT that originates with Robin Gandy (1980) and has been developed primarily by Wilfried Sieg (2002, 2006b, 2008). According to Gandy and Sieg, Physical CT pertains to what can be "computed" by certain "discrete dynamical systems". Gandy and Sieg define their discrete dynamical systems in terms of a set of assumptions. Specifically, they postulate finiteness and locality conditions, such as discrete states, discrete dynamics, a lower bound on the size of atomic components, and an upper bound on signal propagation. Gandy and Sieg prove that anything "computable" by any of their discrete dynamical systems is computable by Turing machines. Some of their assumptions—such as a lower bound on the size of atomic components and an upper bound on signal propagation—are empirically well-motivated. But other assumptions—such as discreteness of states and discreteness of dynamics—are not empirically justified. Thus, there remains the empirical question of whether there are physical systems that violate Gandy and Sieg's assumptions and yet are computationally more powerful than Turing machines. This is the real question of interest here, and it is not settled by Gandy and Sieg's work. This is probably why Gandy and Sieg's work has had relatively little impact on discussions of physical computability. For related discussions of Gandy and Sieg's approach, see Shagrir and Pitowsky 2003 and Copeland and Shagrir 2007.

stronger than Mathematical CT, which is formulated in terms of computability by effective procedure. In addition to physical processes that follow effective procedures, Modest Physical CT may cover processes that exploit properties of spacetime (as in relativistic computing), continuous dynamical processes (as in analog computers and certain kinds of neural networks), and quantum processes (as in quantum computing).

Since Modest Physical CT is restricted by epistemologically relevant criteria, it doesn't raise the worries associated with Bold Physical CT—namely, that it's too easy to falsify and irrelevant to the epistemological notion of computability that motivates CT. It also allows us to shed light on why most computability theorists and computer scientists believe in Physical CT. To illustrate how to assess Modest Physical CT and why it is plausible, I will briefly discuss some purported counterexamples.

2. Hypercomputation: Genuine and Spurious

The term *hypercomputer* is often used for any system that yields the values of a Turing-uncomputable function. If what counts as yielding the values of a function is left unspecified, any of the systems discussed in Section 3 of Chapter 15, such as systems with genuinely random outputs and systems that manipulate arbitrary real-valued quantities, would count as hypercomputers. But in discussing Bold Physical CT, we saw that yielding the values of a function that is Turing-uncomputable, without further constraints, is not enough for genuine physical computation.

By analogy with the distinction between Bold Physical CT and Modest Physical CT, let's distinguish between a weak and a strong notion of hypercomputation by distinguishing between genuine and spurious hypercomputers.

A *spurious* hypercomputer is a physical system that fails to satisfy at least one of the first four constraints on physical computation (Chapter 15, Section 2). Examples include processes whose inputs or outputs are arbitrary real-valued quantities (which are not readable without error) and genuine random processes (which have no rule characterizing the inputs and outputs independently of the process, and are neither repeatable nor settable). These putative hypercomputers are spurious because they cannot be used by an observer to compute arbitrary values of an independently defined function on an input chosen by the user, as ordinary computing systems can (given enough time and space). Since spurious hypercomputers are not computing systems, they are irrelevant to Modest Physical CT.

A *genuine* hypercomputer is a physical system that satisfies at least the first four constraints on physical computation. It has readable inputs and outputs, there is a rule characterizing its input-output properties that may be defined independently of the process itself, and its processes are repeatable and settable. There remains the question of whether any genuine hypercomputers are physically constructible and reliable. If they are, they refute Modest Physical CT.

Many schemes for putative hypercomputers have been proposed. In some cases, it is obvious that they are not physically constructible. For instance, infinitely accelerating Turing machines (Copeland 2002a) are Turing machines that perform each computational operation twice as quickly as the previous one. As a consequence, infinitely accelerating Turing machines complete an infinite number of operations (a *supertask*) within twice the time it takes them to perform their first operation. This allows infinitely accelerating Turing machines to compute functions, such as the halting function, that are not Turing-computable. But infinitely accelerating Turing machines are usually discussed as notional entities, without suggesting that they can be constructed. Purely notional systems, of course, do not falsify Modest Physical CT. To do that, a system must satisfy at least the fifth and sixth constraints on physical computation: it must be physically constructible and it must operate reliably.

One way to construct something like an infinitely accelerating Turing machine would be to make a computing machine that, after performing some computational operations, builds a smaller and faster copy of itself (Davies 2001). The smaller and faster copy will also perform some operations and then build a faster and smaller copy, and so on. Given appropriate assumptions, the resulting series of infinitely shrinking machines will be able to complete an infinite number of computational operations within a finite time, thereby surpassing the power of Turing machines. While infinitely shrinking machines appear to be consistent with Newtonian mechanics, Davies (2001, 672) points out that the atomic and quantum mechanical nature of matter in our universe makes infinitely shrinking machines physically impossible.

In recent years, several designs for hypercomputation have been proposed. If genuine hypercomputation turns out to be physically constructible and reliable, Modest Physical CT would be refuted.

3. Relativistic Hypercomputers

One of the best-known proposals for a hypercomputer is due to Mark Hogarth (1994, 2004), who developed an idea of Itamar Pitowsky (1990; see also Etesi and Németi 2002). I will now briefly discuss Hogarth's proposal to illustrate what is required for a genuine hypercomputer to falsify Modest Physical CT.

Hogarth's relativistic hypercomputers exploit the properties of a special kind of spacetime, called *Malament-Hogarth spacetime*, which is physically possible at least in the sense of constituting a solution to Einstein's field equations for General Relativity. Malament-Hogarth spacetimes contain what may be called spacetime *edges*—regions containing an *infinite* time-like trajectory λ that can be circumvented by a *finite* time-like trajectory g. In other words, λ and g have a common origin—here called a *bifurcation*—and there is a spacetime point p on g such that λ, even though it is infinite, lies entirely in p's chronological past.

In addition to the operations performed by Turing machines (TMs), relativistic hypercomputers exploit five further primitive operations, described by the following

instructions: (1) Position yourself at a bifurcation (where both λ and g start); (2) Launch a TM along λ while you remain on g; (3) Pass the edge (i.e., go to point p, by which time g has circumvented λ); (4) Send a signal (from λ to g); and (5) Receive a signal (coming from λ).

With the resources offered by relativistic hypercomputers, we can define a procedure for solving the halting problem for Turing machines. The halting problem asks, given a TM t and an input x, does t halt on x? Exploiting the power of relativistic hypercomputers, a procedure for solving the halting problem can be defined as follows:

1. Prepare t with input x and add to t's instructions the instruction to send a signal (from λ to g) upon halting.
2. Position yourself at a bifurcation.
3. Launch t along λ while you remain on g.
4. Pass the edge while receiving a signal coming from λ, if there is one.

In the finite time it takes an observer to travel through g, this procedure determines whether t halts on x. This is because by the time g circumvents λ, which it will do by the definition of Malament-Hogarth spacetime, the observer traveling through g will receive a signal if and only if t halts on x. Therefore, the above procedure solves the halting problem for TMs.[2]

Are relativistic hypercomputers genuine hypercomputers? They are if they satisfy Sub-constraints 1–4. Since they operate on strings of digits, they appear to satisfy Sub-constraint 1 (readable inputs and outputs). This appearance hides the difficulty in transmitting t's output from λ to g in such a way that the receiver can read it. This is a serious challenge, which I will discuss later under the rubric of reliability. Since the functions relativistic hypercomputers allegedly compute (e.g., the halting function) are defined independently of their activity, they satisfy Sub-constraint 2 (process-independent rule). As Hogarth initially defines them, however, relativistic hypercomputers fail to satisfy Sub-constraints 3 (repeatability). This is because relativistic hypercomputers, as they are usually defined, have access to at most one spacetime edge. If there is only one accessible edge, a relativistic hypercomputer may be run only once, on one input, in the history of the universe. This violates repeatability. This point is generally ignored in the literature on relativistic hypercomputers.

Hogarth also defines Malament-Hogarth spacetimes that contain an infinite number of edges. For our purposes, we don't need an actual infinity of edges; we only need a large number. If they are accessible one after the other, each of them may be exploited to run a distinct computation. That would give us enough resources to repeat computations or compute different values of the same function, thereby

[2] For a more detailed treatment of what relativistic hypercomputers can compute under various conditions, see Welch 2008.

satisfying Sub-constraint 3.[3] As we shall see, however, the hypothesis that one spacetime edge can be exploited successfully by a relativistic hypercomputer is fantastic enough.

As to settability, some components of a relativistic hypercomputer get lost in the course of the computation—they are never recovered after they are launched along λ. So relativistic hypercomputers are not resettable. But at least they are settable, because each TM can be set to compute the value of a desired function for a desired input. Thus, relativistic hypercomputers satisfy Sub-onstraint 4 (Settability).

Given the above discussion, we may tentatively conclude that relativistic hypercomputers satisfy Sub-onstraints 1–4, which makes them genuine hypercomputers. It remains to be seen whether they falsify Modest Physical CT. For that, they must be physically constructible and reliable.

Constructing a relativistic hypercomputer is highly nontrivial. The first question is whether our spacetime is Malament-Hogarth; the answer is currently unknown. An example of a region possessing the Malament-Hogarth property is the region surrounding a huge, slowly rotating black hole; there is evidence that our universe contains such regions (Etesi and Németi 2002). If the universe contains no Malament-Hogarth regions, Hogarth's relativistic computers are not physically constructible. But even if there are Malament-Hogarth regions in our universe, there remain formidable obstacles.

In order to exploit the special properties of Malament-Hogarth spacetimes, relativistic hypercomputers need to be started at a bifurcation. So, for a user to do anything with a relativistic hypercomputer, there must be bifurcations within regions of spacetime that she can access. Otherwise, if all bifurcations are out of reach, she cannot exploit them to run relativistic hypercomputers. But notice that the huge, rotating black hole that is closest to us, which is the basis for Etesi and Németi's proposed implementation of a relativistic hypercomputer, is presumably out of our reach as well as the reach of our descendants.[4]

Additionally, in order to work with a relativistic hypercomputer, a user needs to *know* that she is at a bifurcation. She must also know how to launch a TM along the infinite trajectory λ that starts at the bifurcation while proceeding along the finite trajectory g. Finally, she must know when she has circumvented λ. Therefore, for relativistic hypercomputers to be constructible, it must be possible to discover when one is in the presence of a bifurcation, how to launch machines along λ, how to

[3] Repeatability is not the purpose for which Hogarth introduces his spacetimes with infinitely many edges; I'm putting them to a different use. Cf. Shagrir and Pitowsky's discussion of their Objection #3, which they eventually handle in the same way (2003, 91–3).

[4] Someone may object that the interest in Modest Physical CT is simply the question of what computations the physical laws allow; whether the needed resources are too far away is irrelevant. But this is precisely what I've been arguing against. If needed resources are too far away, a finite observer (of finiteness comparable to ours) cannot use them. Any scheme requiring resources that are too far away violates the usability constraint and thus fails to refute Modest Physical CT.

proceed along g, and when one has circumvented λ. I've never seen any mention of these issues in the current literature on relativistic hypercomputation. Yet an observer who happens to travel on g while a Turing machine happens to travel on λ performs a computation only if the observer can deliberately use this set up to learn the desired values of a function.

Another difficulty is that in order to run their TM for an infinite amount of time, which they may need to do, relativistic hypercomputers need an unbounded memory store. The reason is that in some cases, the hypercomputer gives a correct output just in case its TM does not halt; to perform its infinitely many steps correctly, the TM needs an unbounded memory in which to store an unbounded number of digits (cf. Shagrir and Pitowsky 2003, 88–90). Because of this, a relativistic hypercomputer demands more than a Turing machine, which requires only a finite amount of memory to produce any given correct output. (Plus, recall from Chapter 15, Section 2 that according to Modest Physical CT, Turing machines act as the upper bound to what is physically computable, which does not require that they be physically constructible.)

An unbounded memory may require an unbounded amount of matter-energy. Recent evidence suggests that the universe is infinite not only in space and time, but also in matter-energy. Thus, letting our relativistic hypercomputer spread over the universe, and assuming that the universe's expansion does not keep accelerating forever, there may be enough matter-energy for a relativistic hypercomputer (Németi and Dávid 2006). But it's not clear how spreading a relativistic hypercomputer over the entire universe can be reconciled with its need to travel through a specific time-like trajectory λ. If we are unlucky and there is no way to harness an unbounded amount of matter-energy, there might be other ways to keep a relativistic hypercomputer running; for one thing, recent developments in quantum computing suggest that storing one digit does not require a fixed amount of matter-energy (Németi and Dávid 2006). In any case, building an unbounded memory is a daunting challenge.

Yet more exotic obstacles to the physical constructibility of relativistic hypercomputers involve the evaporation of black holes and the eventual decay of matter, both of which are possible within a finite amount of time according to some current physical theories. Either of these possibilities might prevent the completion of a putative relativistic hypercomputation. I cannot do justice to these issues here. Suffice it to say that the solutions proposed, such as sending matter into a black hole to prevent its evaporation (Németi and Dávid 2006), have not been worked out in great detail. If all of these practical obstacles could be overcome, and if our other needs were fulfilled—two very big ifs—then relativistic hypercomputers would be physically constructible.

The final constraint is reliability. One serious problem is the successful decoding of the signal, coming from λ, that carries the result of the computation. It is difficult to ensure that the signal is distinguishable from other possible signals coming from

outer space. In addition, the signal is subject to amplification. If the signal has infinitely long duration, upon amplification it will destroy the receiving agent (Earman and Norton 1993). In such a case, the completion of the computation leads to the user's destruction. If the signal is finite, under Etesi and Németi's proposal, gravitational forces would shorten the signal: as the receiver approaches the edge, the signal tends to zero length. Therefore, decoding the signal will require time measurements of unbounded precision (Etesi and Németi 2002). Since such measurements are unlikely to be available and appear to violate quantum mechanical constraints, Németi and Dávid (2006) and Andréka, Németi, and Németi (2009, 508–9) propose alternative solutions to the reception problem. One solution involves sending a messenger from λ towards g; roughly speaking, although the messenger cannot reach g, it can come close enough to transmit the signal in a decodable form without damaging the receiver. None of these solutions appear especially easy to implement.

Even if the problem of decoding the signal can somehow be solved, two more problems remain. First, it is unlikely that a machine can operate for an infinite amount of time without breaking down and eventually being unable to repair itself.[5] Second, as Pitowsky (2007) argues, the singularities involved in relativistic hypercomputation (such as electromagnetic waves whose energies are too high to be described by known laws of physics) may indicate a limitation of General Relativity rather than the physical possibility of hypercomputation.

Relativistic hypercomputers are fascinating. They are a fruitful contribution to the debate on the foundations of physics. But at the moment, they are not even close to being technologically practical. It's not clear that relativistic hypercomputers are physically possible in any robust sense, and in any case it is extremely unlikely that relativistic hypercomputers will ever be built and used successfully. So for now and the foreseeable future, relativistic hypercomputers do not falsify Modest Physical CT.

4. Other Challenges to Modest Physical CT

I will not discuss all hypercomputer designs proposed in the literature in detail, but I will briefly mention two other widely discussed examples.

Neural networks (Chapter 13) have sometimes been proposed as computing systems that may go beyond Turing-computability.[6] This opinion is unwarranted. During the last couple of decades, there has been considerable progress in the study of the computational and complexity properties of large classes of neural networks.

[5] Button has independently argued that a relativistic hypercomputer will malfunction with probability 1 and concludes that this is enough to make it useless (2009, 779).

[6] Cf.: 'connectionist models . . . may possibly even challenge the strong construal of Church's Thesis as the claim that the class of well-defined computations is exhausted by those of Turing machines' (Smolensky 1988, 3). See also Horgan 1997, 25.

The relevant systems have digital inputs and outputs (so as to satisfy Constraint 1) but may have, and typically do have, non-digital internal processes. If we restrict our attention to classes of connectionist systems that contain all systems with current or foreseeable practical applications, the main results are the following. Feedforward networks with finitely many processing units are computationally equivalent to Boolean circuits with finitely many gates. Recurrent networks with finitely many units are equivalent to finite state automata. Networks with unbounded tapes or with an unbounded number of units are equivalent to Turing machines.[7]

Neural networks more powerful than Turing machines may be defined, however, by exploiting the expressive power of real numbers. The best-known networks of this kind are Analog Recurrent Neural Networks (ARNNs) (Siegelmann 1999). ARNNs should not be confused with analog computers in the traditional sense (Chapter 12) or with neural networks that manipulate continuous variables similarly to analog computers (Chen and Chen 1993). Whereas analog computers and neural networks that are similar to analog computers manipulate real (i.e., continuous) variables without relying on the exact value of arbitrary real-valued quantities, ARNNs manipulate strings of digits by (possibly) relying on the exact value of arbitrary real-valued quantities. Specifically, the weights connecting individual processing units within ARNNs can take exact values of arbitrary real numbers, including values that are Turing-uncomputable. When their weights are Turing-uncomputable, ARNNs can go beyond the power of Turing-machines: they can compute any function over binary strings. The very features that make some ARNNs more powerful than Turing machines, however, also prevent them from being built and operated reliably. The reasons are roughly the same that militate against Blum et al.'s "computations" (Chapter 15, Section 3): first, hypercomputational ARNNs require unboundedly precise weights, and second, such weights are not Turing computable (Davis 2004a, Schonbein 2005, Siegelmann 1999, 148).

Quantum computing has also been invoked as a possible source of hypercomputation. Quantum computing is the manipulation of qubits (or more generally, qudits) in accordance with the laws of quantum mechanics. Qubits are variables that, like bits, can be prepared or measured in one or two states, '0' and '1'. Unlike bits, qubits can (i) take states that are a superposition of '0' and '1' and (ii) become entangled with each other during a computation. A surprising feature of quantum computing is that it allows computing certain functions much faster than any known classical computation (Shor 1994). But while mainstream quantum computing may be more efficient than classical computing, it does not allow computing any functions beyond those computable by Turing machines (Nielsen and Chuang 2000).

[7] For some classical results and discussions, see McCulloch and Pitts 1943; Kleene 1956; Minsky 1967; Minsky and Papert 1988. For more recent results, reviews, and discussions, see Hartley and Szu 1987; Hong 1988; Franklin and Garzon 1990; van der Velde 1993; Siu et al. 1995; Sima and Orponen 2003.

Some authors have questioned whether the mainstream quantum computing paradigm is general enough and, if not, whether some aspects of quantum mechanics may be exploited to design a quantum hypercomputer (Nielsen 1997; Calude and Pavlov 2002). The most prominent proposal for a quantum hypercomputer is by Tien Kieu (2002, 2003, 2004, 2005). He argues that an appropriately constructed quantum system can decide whether an arbitrary Diophantine equation has an integral solution—a problem which is known to be unsolvable by Turing machines. Kieu's method involves encoding a specific instance of the problem in an appropriate Hamiltonian, which represents the total energy of a quantum system. Kieu shows that such a system can dynamically evolve over time into an energy ground state that encodes the desired solution.

Unfortunately, Kieu's scheme does not appear to be workable. For one thing, it requires infinite precision in setting up and maintaining the system (Hodges 2005). For another thing, Kieu does not provide a successful criterion for knowing when the system has evolved into the solution state, and the problem of determining when the solution state is reached is unsolvable by Turing machines (Smith 2006b; Hagar and Korolev 2007; Pitowsky 2007). Thus, operating Kieu's proposed hypercomputer would require already possessing hypercomputational powers.

In conclusion, the most prominent candidate hypercomputers proposed so far have not been shown to be physically constructible and reliable.[8] For the time being, Modest Physical CT remains plausible. It may well be that, for all practical purposes, any function that is physically computable is Turing-computable.

5. Conclusion

In the literature on computation in physical systems, there is growing concern with whether various proposals for physical computation lead to physically usable processes (e.g., Fitz 2006; Németi and Dávid 2006; Ord 2006; Smith 2006a; Beggs and Tucker 2007; Button 2009). In this chapter and the previous one, I have urged a more explicit, careful, and systematic treatment of this problem and the related problem of formulating an adequate version of Physical CT.

In formulating Physical CT, we should mind the reason computability is interesting in the first place—it's the epistemological notion of which antecedently defined problems, defined over denumerable domains, can be solved, either by effective procedures (Mathematical CT) or by physical means (Physical CT).

Mathematical CT does not rule out the possibility that we construct a genuine hypercomputer. It merely rules out the possibility that a hypercomputer computes Turing-uncomputable functions by following an effective procedure.

[8] For some related skepticism about these and other hypercomputation proposals, see Cotogno 2003; Davis 2006; Galton 2006; Potgieter 2006. For a mistake in Cotogno's paper (which is irrelevant here), see Welch 2004 and Ord and Kieu 2005.

If we formulate physical versions of Physical CT too strongly—what I called Bold Physical CT—we face two related problems. First, it becomes relatively easy to falsify CT, but the putative counterexamples have no apparent practical utility. Second, and more importantly, these versions of CT are irrelevant to the epistemological notion of computability that motivated CT in the first place. They change the subject.

There is nothing wrong with changing the subject if you are interested in something else. There are legitimate and interesting questions pertaining to which aspects of which physical systems can be computationally approximated to what degree, which systems of equations give rise to computable solutions, and more. These questions are not the same as whether Physical CT (properly so called) is true, but they do deserve to be investigated in their own right.

Physical CT should be formulated modestly, using a notion of physical computation that is broader than that of computation by effective procedure but considerably narrower than the general notion of a physical process. Such a notion should satisfy an appropriate usability constraint: For a process to count as a genuine physical computing system, it need not follow an effective procedure but it must still be usable by a finite observer to solve problems of a certain kind. That is, it must generate readable outputs from readable inputs according to a fixed rule that links the inputs to the outputs without making reference to the physical process itself. It must also be repeatable, settable, constructible, and reliable.

It is important to understand the exact scope of Modest Physical CT. Modest Physical CT does not entail that everything physical is a computing system (cf. Chapter 4). It only says that *if* something physical is a computing system, *then* the functions it computes are Turing-computable.

Modest Physical CT is true if and only if genuine hypercomputers are impossible in the sense that they do not satisfy our usability constraints. Whether any hypercomputer does satisfy our usability constraints remains an open empirical question, as does Modest Physical CT. As yet, however, there is no hard evidence against Modest Physical CT. Instead, there are good reasons to believe Modest Physical CT. All computing systems that have been physically built, are in the process of being built, or are likely to ever be built, only compute functions that are Turing-computable.

Epilogue: The Nature of Computation

In this book, I defended a mechanistic account of concrete, or physical, computation. A physical system is a computing system just in case it has the following characteristics:

- It is a functional mechanism.
- One of its functions is to manipulate vehicles based solely on differences between different portions of the vehicles according to a rule defined over the vehicles.

This is not a set of sharp necessary and sufficient conditions but a set of features that any paradigmatic computing system has, and any paradigmatic case of non-computing system lacks.

Any physical process that is performed by a functional mechanism and satisfies the second clause above is a concrete computation. For instance, the characteristic processes performed by Turing machines, finite state automata, neural networks (including standard digital computers, which are a kind of neural network—cf. Chapter 13), analog computers, and genuine hypercomputers qualify as computations.

Any physical process that lacks at least one of these characteristics may be subject to computational modeling but is not a concrete computation (cf. Chapter 4). For instance, hurricanes, spin-glasses, and galaxies do not perform computations because they are not even *functional* mechanisms—they do not have any teleological functions—although they are mechanisms in a broader, non-teleological sense. Stomachs and washing machines are functional mechanisms but their functions are defined in terms of specific physical effects, such as specific chemical transformations of edible substances or removing dirt from clothes; therefore, their function is not computational. Finally, any device that produces random outputs fails to follow an appropriate rule; therefore, it does not perform computations.

As with most interesting notions, there may well be cases at the boundaries between clear cases of computations and clear cases of non-computational processes. An example may be a device, not hooked up to any larger system, which could be described as a lonely logic gate. Is one of its *functions* to manipulating vehicles based

solely on differences between different portions of the vehicles according to a rule defined over the vehicles? In principle, the answer to this question depends on its contributions to the goals of organisms (Chapter 6). In practice, there may be devices for which a definitive answer may be difficult to come by. Maybe there are devices whose computational contributions to the goals of organisms are so rare and unstable that they fall in the grey area between systems that compute and systems that do not compute. If so, that's ok.

In addition to being extensionally adequate, the mechanistic account has a number of desirable consequences.

Computation is objective. Whether a system performs a computation is one of its objective properties. Artifacts may be engineered to perform computations in an objective sense. This provides an objective basis for the practices of computer science and computer engineering. Natural systems, such as brains, may be empirically investigated to find out whether they perform computations and which computations they perform. This provides an objective basis on which to build a computational science of natural computation.

Computation is explanatory. When a system performs a computation, that computation explains the behavior of the system. Computational explanation is a special kind of mechanistic explanation that applies to systems that manipulate appropriate vehicles by following appropriate rules. This distinguishes computing systems from other systems, which may be modeled computationally but do not perform computations.

Computation can go wrong. When a computing system is poorly designed, poorly built, poorly used, or malfunctions, the result is a miscomputation. While it is hard or impossible to make sense of miscomputation within traditional accounts of computation, miscomputation finds an adequate explication within the mechanistic account.

There are many kinds of computation and computing system (Chapters 8–13). The mechanistic account has the resources to distinguish between and explain different kinds of computing system with many different properties, including computations that use different vehicles (digital, analog, neural), systems that are computationally more or less powerful (computing functions of single digits, Boolean functions, functions of finite domains, functions of infinite domains, executing programs), systems that are hardwired vs. systems that are programmable, or systems that are more or less flexible (special-purpose, general-purpose, universal).

Computation does not presuppose representation but is compatible with it (Chapters 3, 14). Computer scientists and cognitive scientists are free to investigate the representational properties of the systems they study and use representational content for legitimate purposes, but they need not assume that computational vehicles are representations, and even computations that manipulate representations and may be individuated semantically can also be individuated non-semantically.

Concrete computation is, most likely, Turing-computable. Any computing system that has ever been built and is likely to ever be built and used performs computations that are Turing-computable. Establishing this fact (Chapters 15, 16) requires examining empirical evidence about what is and isn't physically possible. How could this be?

Computation is both a mathematical notion, which may be formalized and studied mathematically, and a physical one, which must be studied empirically. Therefore, establishing what is and isn't computable by various physical means (effective procedures vs. physical computation more generally) requires considering both mathematical evidence about what can be computed by effective procedures (a mathematical notion with direct empirical consequences) and empirical evidence about what is physically possible. In Chapter 15, we briefly reviewed mathematical arguments for the Mathematical Church-Turing thesis, according to which every function that is computable by an effective procedure is computable by a Turing machine. In Chapter 16, we saw empirical arguments for the Physical Church-Turing thesis, according to which every function that is computable by a physical system is computable by a Turing machine.

I have given a mechanistic account of concrete computation. I have explored some fascinating questions about concrete computation. I hope some of what I said was interesting and true, or at least valuable. I hope some of you, kind readers, will continue this exploration.

Appendix: Computability

1. Effective Calculability

This appendix introduces some basic mathematical notions related to computation and computability. This first section is devoted to the pre-theoretical notion of effective calculability, or computability by an effective procedure (computability for short). This informal notion motivates the formally defined notion of Turing-computability, which I introduce in the following section. In the last section, I briefly introduce the Church-Turing thesis, which says that the formal notion is an adequate formalization of the informal one.

During the first decades of the 20th century, mathematicians' interest in computable functions lay in the *foundations* of mathematics. Different philosophical approaches were proposed. L. E. J. Brouwer was the main supporter of *intuitionism*, according to which an existence proof for a mathematical object was admissible only if constructive (Brouwer 1975). David Hilbert proposed his *proof theory* to formalize in axiomatic fashion mathematical reasoning in an attempt to establish the foundations of mathematics without endorsing Brouwer's restrictions (van Heijenoort 1967; Hilbert and Ackermann 1928). This formalization allowed Hilbert to formulate rigorously the decision problem for first-order logic. A *decision problem* requests a method for answering a yes-or-no question concerning a domain of objects: "Given any sequence of symbols, is it a formula?" or "Given any formula, is it provable?" A solution to a decision problem is an effective procedure: a uniform method or algorithm specified by a finite set of instructions, by which any instance of the question can be answered in a finite number of steps. "Effective procedure" is a term used by some mathematicians in place of "algorithm"; an effective procedure cannot appeal to non-extensionally definable capacities like intuition, creativity, or guesswork, and it always generates the correct answer.[1]

Lacking a rigorous definition of "effective procedure," mathematicians called it an "intuitive concept" to distinguish it from formally defined mathematical concepts.[2] Kurt Gödel proposed replacing "effective procedures" with a rigorously defined concept, that of "recursive functions," but he didn't rule out that some effective procedures might not be included within recursive functions (1931, 1934). Alonzo Church (1936) and Alan Turing (1936–7) strengthened Gödel's tentative identification of effective procedures and recursive functions to a general thesis, now called the Church-Turing thesis. Based on the Church-Turing thesis, Church and Turing proved that some functions are not computable. For example, Turing pointed out that he and Church used different definitions but reached "similar conclusions,"

[1] After the work of Turing and others, "effective procedure" was also used for procedures *not* guaranteed to generate all values of a total function, that is, they calculated only the values of a partial function (cf. Wang 1974, 84).

[2] To refer to the intuitive notion of effective procedure, different authors used different terms. Instead of "procedure," some used "process," "method," or "rule." Instead of "effective," some used "finite," "finite combinatorial," "mechanical," "definite," "constructively defined," or "algorithmic." Some of the terms used as synonyms of "effectively calculable" are listed by Gödel 1965, 72; Kleene 1987a, 55–6.

i.e., that "the Hilbertian Entscheidungsproblem [i.e., the decision problem for first-order logic] can have no solution" (Turing 1936–7, 116, 117, 145).

The notion of *effective procedure* can be informally defined as follows:

- *Executability*: The procedure consists of a *finite* number of *deterministic instructions* (i.e., instructions determining a unique next step in the procedure), which have *finite* and *unambiguous* specifications commanding the execution of a *finite* number of *primitive operations*.
- *Automaticity*: The procedure requires *no intuitions* about the domain (e.g., intuitions about numbers), no ingenuity, no invention, and no guesses.
- *Uniformity*: The procedure is the same for each possible argument of the function.
- *Reliability*: If the procedure terminates, the procedure generates the *correct* value of the function for each argument after a *finite* number of primitive operations are performed.

When we have a formalized language in which both a domain and operations over objects in that domain are formally defined, we can talk about lists of formalized instructions and call them *programs*. Programs are the formal replacement of algorithms or effective procedures. Because of this, programs are said to *implement* algorithms (or procedures).

Not all mathematical procedures are effective procedures or algorithms. There are also *heuristics*. Heuristics are sequences of operations that search for the value of a function but are *not* guaranteed to find all the values of the function for every argument. A heuristic search *may* find the value of the function being computed but it may also find an output that only approximates that value to some degree. A heuristic for a given function may be preferable to an algorithm for the same function when the heuristic produces good enough results within a reasonable amount of time and using a reasonable amount of memory space, while the algorithm takes too long or requires too much memory space to produce its results.

A caveat needs to be added about the relationship between programs (and procedures) and the functions they compute. A program is a definition of a function, from its inputs to its outputs; when the program doesn't halt, the function isn't defined. So, by definition, any program computes the values of the function that *it* defines: it implements an algorithm or effective procedure relative to that function. But typically a program is written to find values for a function that is defined independently of the program's existence. When that happens, the program may or may not be implementing an algorithm that finds the values of that function. Many programs do not always find the values of the independently defined function they are designed to compute, but rather they find approximations to those values. In such cases, relative to the independently defined functions, those programs are said to implement not algorithms but *heuristics*.[3]

[3] The philosophical literature is not always clear on this point. For example:

> The possibility of heuristic procedures on computers is sometimes confusing. In one sense, every digital computation (that does not consult a randomizer) is algorithmic; so how can any of them be heuristic? The answer is again a matter of perspective. Whether any given procedure is algorithmic or heuristic depends on how you describe the task (Haugeland 1997, 14).

This is unclear at best. Relative to its task (e.g., multiplying numbers, solving the traveling salesman problem, or winning chess games), a procedure (or a program) is algorithmic or heuristic depending on whether it is guaranteed to solve each instance of the task. Relative to generating its outputs from its inputs

2. Computability Theory

This section reviews some notions and results of classical computability theory. Computability theory studies what functions are computable, what mathematical properties they have, and the mathematical properties of computing mechanisms. Computable functions are identified with the class of recursive functions, inductively defined. As a specific example of computing mechanism, I will give Turing machines. Using Turing machines and recursive functions, I will introduce the notion of universal computing mechanism and the unsolvability of the halting problem.

2.1. Notation

$\{a_1, a_2, \ldots a_n\}$	Set of n objects $a_1, \ldots a_n$
$(a_1, a_2, \ldots a_n)$	List (or n-tuple) of n objects $a_1, \ldots a_n$
$a \in A$	a is an element of set A
N	Set of natural numbers 0, 1, 2, …
$f: A_1 \rightarrow A_2$	f is a function from A_1 to A_2
Domain of f	Set of all a such that $(a, b) \in f$ for some b
Range of f	Set of all of f(a) for a in the domain of f
Partial function on A	Function whose domain is a subset of A
Total function on A	Function whose domain is A
Alphabet	Nonempty set Σ of objects called symbols
Word or string on Σ	List of symbols on Σ, (instead of $(a_1, a_2, \ldots a_n)$, we write $a_1a_2 \ldots a_n$)
$\lvert u \rvert = n$, where $u = a_1a_2 \ldots a_n$	n is the length of u
a_1^n	concatenation of n symbols a_1
Σ^*	Set of all words on alphabet Σ
Language on Σ	Any subset of Σ^*
uv, where $u, v \in \Sigma^*$	Concatenation of u and v
Predicate on a set A	A total function $P: A \rightarrow \mathbf{N}$ such that for each $a \in A$, either $P(a) = 1$ or $P(a) = 0$, where 1 and 0 represent truth values
$R = \{a \in A \lvert P(a)\}$, P a predicate on A	R is the set of all $a \in A$ such that $P(a) = 1$; P is called the characteristic function of R
Pr (k)	k^{th} prime in order of magnitude
$\Psi_M^n(x_1, \ldots x_n)$	n-ary function computed by TM program M; when n = 1 we omit n

(regardless of whether the outputs are a solution to that instance of the task), a procedure (or a program) is always algorithmic.

Another example of confusion about this point is manifested by Dennett's statement (1975, 83) that human beings may not be Turing machines (TMs), because humans may be implementing heuristics rather than algorithms. This presupposes that TMs implement only algorithms and not heuristics. Now, it is true that every TM implements an algorithm that generates its outputs given its inputs. But relative to the problem TMs are designed to solve, TMs—like any other computing mechanisms—may well implement heuristics.

Computability theory applies to general word functions f: $\Sigma^* \rightarrow \Sigma'^*$, where Σ^* is the set of all words on alphabet Σ. Since words can be effectively encoded as natural numbers and vice versa (see Section 2.4 below for an example of such an encoding), in this section we follow the standard convention of developing the theory with respect to number-theoretic functions f: $\mathbf{N} \rightarrow \mathbf{N}$, without loss of generality.[4] Therefore in this section, unless otherwise specified, 'number' means natural number, and 'function' means function on natural numbers. For the exposition of the material in this section, I drew mostly from Davis 1958 and Davis et al. 1994.

2.2. Recursive functions

This section introduces the definition of the primitive recursive functions on the basis of three primitive base functions and two primitive operations. Then, by means of one further primitive operation, the class of partial recursive functions is defined.

The class of primitive recursive functions is defined inductively as follows.

Base functions:

Null function. $n(x) = 0$.
Successor function. $s(x) = x + 1$.
Projection functions. $u_{i^n}(x_1, \ldots x_n) = x_i$.

Operations:

Composition. Let f be a function of k variables and let $g_1, \ldots g_k$ be functions of n variables. Let:

$$h(x_1, \ldots x_n) = f(g_1(x_1, \ldots x_n), \ldots g_k(x_1, \ldots x_n)).$$

Then h is obtained from f and $g_1, \ldots g_k$ by composition.

Primitive recursion. Let f be a function of n variables and let g be a function of n+2 variables. Let:

$$h(x_1, \ldots x_n, 0) = f(x_1, \ldots x_n)$$
$$h(x_1, \ldots x_n, t + 1) = g(t, h(x_1, \ldots x_n, t), x_1, \ldots x_n)$$

Then h is obtained from f and g by primitive recursion.

Definition 1. A function f of n variables is *primitive recursive* if and only if it can be obtained from the base functions by finitely many operations of composition and primitive recursion.

Examples of primitive recursive functions include *addition, multiplication, exponentiation, predecessor*, and many other useful functions (see Davis et al. 1994, section 3.4).

In the present context, predicates are total functions whose values are 0 or 1 (representing true and false). Any primitive recursive function whose values are 0 and 1 is called a primitive recursive predicate. An example of a primitive recursive predicate is *equality*.

It can be easily shown, by induction on the definition of primitive recursive function, that every primitive recursive function is total.

[4] Computability theory can also be developed directly in terms of string functions (Machtey and Young 1978).

Next, we introduce a further operation:

Minimalization (unbounded). Let P be a predicate of n+1 variables. We write $\min_y P(x_1, \ldots x_n, y)$ for the least value of y for which the predicate P is true if there is one. If there is no such value of y, then $\min_y P(x_1, \ldots x_n, y)$ is undefined.

Unbounded minimalization of a predicate can easily produce a function that is not total. An example is provided by *subtraction*:

$$x - y = \min_z(y + z = x),$$

which is undefined for $x < y$.

Definition 2. A function f is *partial recursive* if and only if it can be obtained from the base functions by finitely many operations of composition, primitive recursion, and minimalization.

A partial recursive function that is total is called total recursive.

2.3. Turing machines

Turing machines (TMs) are the best-known computing machines. They have two main components. First, there is a two-way potentially infinite tape divided into squares; each square contains one symbol (which may be an empty square). Second, there is an active device that can be in one of a finite number of states. The active device acts on the tape in one of four ways: it reads the symbol on a square, writes a symbol on a square, moves one square to the left, or moves one square to the right. TMs' active devices operate in discrete time. At any instant, the active device reads the symbol on one of the tape's squares. Then, the symbol on that square and the device's current state determine what the active device does: what state it goes into and whether it moves left, moves right, or writes a symbol on the current square (and which symbol it writes). When this happens, we say that an active device *responds* to its internal state and symbol on the tape. All TMs have this structure in common.

Although strictly speaking it is the active devices of TMs that perform operations (on the tape, which is passive), for simplicity I follow the standard convention of ascribing activities to TMs *simpliciter*. TMs are distinguished from one another by the alphabet they operate on, by the number of their internal states, and more importantly by the particular actions they perform in response to their internal states and the symbols on the tape. A description of the way a particular TM responds to a particular state and symbol is here called an *instruction*. A set of instructions, which uniquely identifies a TM, is called a *TM program*.

To avoid confusion, TMs should be kept distinct from the TM programs that describe their behavior. Unlike digital computers, which compute by executing programs, ordinary TMs do not operate by responding to the TM programs that describe their behavior. Ordinary TMs simply behave in the way described by their TM programs; in other words, their behavior satisfies the instructions contained in their TM program. A TM program identifies a computational process uniquely, and a TM that satisfies the instructions listed in the program is its canonical implementation (i.e., the implementation given by Turing). But the computations defined by TM programs can also be carried out by humans or machines other than TMs.

Moreover, in Section 2.4 we shall see that TM programs can be encoded using the alphabet that TMs operate on, and then written on TM tapes. There are special TMs, called universal TMs, which can respond to any TM program written on their tape so as to mimic the behavior of the TMs described by the program. Since universal TMs do compute by responding to TM programs written on their tape, I say that they *execute* TM programs. Needless to say, the behavior of universal TMs is also described by their own TM programs, called universal TM programs. Universal TMs execute the programs written on their tape, but not the universal TM programs that describe their behavior.

In formally defining TM tables, I will use the following ingredients:

Symbols denoting *internal states* of TMs' active devices: q_1, q_2, q_3, ...

Symbols denoting *symbols* that TMs can print on the tape: S_0, S_1, S_2, ... The set of S_i's is our *alphabet*.

Symbols denoting *primitive operations*: R (move to right), L (move to left).

Expressions: finite sequences of symbols.

Instructions: expressions having one of the following forms:

(1) $q_i S_j S_k q_l$,
(2) $q_i S_j R q_l$,
(3) $q_i S_j L q_l$,
(4) $q_i S_j q_k q_l$,

Quadruples of the first type mean that in state q_i reading symbol S_j, the active device will print S_k and go into state q_l. Quadruples of the second type mean that in state q_i reading symbol S_j, the active device will move one square to the right and go into state q_l. Finally, quadruples of the third type mean that in state q_i reading symbol S_j, the active device will move one square to the left and go into state q_l.[5]

We are now ready to define (deterministic) TM programs, their alphabets, and their instantaneous descriptions or snapshots:

(Deterministic) TM program: set of instructions that contains no two instructions whose first two symbols are the same.

Alphabet of a TM program: all symbols S_i in the instructions except S_0. For convenience, sometimes I will write S_0 as B (blank), and S_1 as 1.

Snapshot: expression that contains exactly one q_i, no symbols for primitive operations, and is such that q_i is not the right-most symbol.

A snapshot describes the symbols on a TM tape, the position of the active device along the tape, and the state of the active device. In any snapshot, the S_i's represent the symbols on the tape, q_i represents the state of the active device, and the position of q_i among the S_i's represents the position of the device on the tape. For any tape and any TM program at any computation step, there is a snapshot representing the symbols written on the tape, the state of the device, and its position on the tape. At the next computation step, we can replace the old snapshot by its *successor snapshot*, whose difference from its predecessor indicates all the changes (of the tape,

[5] Instructions of the fourth type serve to define special TMs called *oracle* TMs and will not be used here.

position, and state of the device) that occurred at that step. A snapshot without successors with respect to a TM program M is called a *terminal snapshot* with respect to that program.

Using the notion of snapshot, we can rigorously define computations by TM programs:

Computation by a TM program M: finite sequence of snapshots $a_1 \dots a_n$ such that $1 \leq i < p$, a_i+1 is the successor of a_i, and a_n is terminal with respect to M. I call a_n the *resultant* of a_1 with respect to M.[6]

For example, let M consist of the following instructions:

$$q_1 S_0 R\ q_1,$$
$$q_1 S_1 R\ q_1.$$

The following are computations of M, whose last line is the resultant of the first line with respect to M:

$$\begin{array}{ll} (1) & q_1 S_0 S_0 S_0 \\ & S_0 q_1 S_0 S_0 \\ & S_0 S_0 q_1 S_0 \\ & S_0 S_0 S_0 q_1 \\ (2) & q_1 S_1 S_1 S_1 \\ & S_1 q_1 S_1 S_1 \\ & S_1 S_1 q_1 S_1 \\ & S_1 S_1 S_1 q_1 \\ (3) & q_1 S_1 S_0 \\ & S_1 q_1 S_0 \\ & S_1 S_0 q_1. \end{array}$$

With each number n we associate the string $\mathbf{n} = 1^{n+1}$. Thus, for example, 4 = 11111. With each k-tuple $(n_1, n_2, \dots n_k)$ of integers we associate the tape expression $(\mathbf{n_1}, \mathbf{n_2}, \dots \mathbf{n_k})$, where:

$$(\mathbf{n}_1, \mathbf{n}_2, \dots \mathbf{n}_k) = \mathbf{n}_1 B \mathbf{n}_2 B \dots B \mathbf{n}_k.$$

Thus, for example, (1, 3, 2) = (**1**, **3**, **2**) = 11B1111B111.

Given an initial snapshot and a program, either there is a computation or there isn't (if there isn't, it's because the list of snapshots is infinite).

Definition 3. An n-ary function $f(x_1, \dots x_n)$ is *Turing-computable* if and only if there is a Turing Machine M such that: $f(x_1, \dots x_n)$ is defined if and only if there is a computation of M whose first snapshot is $q_1(\mathbf{x_1}, \dots \mathbf{x_n})$ and whose resultant contains n_j+1 occurrences of the symbol 1, where $f(x_1, \dots x_n) = n_j$. I write:

$$f(x_1, \dots x_n) = \psi^n_M(x_1, \dots x_n).$$

Turing-computability and partial recursiveness are equivalent notions in the following sense. A function is partial recursive if and only if it is Turing-computable, and it is total recursive if and only if it is a total Turing-computable. In one direction, this is shown by constructing TM programs computing each of the base functions and by showing that TM programs can be

[6] In the rest of the book, I use 'computation' more broadly so as to include infinite sequences of snapshots as computations.

manipulated in ways corresponding to the three operations (for the details of the construction, see Davis 1958). The other direction is addressed in the following section.

2.4. Gödel numbers of TM programs

One way to develop the theory of TM programs is by using recursive functions. I use a method, developed by Gödel (1931), that allows us to use natural numbers as a code for TM instructions, and therefore for TM programs. By studying the properties of TM programs in this way, I will demonstrate the results that we are interested in, namely the existence of universal TMs and the unsolvability of the halting problem. The method followed here has the great advantage of avoiding long and laborious mathematical constructions.

The basic symbols used in formulating TM programs are the following:

R, L
$S_0, S_1, S_2, \ldots$
$q_1, q_2, q_3, \ldots$

We associate each of these symbols to an odd number ≥ 3, as follows:

3 R
5 L
7 S_0
9 q_1
11 S_1
13 q_2
etc.

Therefore, for any expression M there is now a finite sequence of odd integers $a_1, a_2, \ldots a_n$ associated to M. Now we'll associate a single number with each such sequence and hence with each expression.

Definition 4. Let M be an expression consisting of the symbols $a_1, a_2, \ldots a_n$. Let $b_1, b_2, \ldots b_n$ be the corresponding integers associated with these symbols. Then the *Gödel number* of M is the following integer:

$$r = \prod_{k=1}^{n} Pr(k)^{a_k}$$

We write $g_n(M) = r$, and $M = Exp(r)$. If M is the empty expression, we let $g_n(M) = 1$.

Definition 5. Let $M_1, M_2, \ldots M_n$ be a finite sequence of expressions. Then, the Gödel number of this sequence of expressions is the following integer:

$$r = \prod_{k=1}^{n} Pr(k)^{g_n(M_k)}$$

It is easy to prove that any expression and any sequence of expressions have a unique Gödel number. Since TM programs are sets of instructions not lists of them, any TM program consisting of n instructions has n! Gödel numbers.

Definition 6. For each n>0, let $T_n(z, x_1, \ldots x_n, y)$ be the predicate that means, for given $z, x_1, \ldots x_n, y$, that z is a Gödel number of a TM program Z, and that y is the Gödel number of a computation, with respect to Z, beginning with snapshot $q_1(\mathbf{x_1}, \ldots \mathbf{x_n})$.

These predicates express the essential elements of the theory of TM programs. Davis 1958 contains the detailed construction proving that for each n>0, $T_n(z, x_1, \ldots x_n, y)$ is primitive recursive, that every Turing-computable function is partial recursive, and that every total Turing-computable function is total recursive.

2.5. Universal TM programs

We are now ready to demonstrate that there are universal TMs, which compute any function computable by a TM. Consider the partial recursive binary function $f(z, x) = u_{1^1}(\min_y T(z, x, y))$. Since this function is Turing-computable, there is a TM program U such that:

$$\psi_{U^2}(z, x) = f(z, x)$$

This program is called *universal* TM program. It can be employed to compute any partially computable (singulary; but generalizable to n-ary) function as follows: If Z_0 is any TM program and if z_0 is a Gödel number of Z_0, then:

$$\psi_{U^2}(z_0, x) = \psi_{Z_0}(x)$$

Thus, if the number z_0 is written on the tape of U, followed by the number x_0, U will compute the number $\psi_{Z_0}(x_0)$

2.6. Unsolvability of the halting problem

We now discuss the function HALT(x, y), defined as follows. For a given y, let P be the TM program such that $g_n(P) = y$. Then HALT(x, y) = 1 if $\Psi_P(x)$ is defined and HALT(x, y) = 0 otherwise. In other words HALT(x, y) = 1 if and only if the TM program with Gödel number y eventually halts on input x, otherwise it's equal to 0. We now prove the unsolvability of the halting problem.

Theorem. HALT(x, y) is not a recursive function.

Proof. Define the total function g(x) = HALT(x, x), and the partial function h(x) = 0 if g(x) = 0, h(x) undefined if g(x) = 1. If h is partial recursive, then there is a TM program P^1 with Gödel number i such that for all x, $h(x) = \Psi_{P^1}(x)$. But then:

$$h(i) = \psi_{P^1}(i) = 0 \text{ if and only if } g(i) = 0 \text{ if and only if} \psi_{P^1}(i) \text{is undefined},$$

which is a contradiction. Therefore, h cannot be partial recursive, so that g and hence HALT cannot be total recursive. QED.

This theorem gives us an example of a function that is not computable by a TM program. Computability theory shows that there are infinitely many such functions. Assuming the truth of the Church-Turing thesis, which will be discussed in the next section, we conclude that there is no algorithm computing the halting function. The same holds for any other non-Turing-computable total function.

3. The Church-Turing thesis

Turing (1936–7) introduced his machines as a way to make precise the informal notion of algorithmic computability or effective calculability, as I introduced them in the first section. Church (1936) proposed a similar thesis using recursive functions, which, as we've seen, are computationally equivalent to TMs. After its proponents, Stephen Kleene (1952) dubbed this the Church-Turing thesis:

(CT) Any function that is effectively calculable is Turing-computable.

CT is generally accepted among mathematicians and computer scientists on what they consider overwhelming evidence in its favor.

In summary, this appendix introduced the informal notion of effective calculability and its formal counterpart—Turing computability. According to the canonical view, CT connects the informal notion of effective calculability, or computability by effective procedure, with the formal one of Turing-computability. There can be no rigorous proof of CT, but there is overwhelming evidence in its favor.

Bibliography

Abbott, A. A., C. S. Calude, J. Conder, and K. Svozil (2012). "Strong Kochen-Specker theorem and incomputability of quantum randomness." *Physical Review A* 86(6): 062109.

Adams, F. (1979). "A Goal-State Theory of Function Attributions." *Canadian Journal of Philosophy* 9(3): 493–518.

Aizawa, K. and C. Gillett (ms.). "Multiple Realization and Methodology in Neuroscience and Psychology."

Albert, D. A., R. Munson, and M. D. Resnik (1988). *Reasoning in Medicine: An Introduction to Clinical Infere*nce. Baltimore: The Johns Hopkins University Press.

Andréka, H., I. Németi, and P. Németi (2009). "General Relativistic Hypercomputing and Foundation of Mathematics." *Natural Computing* 8: 499–516.

Arkoudas, K. (2008). "Computation, Hypercomputation, and Physical Science." *Journal of Applied Logic* 6: 461–75.

Armstrong, D. M. (2010). *Sketch for a Systematic Metaphysics*. Oxford, UK: Oxford University Press.

Aspray, W. (1990). *John von Neumann and the Origins of Modern Computing*. Cambridge, MA: MIT Press.

Atanasoff, J. V. (1940). *Computing Machine for the Solution of Large Systems of Linear Algebraic Equations*. Ames, Iowa: Iowa State College.

Atanasoff, J. V. (1984). "Advent of Electronic Digital Computing." *Annals of the History of Computing* 6(3): 229–82.

Baddeley, R., Hancock, P., et al. (Eds.), (2000). *Information Theory and the Brain*. Cambridge: Cambridge University Press.

Balaguer, M. (2009). "Platonism in Metaphysics." In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy* (Summer 2009 Edition), URL = <http://plato.stanford.edu/archives/sum2009/entries/platonism/>. Stanford, CA: The Metaphysics Research Lab, Centre for the Study of Language and Information, Stanford University.

Balaguer, M. (2013). "Fictionalism in the Philosophy of Mathematics." In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy* (Fall 2013 Edition), URL = <http://plato.stanford.edu/archives/fall2013/entries/fictionalism-mathematics/>. Stanford, CA: The Metaphysics Research Lab, Centre for the Study of Language and Information, Stanford University.

Barwise, J. and J. Seligman (1997). *Information Flow: The Logic of Distributed Systems*. Cambridge: Cambridge University Press.

Bechtel, W. (2008). *Mental Mechanisms: Philosophical Perspectives on Cognitive Neuroscience*. London, Routledge.

Bechtel, W. and A. Abrahamsen (2002). *Connectionism and the Mind: Parallel Processing, Dynamics, and Evolution in Networks*. Malden, MA: Blackwell.

Bechtel, W. and J. Mundale (1999). "Multiple Realizability Revisited: Linking Cognitive and Neural States." *Philosophy of Science* 66: 175–207.

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

Beggs, E. J. and J.V. Tucker (2007). "Can Newtonian Systems, Bounded in Space, Time, Mass and Energy Compute all Functions?" *Theoretical Computer Science* 371: 4–19.

Bigelow, J. and R. Pargetter (1987). "Functions." *The Journal of Philosophy* 86: 181–96.

Black, R. (2000). "Proving Church's Thesis." *Philosophia Mathematica* 8: 244–58.

Blanchowicz, J. (1997). "Analog Representation beyond Mental Imagery." *The Journal of Philosophy* 94(2): 55–84.

Block, N. (1986). "Advertisement for a Semantics for Psychology." In P. French, T.E. Uehling, jr, and H.K. Wettstein (Eds.), *Midwest Studies in Philosophy X: Studies in the Philosophy of Mind*. Minneapolis: University of Minnesota Press.

Block, N. (1997). "Anti-Reductionism Slaps Back." In *Mind, Causation, World, Philosophical Perspectives* 11: 107–33.

Block, N. and J. A. Fodor (1972). "What Psychological States Are Not." *Philosophical Review* 81(2): 159–81.

Blum, L., F. Cucker, M. Shub, and S. Smale (1998). *Complexity and Real Computation*. New York: Springer.

Bontly, T. (1998). "Individualism and the Nature of Syntactic States." *British Journal for the Philosophy of Science* 49: 557–74.

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

Boshernitzan, M. (1986). "Universal Formulae and Universal Differential Equations." *The Annals of Mathematics, 2nd Series* 124(2): 273–91.

Bowie, G. L. (1973). "An Argument Against Church's Thesis." *The Journal of Philosophy* 70: 66–76.

Bradbury J. W., Vehrencamp S.L. (2000). "Economic models of animal communication." *Animal Behavior* 59(2): 259–68.

Brennecke, A. (2000). "Hardware Components and Computer Design." In R. Rojas and U. Hashagen (Eds.), *The First Computers-History and Architectures* (53–68). Cambridge, MA: MIT Press.

Bromley, A. G. (1983). "What Defines a 'General-Purpose' Computer?" *Annals of the History of Computing* 5(3): 303–5.

Brouwer, L. E. J. (1975). *Collected Works, Vol. 1*. Amsterdam: North-Holland.

Brown, C. (2012). "Combinatorial-State Automata and Models of Computation." *Journal of Cognitive Science* 13(1): 51–73.

Bub, J. (2005). "Quantum Mechanics is About Quantum Information." *Foundations of Physics* 35(4): 541–60.

Bueno, O. (2009). "Mathematical Fictionalism," in O. Bueno and O. Linnebo (Eds.), *New Waves in Philosophy of Mathematics* (59–79). Basingstoke: Palgrave Macmillan.

Buller, D. J. (1998). "Etiological Theories of Function: A Geographical Survey." *Biology and Philosophy* 13(4), 505–27.

Burge, T. (1986). "Individualism and Psychology." *Philosophical Review* 95: 3–45.

Burge, T. (2010). *Origins of Objectivity*. Oxford: Oxford University Press.

Burks, A. R. (2002). *Who Invented the Computer?* Amherst: Prometheus.

Burks, A. R. and A. W. Burks (1988). *The First Electronic Computer: The Atanasoff Story*. Ann Arbor: University of Michigan Press.

Button, T. (2009). "SAD Computers and Two Versions of the Church-Turing Thesis." *British Journal for the Philosophy of Science* 60(4): 765–92.

Calude, C. S. (2005). "Algorithmic Randomness, Quantum Physics, and Incompleteness." In M. Margenstern (Ed.), *Proceedings of the Conference "Machines, Computations and Universality." (MCU 2004)* (1–17). Berlin: Springer.

Calude, C. S. and Pavlov, B. (2002). "Coins, Quantum Measurements, and Turing's Barrier." *Quantum Information Processing* 1(1–2): 107–27.

Chalmers, D. J. (1994). "On Implementing a Computation." *Minds and Machines* 4: 391–402.

Chalmers, D. J. (1996a). *The Conscious Mind: In Search of a Fundamental Theory*. Oxford: Oxford University Press.

Chalmers, D. J. (1996b). "Does a Rock Implement Every Finite-State Automaton?" *Synthese* 108: 310–33.

Chalmers, D. J. (2011). "A Computational Foundation for the Study of Cognition." *Journal of Cognitive Science* 12(4): 323–57.

Chalmers, D. J. (2012). "The Varieties of Computation: A Reply." *Journal of Cognitive Science* 13.3: 211–48.

Care, C. (2010). *Technology for Modelling: Electrical Analogies, Engineering Practice, and the Development of Analogue Computing*. Berlin: Springer.

Chen, T. and H. Chen (1993). "Approximations of Continuous Functionals by Neural Networks with Application to Dynamic Systems." *IEEE Transactions on Neural Networks* 4(6): 910–18.

Chrisley, R. L. (1995). "Why Everything Doesn't Realize Every Computation." *Minds and Machines* 4: 403–30.

Christensen, W. D. and M. H. Bickhard (2002). "The Process Dynamics of Normative Function." *The Monist* 85(1): 3–28.

Church, A. (1932). "A Set of Postulates for the Foundation of Logic." *Annals of Mathematics* 33: 346–66.

Church, A. (1936). "An Unsolvable Problem in Elementary Number Theory." *The American Journal of Mathematics* 58: 345–63.

Church, A. (1940). "On the Concept of a Random Sequence." *American Mathematical Society Bulletin* 46: 130–5.

Church, A. (1956). *Introduction to Mathematical Logic*. Princeton: Princeton University Press.

Churchland, P. M. (1979). *Scientific Realism and the Plasticity of Mind*. Cambridge: Cambridge University Press.

Churchland, P. M. (1989). *A Neurocomputational Perspective*. Cambridge, MA: MIT Press.

Churchland, P. M. and P. S. Churchland (1990). "Could a Machine Think?" *Scientific American* 262: 26–31.

Churchland, P. S. (1986). *Neurophilosophy*. Cambridge, MA: MIT Press.

Churchland, P. S., C. Koch, and T. J. Sejnowski (1990). "What is Computational Neuroscience?" In E. L. Schwartz (Ed.), *Computational Neuroscience* (46–55). Cambridge, MA: MIT Press.

Churchland, P. S. and T.J. Sejnowski (1992). *The Computational Brain*. Cambridge, MA: MIT Press.

Cleland, C. E. (1993). "Is the Church-Turing Thesis True?" *Minds and Machines* 3: 283–312.

Cohen, I. B. (1999). *Howard Aiken: Portrait of a Computer Pioneer*. Cambridge, MA: MIT Press.

Cohen, I. B. (2000). "Howard Aiken and the Dawn of the Computer Age." In R. Rojas and U. Hashagen (Eds.), *The First Computers-History and Architectures*. Cambridge, MA: MIT Press: 107–20.

Cohen, J. and A. Meskin (2006). "An objective counterfactual theory of information." *Australasian Journal of Philosophy* 84: 333–52.

Collins, J., N. Hall, and L. A. Paul (Eds.), (2004). *Causation and Counterfactuals*. Cambridge, MA: MIT Press.

Copeland, B. J. (1996). "What is Computation?" *Synthese* 108: 224–359.

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* 96(1): 5–32.

Copeland, B. J. (2002a). "Accelerating Turing Machines." *Minds and Machines* 12(2): 281–301.

Copeland, B. J. (2002b). "The Church-Turing Thesis." In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy* (Fall 2002 Edition), URL = <Http://plato.stanford.edu/archives/fall2002/entries/church-turing/>. Stanford, CA: The Metaphysics Research Lab, Centre for the Study of Language and Information, Stanford University.

Copeland, B. J. (2006). "Turing's Thesis." In A. Olszewski, J. Wolenski and R. Janusz (Eds.), *Church's Thesis after 70 Years* (147–74). Heusenstamm: Ontos.

Copeland, B. J., C. J. Posy, and O. Shagrir (2013). *Computability: Turing, Gödel, Church, and Beyond*. Cambridge, MA: MIT Press.

Copeland, J. and O. Shagrir (2007). "Physical Computation: How General Are Gandy's Principles for Mechanisms?" *Minds and Machines* 17: 217–231.

Corcoran, J., W. Frank, and M. Maloney (1974). "String Theory." *The Journal of Symbolic Logic* 39(4): 625–37.

Cotogno, P. (2003). "Hypercomputation and the Physical Church-Turing Thesis." *British Journal for the Philosophy of Science* 54: 181–223.

Cowan, J. D. (1990). "McCulloch-Pitts and Related Neural Nets from 1943 to 1989." *Bulletin of Mathematical Biology* 52(1/2): 73–97.

Crandall, B., G. Klein, et al. (2006). *Working Minds: A Practitioner's Guide to Task Analysis*. Cambridge, MA: MIT Press.

Crane, T. (1990). "The Language of Thought: No Syntax Without Semantics." *Mind and Language* 5(3): 187–212.

Craver, C. (2001). "Role Functions, Mechanisms, and Hierarchy." *Philosophy of Science* 68: 53–74.

Craver, C. F. (2004). "Dissociable Realization and Kind Splitting." *Philosophy of Science* 71(4): 960–71.

Craver, C. F. (2006). "When Mechanistic Models Explain." *Synthese* 153(3): 355–76.

Craver, C. F. (2007). *Explaining the Brain*. Oxford: Oxford University Press.

Craver, C. F. (2012). "Functions and Mechanisms: A Perspectivalist Account." In P. Huneman (Ed.), *Functions*. Dordrecht: Springer.

Craver, C. F. and L. Darden (2001). "Discovering Mechanisms in Neurobiology." In P. Machamer, R. Grush, and P. McLaughlin (Eds.), *Theory and Method in the Neurosciences* (112–37). Pittsburgh, PA: University of Pittsburgh Press.

Cummins, R. (1975). "Functional Analysis." *Journal of Philosophy* 72(20): 741–65.

Cummins, R. (1977). "Programs in the Explanation of Behavior." *Philosophy of Science* 44: 269–87.

Cummins, R. (1983). *The Nature of Psychological Explanation*. Cambridge, MA: MIT Press.

Cummins, R. (1989). *Meaning and Mental Representation*. Cambridge, MA: MIT Press.

Cummins, R. (2000). "'How does it work?' vs. 'What are the laws?' Two Conceptions of Psychological Explanation." In F. C. Keil and R. A. Wilson (Eds.), *Explanation and Cognition*. Cambridge, MA: MIT Press.

Cummins, R. (2002). "Neo-teleology." In A. Ariew, R. Cummins and M. Perlman (Eds.), *Functions: New Essays in the Philosophy of Psychology and Biology*. Oxford: Oxford University Press.

Cummins, R. and G. Schwarz (1991). "Connectionism, Computation, and Cognition." In T. Horgan and J. Tienson (Eds.), *Connectionism and the Philosophy of Mind* (60–73). Dordrecht: Kluwer.

Davidson, D. (1970). "Mental Events." In L. Foster and J. W. Swanson (Eds.), *Experience and Theory*. Amherst, MA: University of Massachusetts Press. Reprinted in D. Davidson (1980), *Essays on Actions and Events*. Oxford: Clarendon Press.

Davies, E. B. (2001). "Building Infinite Machines." *British Journal for the Philosophy of Science* 52(4): 671–82.

Davis, M. (1958). *Computability and Unsolvability*. New York: McGraw-Hill.

Davis, M. (1982). "Why Gödel Didn't Have Church's Thesis." *Information and Control* 54: 3–24.

Davis, M. (2004a). "The Myth of Hypercomputation." In C. Teuscher (Ed.), *Alan Turing: Life and Legacy of a Great Thinker* (195–211). Berlin: Springer.

Davis, M., Ed. (2004b). *The Undecidable: Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions*. Dover: Mineola.

Davis, M. (2006). "The Church-Turing Thesis: Consensus and Opposition." In A. Beckmann, U. Berger, B. Löwe, and J. V. Tucker (Eds.), *Logical Approaches to Computational Barriers: Second Conference on Computability in Europe, CiE 2006, Swansea, UK, July 2006, Proceedings* (125–32). Berlin: Springer-Verlag.

Davis, M., R. Sigal, and E. J. Weyuker (1994). *Computability, Complexity, and Languages*. Boston: Academic.

Dayan, P. and L. F. Abbott (2001). *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*. Cambridge, MA: MIT Press.

Dennett, D. C. (1969). *Content and Consciousness*. London: Routledge and Kegan Paul.

Dennett, D. C. (1975). "Why the Law of Effect Will Not Go Away." *Journal of the Theory of Social Behavior* 5: 169–87. Reprinted in Dennett 1978, 71–89.

Dennett, D. C. (1978). *Brainstorms*. Cambridge, MA: MIT Press.

Dennett, D. C. (1987). *The Intentional Stance*. Cambridge, MA: MIT Press.

Dennett, D. C. (1991). *Consciousness Explained*. Boston, MA: Little, Brown.

Dershowitz, N. and Y. Gurevich (2008). "A Natural Axiomatization of Computability and Proof of Church's Thesis." *The Bulletin of Symbolic Logic* 14(3): 299–350.

Deutsch, D. (1985). "Quantum Theory, the Church-Turing Principle and the Universal Quantum Computer," *Proceedings of the Royal Society of London A* 400: 97–117.

Deutsch, J. A. (1960). *The Structural Basis of Behavior*. Chicago: University of Chicago Press.

Devitt, M. (1990). "A Narrow Representational Theory of the Mind." In W. Lycan (Ed.), *Mind and Cognition: A Reader*. New York: Blackwell.

Devitt, M. and K. Sterelny (1999). *Language and Reality: An Introduction to the Philosophy of Language*. Cambridge, MA: MIT Press.

Dewhurst, J. (2014). "Mechanistic Miscomputation: A Reply to Fresco and Primiero." *Philosophy and Technology* 27(3): 495–8.

Dewhurst, J. (ms.). "Rejecting the Received View: Representation, Computation, and Observed-Relativity."

Dietrich, E. (1989). "Semantics and the Computational Paradigm in Cognitive Psychology." *Synthese* 79: 119–41.

Dresner, E. (2003). "'Effective Memory' and Turing's Model of Mind." *Journal of Experimental and Theoretical Artificial Intelligence* 15(1): 113–23.

Dretske, F. I. (1981). *Knowledge and the Flow of Information*. Cambridge, MA: MIT Press.

Dretske, F. I. (1986). "Misrepresentation." In R. Bogdan (Ed.), *Belief: Form, Content, and Function* (17–36). New York: Oxford University Press.

Dretske, F. I. (1988). *Explaining Behavior*. Cambridge, MA: MIT Press.

Dreyfus, H. L. (1979). *What Computers Can't Do*. New York: Harper & Row.

Duffin, R. J. (1981). "Rubel's Universal Differential Equation." *Proceedings of the National Academy of Sciences USA* 78(8): 4661–2.

Earman, J. (1986). *A Primer on Determinism*. Dordrecht: D. Reidel.

Earman, J. and J. Norton (1993). "Forever is a Day: Supertasks in Pitowsky and Malament-Hogarth Spacetimes." *Philosophy of Science* 60: 22–42.

Edelman, G. M. (1992). *Bright Air, Brilliant Fire: On the Matter of the Mind*. New York: Basic Books.

Edelman, S. (2008). *Computing the Mind: How the Mind Really Works*. Oxford: Oxford University Press.

Egan, F. (1992). "Individualism, Computation, and Perceptual Content." *Mind* 101(403): 443–59.

Egan, F. (1995). "Computation and Content." *Philosophical Review* 104: 181–203.

Egan, F. (1999). "In Defence of Narrow Mindedness." *Mind and Language* 14(2): 177–94.

Egan, F. (2003). "Naturalistic Inquiry: Where does Mental Representation Fit in?" In L. M. Antony and N. Hornstein (Eds.), *Chomsky and His Critics* (89–104). Malden, MA: Blackwell.

Egan, F. (2010). "Computational Models: A Modest Role for Content." *Studies in the History and Philosophy of Science* 41(3): 253–9.

Egan, F. (2014). "How to Think about Mental Content." *Philosophical Studies* 170: 115–35.

Eliasmith, C. (2003). "Moving Beyond Metaphors: Understanding the Mind for What It Is." *Journal of Philosophy* 100: 493–520.

Engelsohn, H. S. (1978). *Programming Programmable Calculators*. Rochelle Park, NJ: Hayden.

Ermentrout, G. B. and D. H. Terman (2010). *Mathematical Foundations of Neuroscience*. Berlin: Springer.

Etesi, G. and I. Németi (2002). "Non-Turing Computations via Malament-Hogarth Spacetimes." *International Journal of Theoretical Physics* 41: 342–70.

Fano, R. M. (1961). *Transmission of Information: A Statistical Theory of Communications*. New York: MIT Press.

Feest, U. (2003). "Functional Analysis and the Autonomy of Psychology." *Philosophy of Science* 70: 937–48.

Feldman, J. A. and Ballard, D. H. (1982). "Connectionist Models and their Properties." *Cognitive Science* 6: 205–54. Reprinted in Anderson & Rosenfeld, 1988, 484–508.

Feynman, R. P. (1982). "Simulating Physics with Computers." *International Journal of Theoretical Physics* 21(6–7): 467–88.

Field, H. (1978). "Mental Representation." *Erkenntnis* 13: 9–61.

Field, H. (1980). *Mathematics without Numbers: A Defence of Nominalism*. Oxford: Blackwell.

Field, H. (1989). *Realism, Mathematics, and Modality*. New York, NY: Blackwell.

Fisher, R. (1935). *The Design of Experiments*. Edinburgh: Oliver and Boyd.

Fitz, H. (2006). "Church's Thesis and Physical Computation." In A. Olszewski, J. Wolenski, and R. Janusz (Eds.), *Church's Thesis after 70 Years* (175–219). Heusenstamm: Ontos.

Floridi, L. (2005). "Is semantic information meaningful data?" *Philosophy and Phenomenological Research* 70(2): 351–70.

Fodor, J. A. (1965). "Explanations in Psychology." In M. Black (Ed.), *Philosophy in America*. London: Routledge and Kegan Paul.

Fodor, J. A. (1968a). "The Appeal to Tacit Knowledge in Psychological Explanation." *Journal of Philosophy* 65: 627–40.

Fodor, J. A. (1968b). *Psychological Explanation*. New York, Random House.

Fodor, J. A. (1974). "Special Sciences." *Synthese* 28: 77–115.

Fodor, J. A. (1975). *The Language of Thought*. Cambridge, MA: Harvard University Press.

Fodor, J. A. (1978). "Tom Swift and His Procedural Grandmother." *Cognition* 6; 229–47.

Fodor, J. A. (1980). "Methodological Solipsism Considered as a Research Strategy in Cognitive Psychology." *Behavioral and Brain Sciences* 3(1).

Fodor, J. A. (1981). "The Mind-Body Problem." *Scientific American* 244 (January 1981). Reprinted in J. Heil, (Ed.) (2004a), *Philosophy of Mind: A Guide and Anthology* (168–82). Oxford: Oxford University Press.

Fodor, J. A. (1983). *The Modularity of Mind*. Cambridge, MA: MIT Press.

Fodor, J. A. (1987). *Psychosemantics*. Cambridge, MA: MIT Press.

Fodor, J. A., Pylyshyn, Z. W. (1988). "Connectionism and cognitive architecture." *Cognition* 28: 3–71.

Fodor, J. A. (1990). *A Theory of Content and Other Essays*. Cambridge, MA: MIT Press.

Fodor, J. A. (1997). "Special Sciences: Still Autonomous after All These Years." *Philosophical Perspectives, 11, Mind, Causation, and the World*: 149–63.

Fodor, J. A. (1998). *Concepts*. Oxford: Clarendon Press.

Fodor, J. A. (2008). *LOT 2: The Language of Thought Revisited*. Oxford: Oxford University Press.

Folina, J. (1998). "Church's Thesis: Prelude to a Proof." *Philosophia Mathematica* 6: 302–23.

Frank, R. (1994). "Instruments, Nerve Action, and the All-or-None Principle." *Osiris* 9: 208–35.

Franklin, S. and Garzon, M. (1990). "Neural Computability." In O. Omidvar (Ed.), *Progress in Neural Networks* (127–45). Norwood, NJ: Ablex.

Fredkin, E. (1990). "Digital Mechanics: An Information Process Based on Reversible Universal Cellular Automata." *Physica D* 45: 254–70.

Freeman, W. J. (2001). *How Brains Make Up Their Minds*. New York: Columbia University Press.

Fresco, N. (2013). *Physical Computation and Cognitive Science*. New York, NY: Springer.

Fresco, N. and G. Primiero (2013). "Miscomputation." *Philosophy & Technology* 26, 253–72.

Fresco, N. and M. J. Wolf (2014). "The Instructional Information Processing Account of Digital Computation." *Synthese* 191(7): 1469–92.

Fuchs, C. A. (2004). "Quantum Mechanics as Quantum Information (and only a little more)." In A. Khrennikov (Ed.), *Quantum Theory: Reconsiderations of Foundations* (463–543). Växjö, Sweden: Växjö University Press.

Gallistel, C. R. and J. Gibbon (2002). *The Symbolic Foundations of Conditioned Behavior*. Mahwah, NJ: Lawrence Erlbaum Associates.

Galton, A. (2006). "The Church–Turing Thesis: Still Valid After All These Years?" *Applied Mathematics and Computation* 178: 93–102.

Gandy, R. (1980). "Church's Thesis and Principles for Mechanism." In J. Barwise, H. J. Keisler, and K. Kuhnen (Eds.), *The Kleene Symposium* (123–48). Amsterdam: North-Holland.

Gandy, R. (1988). "The Confluence of Ideas in 1936." In R. Herken (Ed.), *The Universal Machine: A Half-Century Survey* (55–111). New York: Oxford University Press.

Garson, J. (2003). "The Introduction of Information into Neurobiology." *Philosophy of Science* 70: 926–36.

Garson, J. (2011). "Selected Effects Functions and Causal Role Functions in the Brain: The Case for an Etiological Approach to Neuroscience." *Biology & Philosophy* 26: 547–65.

Garson, J. (2013). "The Functional Sense of Mechanism." *Philosophy of Science* 80: 317–33.

Garson, J. and G. Piccinini (2014). "Functions Must Be Performed at Appropriate Rates in Appropriate Situations." *The British Journal for the Philosophy of Science* 65(1): 1–20.

Glennan, S. (2002). "Rethinking Mechanistic Explanation." *Philosophy of Science* 69: S342–53.

Glennan, S. (2005). "Modeling Mechanisms." *Studies in History and Philosophy of Biological and Biomedical Sciences* 36(2): 443–64.

Globus, G. G. (1992). "Towards a Noncomputational Cognitive Neuroscience." *Journal of Cognitive Neuroscience* 4(4): 299–310.

Gödel, K. (1931). "On Formally Undecidable Propositions of Principia Mathematica and Related Systems I." *Monaschefte für Mathematik und Physik* 38: 173–98.

Gödel, K. (1934). "On Undecidable Propositions of Formal Mathematical Systems." Reprinted in Davis 2004b, 41–71.

Gödel, K. (1936). "Über die Lange von Beweisen." *Ergebnisse eines mathematischen Kolloquiums* 7: 23–4.

Gödel, K. (1946). "Remarks Before the Princeton Bicentennial Conference on Problems in Mathematics." Reprinted in Davis 2004b, 84–8.

Gödel, K. (1965). "Postscriptum." In Davis 2004b, 71–3.

Godfrey-Smith, P. (1993). "Functions: Consensus without unity." *Pacific Philosophical Quarterly* 74: 196–208.

Godfrey-Smith, P. (1994). "A modern history theory of functions." *Noûs* 28(3): 344–62.

Godfrey-Smith, P. (2000). "On the Theoretical Role of 'Genetic Coding'." *Philosophy of Science* 67: 26–44.

Godfrey-Smith, P. (2009). "Triviality Arguments against Functionalism." *Philosophical Studies* 145(2): 273–95.

Goldstine, H. H. and J. von Neumann (1946). "On the Principles of Large Scale Computing Machines." Princeton: Institute for Advanced Studies. Reprinted in John von Neumann, *Collected Works, Vol. V*, Oxford: Pergamon, 1–32.

Goodman, N. (1968). *Languages of Art*. Indianapolis: Bobbs-Merrill.

Gould, S. J. (1974). "The Origin and Function of 'Bizarre' Structures: Antler Size and Skull Size in the 'Irish Elk,' Megaloceros giganteus." *Evolution* 28(2): 191–220.

Grice, H. P. (1957). "Meaning." *The Philosophical Review* 66(3): 377–88.

Griffiths, P. E. (1993). "Functional Analysis and Proper Functions." *British Journal for the Philosophy of Science* 44: 409–22.

Griffiths, P. E. (2001). "Genetic information: A metaphor in search of a theory." *Philosophy of Science* 68(3): 394–412.

Griffiths, P. E. (2009). "In What Sense Does 'Nothing Make Sense Except in the Light of Evolution'?"*Acta Biotheoretica* 57: 11–32.

Grush, R. (2004). "The Emulation Theory of Representation: Motor Control, Imagery, and Perception." *Behavioral and Brain Sciences* 27(3): 377–442.

Gustafson, J. (2000). "Reconstruction of the Atanasoff-Berry Computer." In R. Rojas and U. Hashagen (Eds.), *The First Computers-History and Architectures* (91–106). Cambridge, MA: MIT Press.

Hagar, A. and A. Korolev (2007). "Quantum Hypercomputation–Hyper or Computation?" *Philosophy of Science* 74(3): 347–63.

Haimovici, S. (2013). "A Problem for the Mechanistic Account of Computation." *Journal of Cognitive Science* 14: 151–81.

Hardcastle, V. G. (1999). "Understanding Functions: A Pragmatic Approach." In V. G. Hardcastle (Ed.), *When Biology Meets Philosophy: Philosophical Essays* (27–46). Cambridge, MA: MIT Press.

Harman, G. (1973). *Thought*. Princeton: Princeton University Press.

Harman, G. (1987). "(Nonsolipsistic) Conceptual Role Semantics." In E. Lepore (Ed.), *New Directions in Semantics* (55–81). London: Academic Press.

Harman, G. (1988). "Wide Functionalism." In S. Schiffer and S. Steele (Eds.), *Cognition and Representation* (11–20). Boulder: Westview.

Harman, G. (1999). *Reasoning, Meaning and Mind*. Oxford: Clarendon Press.

Hartley, R. and Szu, H. (1987). "A Comparison of the Computational Power of Neural Network Models." In *Proceedings of the IEEE First International Conference on Neural Networks* (15–22), San Diego: IEEE Press.

Hartley, R. V. (1928). "Transmission of information." *AT&T Technical Journal* 7: 535–63.

Haugeland, J. (1978). "The Nature and Plausibility of Cognitivism." *Behavioral and Brain Sciences* 2: 215–60.

Haugeland, J. (1981). "Analog and Analog." *Philosophical Topics* 12: 213–25.

Haugeland, J. (1997). *Mind Design II*. Cambridge, MA: MIT Press.

Heil, J. (2003). *From an Ontological Point of View*. Oxford: Clarendon Press.

Heil, J. (2012). *The Universe as We Find It*. Oxford: Oxford University Press.

Hilbert, D. and W. Ackermann (1928). *Grunzüge der theoretischen Logik*. Berlin: Springer.

Hodges, A. (2005). 'Can quantum computing solve classically unsolvable problems?', <arxiv.org/pdf/quant-ph/0512248>.

Hodges, A. (2006). "Did Church and Turing Have a Thesis About Machines?" In A. Olszewski, J. Wolenski, and R. Janusz (Eds.), *Church's Thesis after 70 Years* (242–52). Heusenstamm: Ontos.

Hogarth, M. L. (1994). "Non-Turing Computers and Non-Turing Computability." *PSA 1994*1: 126–38.

Hogarth, M. L. (2004). "Deciding Arithmetic Using SAD Computers." *British Journal for the Philosophy of Science* 55: 681–91.

Hong, J. (1988). "On Connectionist Models." *Communications on Pure and Applied Mathematics* 41: 1039–50.

Hopfield, J. J. (1982). "Neural Networks and Physical Systems with Emergent Collective Computational Abilities." *Proceedings of the National Academy of Sciences* 79: 2554–8. Reprinted in Anderson and Rosenfeld (1988). *Neurocomputing: Foundations of Research*. Cambridge, MA: MIT Press, 460–4.

Horgan, T. (1997). "Connectionism and the Philosophical Foundations of Cognitive Science." *Metaphilosophy* 28: 1–30.

Horgan, T. and J. Tienson (1989). "Representation Without Rules." Philosophical Topics. 17(1): 27–43.

Horgan, T. and J. Tienson (1996). *Connectionism and the Philosophy of Psychology*. Cambridge, MA: MIT Press.

Horst, S. W. (1996). *Symbols, Computation, and Intentionality: A Critique of the Computational Theory of Mind*. Berkeley: University of California Press.

Houkes, W. and P. E. Vermaas (2010). *Technical Functions: On the Use and Design of Artefacts*. Berlin: Springer.

Householder, A. S. and H. D. Landahl (1945). *Mathematical Biophysics of the Central Nervous System*. Bloomington: Principia.

Hughes, R. I. G. (1999). "The Ising Model, Computer Simulation, and Universal Physics." In M. S. Morgan and M. Morrison (Eds.), *Models as Mediators* (97–145). Cambridge: Cambridge University Press.

Humphreys, P. (2004). *Extending Ourselves: Computational Science, Empiricism, and Scientific Method*. Oxford: Oxford University Press.

Jackson, A. S. (1960). *Analog Computation*. New York: McGraw-Hill.

Jacquette, D. (1991). "The Myth of Pure Syntax." In L. Albertazzi and R. Rolli (Eds.), *Topics in Philosophy and Artificial Intelligence* (1–14). Bozen: Istituto Mitteleuropeo di Cultura.

Johnson, C. L. (1963). *Analog Computer Techniques*, Second Edition. New York: McGraw-Hill.

Kálmar, L. (1959). "An Argument Against the Plausibility of Church's Thesis." In A. Heyting (Ed.), *Constructivity in Mathematics* (72–80), Amsterdam: North-Holland.

Kantor, F. W. (1982). "An Informal Partial Overview of Information Mechanics." *International Journal of Theoretical Physics* 21(6–7): 525–35.

Kaplan, D. and C. F. Craver (2011). "The Explanatory Force of Dynamical and Mathematical Models in Neuroscience: A Mechanistic Perspective." *Philosophy of Science* 78(4): 601–27.

Katz, M. (2008). "Analog and Digital Representation." *Minds and Machines* 18: 403–8.

Keeley, B. (2000). "Shocking Lessons from Electric Fish: The Theory and Practice of Multiple Realizability." *Philosophy of Science* 67: 444–65.

Kieu, T. D. (2002). "Quantum Hypercomputation." *Minds and Machines* 12(4): 541–61.

Kieu, T. D. (2003). "Computing the Noncomputable," *Contemporary Physics* 44: 51–71.

Kieu, T. D. (2004). "A Reformulation of Hilbert's Tenth Problem through Quantum Mechanics." *Proceedings of the Royal Society A*, 460(2045): 1535–45.

Kieu, T. D. (2005). "An Anatomy of a Quantum Adiabatic Algorithm that Transcends the Turing Computability." *International Journal of Quantum Information* 3(1): 177–83.

Kim, J. (1992). "Multiple Realization and the Metaphysics of Reduction." *Philosophy and Phenomenological Research* 52: 1–26.

Kim, J. (2005). *Physicalism, Or Something Near Enough*. Princeton, NJ: Princeton University Press.

Kitcher, P. (1985). "Narrow Taxonomy and Wide Functionalism." *Philosophy of Science* 52(1): 78–97.

Kleene, S. C. (1935). "A Theory of Positive Integers in Formal Logic." *American Journal of Mathematics* 57: 153–73 and 219–44.

Kleene, S. C. (1952). *Introduction to Metamathematics*. Princeton: Van Nostrand.

Kleene, S. C. (1956). "Representation of Events in Nerve Nets and Finite Automata." In C. E. Shannon and J. McCarthy (Eds.), *Automata Studies* (3–42). Princeton, NJ: Princeton University Press.

Kleene, S. C. (1987a). "Gödel's impression on students of logic in the 1930s." In P. Weingartner and L. Schmetterer (Eds.), *Gödel Remembered* (pp. 49–64). Napoli: Bibliopolis.

Kleene, S. C. (1987b). "Reflections on Church's Thesis." *Notre Dame Journal of Formal Logic* 28: 490–8.

Klein, C. (2008). "Dispositional Implementation Solves the Superfluous Structure Problem." *Synthese* 165: 141–53.

Koch, C. (1999). *Biophysics of Computation: Information Processing in Single Neurons*. New York: Oxford University Press.

Korn, G. A. and T. M. Korn (1972). *Electronic Analog and Hybrid Computers*; Second, Completely Revised Edition. New York: McGraw-Hill.

Kripke, S. (2013). "The Church-Turing 'Thesis' as a Special Corollary of Gödel's Completeness Theorem." In Copeland, Posy, and Shagrir (2013), 77–104.

Leff, H. S. and A. F. Rex (Eds.), (2003). *Maxwell's Demon 2: Entropy, Classical and Quantum Information, Computing*. Bristol: Institute of Physics Publishing.

Leng, M. (2010). *Mathematics and Reality*. New York, NY: Oxford University Press.

Lewens, T. (2004). *Organisms and Artifacts*. Cambridge, MA: MIT Press.

Lewis, D. K. (1971). "Analog and Digital." *Noûs* 5: 321–7.

Lewis, D. K. (1986). "Postscript to 'Causation'." In D. Lewis, *Philosophical Papers*, Vol. 2 (172–213). New York: Oxford University Press.

Li, M. and P. Vitányi (1997). *An Introduction to Kolmogorov Complexity and Its Applications*, Second Edition. New York: Springer.

Linnebo, Ø. (2011). "Platonism in the Philosophy of Mathematics", *The Stanford Encyclopedia of Philosophy* (Fall 2011 Edition), Edward N. Zalta (Ed.), URL = <http://plato.stanford.edu/archives/fall2011/entries/platonism-mathematics/>. Stanford, CA: The Metaphysics Research Lab, Centre for the Study of Language and Information, Stanford University.

Lipshitz, L. and L. A. Rubel (1987). "A Differentially Algebraic Replacement Theorem, and Analog Computability." *Proceedings of the American Mathematical Society* 99(2): 367–72.

Lloyd, S. (2006). *Programming the Universe: A Quantum Computer Scientist Takes on the Cosmos*. New York: Knopf.

Loar, B. (1981). *Mind and Meaning*. Cambridge: Cambridge University Press.
Macdonald, C. and Macdonald, G. (1995). *Connectionism: Debates on Psychological Explanation*, Volume Two. Oxford: Blackwell.
Machamer, P. (2004). "Activities and Causation: The Metaphysics and Epistemology of Mechanisms." *International Studies in the Philosophy of Science* 18(1): 27–39.
Machamer, P. K., L. Darden, and C. F. Craver (2000). "Thinking About Mechanisms." *Philosophy of Science* 67: 1–25.
Machtey, M. and P. Young (1978). *An Introduction to the General Theory of Algorithms*. New York: North Holland.
Maley, C. J. (2011). "Analog and Digital, Continuous and Discrete." *Philosophical Studies* 155(1): 117–31.
Marr, D. (1982). *Vision*. New York: Freeman.
Marr, D. and T. Poggio (1976). "Cooperative computation of stereo disparity." *Science* 194 (4262): 283–7.
Martin, C. B. (1997). "On the Need for Properties: The Road to Pythagoreanism and Back." *Synthese* 112(2): 193–231.
Martin, C. B. (2007). *The Mind in Nature*. Oxford, UK: Oxford University Press.
Maudlin, T. (1989). "Computation and Consciousness." *Journal of Philosophy* 86(8): 407–32.
McCulloch, W. S. and W. H. Pitts (1943). "A Logical Calculus of the Ideas Immanent in Nervous Activity." *Bulletin of Mathematical Biophysics* 7: 115–33.
McLaughlin, P. (2001). *What Functions Explain: Functional Explanation and Self-reproducing Systems*. Cambridge: Cambridge University Press.
Mendelson, E. (1990). "Second Thoughts about Church's Thesis and Mathematical Proofs." *The Journal of Philosophy* 88: 225–33.
Milkowski, M. (2013). *Explaining the Computational Mind*. Cambridge, MA: MIT Press.
Miller, G.A. (1951). *Language and Communication*. New York: McGraw-Hill.
Miller, G. A., E. H. Galanter, and K. H. Pribram (1960). *Plans and the Structure of Behavior*. New York: Holt.
Millikan, R. G. (1984). *Language, Thought, and Other Biological Categories: New Foundations for Realism*. Cambridge, MA: MIT Press.
Millikan, R. G. (1989). "In Defense of Proper Functions." *Philosophy of Science* 56(2): 288–302.
Millikan, R. G. (1993). *White Queen Psychology and Other Essays for Alice*. Cambridge, MA: MIT Press.
Millikan, R. G. (2004). *Varieties of Meaning*. Cambridge, MA: MIT Press.
Minsky, M. L. (1967). *Computation: Finite and Infinite Machines*. Englewood Cliffs, NJ: Prentice-Hall.
Minsky, M. L. (1968). *Semantic Information Processing*. Cambridge, MA: MIT Press.
Minsky, M. L. and S. A. Papert (1988). *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA: MIT Press.
Moen, R. A., J. Pastor, and Y. Cohen (1999). "Antler Growth and Extinction of Irish Elk." *Evolutionary Ecology Research* 1: 235–49.
Mossio M, C. Saborido, and A. Moreno (2009). "An Organizational Account of Biological Functions." *British Journal for the Philosophy of Science* 60: 813–41.
Mundici, D. and W. Sieg (1995). "Paper Machines." *Philosophia Mathematica* 3: 5–30.

Nagel, E. (1977). "Functional Explanations in Biology." *The Journal of Philosophy* 74(5): 280–301.

Nanay, B. (2010). "A Modal Theory of Function." *Journal of Philosophy* 107: 412–31.

Neander, K. (1991). "Functions as Selected Effects: The Conceptual Analyst's Defense." *Philosophy of Science* 58(2): 168–84.

Nelson, R. J. (1982). *The Logic of Mind*. Dordrecht: Reidel.

Németi, I. and G. Dávid (2006). "Relativistic Computers and the Turing Barrier." *Journal of Applied Mathematics and Computation* 178(1): 118–42.

Newell, A. (1980). "Physical Symbol Systems." *Cognitive Science* 4: 135–83.

Newell, A. and H. A. Simon (1976). "Computer Science as an Empirical Enquiry: Symbols and Search." *Communications of the ACM* 19: 113–26.

Nielsen, M. A. (1997). "Computable Functions, Quantum Measurements, and Quantum Dynamics." *Physical Review Letters* 79(15): 2915–18.

Nielsen, M. A. and I. L. Chuang (2000). *Quantum Computation and Quantum Information*. New York: Cambridge University Press.

Norton, J. D. (2003). "Causation as Folk Science." *Philosophers' Imprint* 3(4): 1–22.

O'Brien, G. and J. Opie (2006). "How Do Connectionist Networks Compute?" *Cognitive Processing* 7: 30–41.

Odifreddi, P. (1989). *Classical Recursion Theory: The Theory of Functions and Sets of Natural Numbers*. Amsterdam: North-Holland.

Oppenheim, P. and H. Putnam (1958). "Unity of Science as a Working Hypothesis." In H. Feigl, M. Scriven, and G. Maxwell (Eds.), *Minnesota Studies in the Philosophy of Science, Volume II. Concepts, Theories, and the Mind-Body Problem* (3–36). Minneapolis: University of Minnesota Press.

Ord, T. (2006). "The Many Forms of Hypercomputation." *Applied Mathematics and Computation* 178(1): 143–53.

Ord, T. and T. D. Kieu (2005). "The Diagonal Method and Hypercomputation." *British Journal for the Philosophy of Science* 56: 147–56.

Papineau, D. (1987). *Reality and Representation*. Oxford: Blackwell.

Paseau, A. (2013). "Naturalism in the Philosophy of Mathematics." Edward N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy* (Summer 2013 Edition), URL = <http://plato.stanford.edu/archives/sum2013/entries/naturalism-mathematics/>. Stanford, CA: The Metaphysics Research Lab, Centre for the Study of Language and Information, Stanford University.

Patterson, D. A. and J. L. Hennessy (1998). *Computer Organization and Design: The Hardware/Software Interface*. San Francisco: Morgan Kauffman.

Peacocke, C. (1994a). "Content." In S. Guttenplan (Ed.), *A Companion to the Philosophy of Mind* (219–25). Oxford: Blackwell.

Peacocke, C. (1994b). "Content, Computation, and Externalism." *Mind and Language* 9: 303–35.

Peacocke, C. (1999). "Computation as Involving Content: A Response to Egan." *Mind and Language* 14(2): 195–202.

Penrose, R. (1994). *Shadows of the Mind*. Oxford: Oxford University Press.

Perkel, D. H. (1990). "Computational Neuroscience: Scope and Structure." In E. L. Schwartz (Ed.), *Computational Neuroscience* (38–45). Cambridge, MA: MIT Press.

Piccinini, G. (2003a). "Alan Turing and the Mathematical Objection." *Minds and Machines* 12(1): 23–48.

Piccinini, G. (2003b). *Computations and Computers in the Sciences of Mind and Brain*. Doctoral Dissertation, Pittsburgh, PA: University of Pittsburgh. URL = <http://etd.library.pitt.edu/ETD/available/etd-08132003-155121/>.

Piccinini, G. (2004a). "Functionalism, Computationalism, and Mental States." *Studies in the History and Philosophy of Science* 35(4): 811–33.

Piccinini, G. (2004b). "The First Computational Theory of Mind and Brain: A Close Look at McCulloch and Pitts's 'Logical Calculus of Ideas Immanent in Nervous Activity'." *Synthese* 141(2): 175–215.

Piccinini, G. (2005). "Review of M. Scheutz's Computationalism: New Directions." *Philosophical Psychology* 18: 387–91.

Piccinini, G. (2008). "Computation without Representation." *Philosophical Studies* 137(2): 205–41.

Piccinini, G. (2011). "Computationalism." In E. Margolis, R. Samuels, and S. P. Stich, *Oxford Handbook of Philosophy and Cognitive Science* (222–49). Oxford: Oxford University Press.

Piccinini, G. and S. Bahar (2013). "Neural Computation and the Computational Theory of Cognition." *Cognitive Science* 34: 453–88.

Piccinini, G. and C. Craver (2011). "Integrating Psychology and Neuroscience: Functional Analyses as Mechanism Sketches." *Synthese* 183: 283–311.

Piccinini, G. and C. J. Maley (2014). "The Metaphysics of Mind and the Multiple Sources of Multiple Realizability." In M. Sprevak and J. Kallestrup (Eds.), *New Waves in the Philosophy of Mind* (125–52). New York: Palgrave Macmillan.

Piccinini, G. and A. Scarantino (2011). "Information Processing, Computation, and Cognition." *Journal of Biological Physics*, 37(1), 1–38.

Piccinini, G. and O. Shagrir (2014). "Foundations of Computational Neuroscience." *Current Opinion in Neurobiology* 25: 25–30.

Pierce, J. R. (1980). *An Introduction to Information Theory*. New York: Dover Publications.

Pinker, S. (1997). *How the Mind Works*. New York: Norton.

Pitowsky, I. (1990). "The Physical Church Thesis and Physical Computational Complexity." *Iyyun* 39: 81–99.

Pitowsky, I. (2002). "Quantum Speed-Up of Computations." *Philosophy of Science* 69: S168–77.

Pitowsky, I. (2007). "From Logic to Physics: How the Meaning of Computation Changed Over Time." In S. B. Cooper, B. Lowe, and A. Sorbi, *Computation and Logic in the Real World, Proceedings of the Third Conference on Computability in Europe, CiE 2007, Siena, Italy, June 18–23, 2007, Lecture Notes in Computer Science 4497*. Berlin: Springer (621–31).

Potgieter, P. H. (2006). "Zeno Machines and Hypercomputation." *Theoretical Computer Science* 358: 23–33.

Pour-El, M. B. (1974). "Abstract Computability and Its Relation to the General Purpose Analog Computer (Some Connections Between Logic, Differential Equations and Analog Computers)." *Transactions of the American Mathematical Society* 199: 1–28.

Pour-El, M. B. (1999). "The Structure of Computability in Analysis and Physical Theory: An Extension of Church's Thesis." In E.R. Griffor (Ed.), *Handbook of Computability Theory* (449–71). New York: Elsevier.

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

Putnam, H. (1960). "Minds and Machines." In S. Hook (Ed.), Dimensions of Mind: A Symposium (138–64). New York: Collier.

Putnam, H. (1964). "Robots: Machines or Artificially Created Life?" *Journal of Philosophy* 61: 668–91. Reprinted in Putnam 1975, 386–407.

Putnam, H. (1967a). "The Mental Life of Some Machines." In H. Castañeda, *Intentionality, Minds, and Perception* (p. 177–200). Detroit: Wayne State University Press.

Putnam, H. (1967b). "Psychological Predicates." In *Art, Philosophy, and Religion*. Pittsburgh, PA: University of Pittsburgh Press. Reprinted as "The Nature of Mental States" in W. Lycan (Ed.) (1999). *Mind and Cognition: An Anthology*, Second Edition (27–34). Malden: Blackwell.

Putnam, H. (1975a). *Mind, Language and Reality: Philosophical Papers, Volume 2*. Cambridge: Cambridge University Press.

Putnam, H. (1975b). "Philosophy and our Mental Life." In H. Putnam, *Mind, Language and Reality: Philosophical Papers*, *Volume 2* (291–303). Cambridge: Cambridge University Press.

Putnam, H. (1975c). "The Meaning of 'Meaning'." In K. Gunderson (Ed.), *Language, Mind and Knowledge*. Minneapolis: University of Minnesota Press. Reprinted in Putnam (1975a), 215–71.

Putnam, H. (1988). *Representation and Reality*. Cambridge, MA: MIT Press.

Pylyshyn, Z. W. (1984). *Computation and Cognition*. Cambridge, MA: MIT Press.

Quine, W. V. O. (1960). *Word and Object*. Cambridge, MA: MIT Press.

Quine, W. V. O. (1976). "Whither Physical Objects?" In R.S. Cohen, P.K. Feyerabend, and M. W. Wartofsky (Eds.), *Essays in Memory of Imre Lakatos* (497–504). Dordrecht: Reidel.

Rashevsky, N. (1940). *Advances and Applications of Mathematical Biology*. Chicago: University of Chicago Press.

Rashevsky, N. (1938). *Mathematical Biophysics: Physicomathematical Foundations of Biology*. Chicago: University of Chicago Press.

Rescorla, M. (2007). "Church's Thesis and the Conceptual Analysis of Computability." *Notre Dame Journal of Formal Logic* 2: 253–80.

Rescorla, M. (2012). "Are Computational Transitions Sensitive to Semantics?" *Australasian Journal of Philosophy* 90(4): 703–21.

Rescorla, M. (2013). "Against Structuralist Theories of Computational Implementation." *The British Journal for the Philosophy of Science* 64: 681–707.

Rescorla, M. (2014a). "The Causal Relevance of Content to Computation." *Philosophy and Phenomenological Research* 88(1): 173–208.

Rescorla, M. (2014b). "A Theory of Computational Implementation." *Synthese* 191: 1277–307.

Rescorla, M. (forthcoming). "The Representational Foundations of Computation."

Rey, G. (1997). *Contemporary Philosophy of Mind: A Contentiously Classic Approach*. Cambridge, MA: Blackwell.

Roeder, K. D. (1998). *Nerve Cells and Insect Behavior*, Revised Edition. Cambridge, MA: Harvard University Press.

Rohrlich, F. (1990). "Computer Simulation in the Physical Sciences." *PSA 1990, Vol. 2*: 507–18.

Rojas, R. (1998). "How to Make Zuse's Z3 a Universal Computer." *IEEE Annals of the History of Computing* 20(3): 51–4.

Rojas, R. and U. Hashagen (Eds.), (2000). *The First Computers-History and Architectures*. Cambridge, MA: MIT Press.

Rosen, G. (2012). "Abstract Objects", *The Stanford Encyclopedia of Philosophy* (Spring 2012 Edition), E. N. Zalta (Ed.), URL = <http://plato.stanford.edu/archives/spr2012/entries/abstract-objects/>. Stanford, CA: The Metaphysics Research Lab, Centre for the Study of Language and Information, Stanford University.

Rosenblatt, F. (1958). "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain." *Psychological Review* 65: 386–408. Reprinted in Anderson and Rosenfeld 1988: 92–114.

Rosenblatt, F. (1962). *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Washington, D. C.: Spartan.

Rosenblueth, A., N. Wiener and J. Bigelow (1943). "Behavior, Purpose and Teleology." *Philosophy of Science* 10: 18–24.

Roth, M. (2005). "Program Execution in Connectionist Networks." *Mind and Language* 20(4): 448–67.

Rubel, L. A. (1985). "The Brain as an Analog Computer." *Journal of Theoretical Neurobiology* 4: 73–81.

Rubel, L. A. (1989). "Digital Simulation of Analog Computation and Church's Thesis." *Journal of Symbolic Logic* 54(3): 1011–17.

Rubel, L. A. (1993). "The Extended Analog Computer." *Advances in Applied Mathematics* 14 (1): 39–50.

Rubel, L. A. and M. F. Singer (1985). "A Differentially Algebraic Elimination Theorem with Application to Analog Computability in the Calculus of Variations." *Proceedings of the American Mathematical Society* 94(4): 653–8.

Rumelhart, D. E. and J. M. McClelland (1986). *Parallel Distributed Processing*. Cambridge: MA: MIT Press.

Rupert, R. D. (2008). "Causal Theories of Mental Content." *Philosophy Compass* 3(2): 353–80.

Ryder, D. (2004). "SINBAD Neurosemantics: A Theory of Mental Representation." *Mind and Language* 19(2): 211–40.

Scarantino, A. and G. Piccinini (2010). "Information without Truth." *Metaphilosophy* 43(3): 313–30.

Scheutz, M. (1999). "When Physical Systems Realize Functions…" *Minds and Machines* 9: 161–96.

Scheutz, M. (2001). "Causal versus Computational Complexity." *Minds and Machines* 11: 534–66.

Scheutz, M. (2012). "What It Is to Implement a Computation: A Critical Analysis of Chalmers' Notion of Implementation." *Journal of Cognitive Science* 13(1): 75–106.

Schlosser, G. (1998). "Self-re-Production and Functionality: A Systems-Theoretical Approach to Teleological Explanation." *Synthese* 116(3): 303–54.

Schonbein, W. (2005). "Cognition and the Power of Continuous Dynamical Systems." *Minds and Machines* 15(1): 57–71.

Schonbein, W. (2014). "Varieties of Analog and Digital Representation." *Minds and Machines*. 24(4): 415–38.

Schroeder, T. (2004). "Functions from Regulation." *The Monist* 87(1): 115–35.

Schwartz, P. (2002). "The Continuing Usefulness Account of Proper Functions." In A. Ariew, R. Cummins, and M. Perlman (Eds.), *Functions: New Essays in the Philosophy of Psychology and Biology* (244–60). Oxford: Oxford University Press.

Schwartz, J. T. (1988). "The New Connectionism: Developing Relationships Between Neuroscience and Artificial Intelligence." *Daedalus* 117(1): 123–41.

Searle, J. R. (1980). "Minds, Brains, and Programs." *The Behavioral and Brain Sciences* 3: 417–57.

Searle, J. R. (1992). *The Rediscovery of the Mind*. Cambridge, MA: MIT Press.

Segal, G. (1991). "Defence of a Reasonable Individualism." *Mind* 100: 485–93.

Sellars, W. (1954). "Some Reflections on Language Games." *Philosophy of Science* 21: 204–28. Reprinted in Sellars, 1963.

Sellars, W. (1956). "Empiricism and the Philosophy of Mind." In H. Feigl and M. Scriven (Eds.), *Minnesota Studies in the Philosophy of Science, Vol. I, The Foundations of Science and the Concepts of Psychology and Psychoanalysis*. Minneapolis: University of Minnesota Press. Reprinted in Sellars 1963.

Sellars, W. (1961). "The Language of Theories." In H. Feigl and G. Maxwell (Eds.), *Current Issues in the Philosophy of Science*. New York: Holt, Rinehart, and Winston. Reprinted in Sellars, 1963.

Sellars, W. (1967). *Science and Metaphysics: Variations on Kantian Themes*. London: Routledge and Kegan Paul.

Sellars, W. (1974). "Meaning as Functional Classification." *Synthese* 27: 417–37.

Seyfarth, R. M. and D. L. Cheney (1992). "Meaning and Mind in Monkeys." *Scientific American* 267: 122–9.

Shagrir, O. (1997). "Two Dogmas of Computationalism." *Minds and Machines* 7(3): 321–44.

Shagrir, O. (1999). "What is Computer Science About?" *The Monist* 82(1): 131–49.

Shagrir, O. (2001). "Content, Computation and Externalism." *Mind* 110(438): 369–400.

Shagrir, O. (2002). "Effective Computation by Humans and Machines." *Minds and Machines* 12: 221–40.

Shagrir, O. (2006a). "Gödel on Turing on Computability." In A. Olszewski, J. Wolenski, and R. Janusz (Eds.), *Church's Thesis after 70 Years* (pp. 393–419). Heusenstamm: Ontos.

Shagrir, O. (2006b). "Why We View the Brain as a Computer." *Synthese* 153(3): 393–416.

Shagrir, O. (2010a). "Brains as Analog-Model Computers." *Studies in the History and Philosophy of Science* 41(3): 271–9.

Shagrir, O. (2010b). "Computation, San Diego Style." *Philosophy of Science* 77: 862–74.

Shagrir, O. and I. Pitowsky (2003). "Physical Hypercomputation and the Church-Turing Thesis." *Minds and Machines* 13(1): 87–101.

Shannon, C. E. (1941). "Mathematical Theory of the Differential Analyzer." *Journal of Mathematics and Physics* 20(4): 337–54.

Shannon, C. E. (1948). "A Mathematical Theory of Communication." *AT&T Technical Journal* 27: 379–423, 623–56.

Shannon, C. E. and W. Weaver (1949). *The Mathematical Theory of Communication*. Urbana: University of Illinois Press.

Shapiro, S. (1981). "Understanding Church's Thesis." *Journal of Philosophical Logic* 10: 353–65.

Shapiro, S. (1983). "Remarks on the Development of Computability." *History and Philosophy of Logic* 4: 203–20.

Shapiro, S. (1993). "Understanding Church's Thesis, Again." *Acta Analytica* 11: 59–77.

Shapiro, S. (2013). "The Open Texture of Computability." In Copeland, Posy, and Shagrir 2013, 153–81.

Shapiro, L. A. (1994). "Behavior, ISO Functionalism, and Psychology." *Studies in the History and Philosophy of Science* 25(2): 191–209.

Shapiro, L. A. (1997). "A Clearer Vision." *Philosophy of Science* 64(1): 131–53.

Shapiro, L. A. (2004). *The Mind Incarnate*. Cambridge, MA: MIT Press.

Shor, P. W. (1994). "Algorithms for Quantum Computation: Discrete Logarithms and Factoring." *Proceedings of the 37th Annual IEEE Symposium on Foundations of Computer Science* (124–34). Los Alamitos, CA: IEEE Computer Society Press.

Sieg, W. (1994). "Mechanical Procedures and Mathematical Experience." In A. George (Ed.), Mathematics and Mind (71–117). New York: Oxford University Press.

Sieg, W. (1997). "Step by Recursive Step: Church's Analysis of Effective Calculability." *Bulletin of Symbolic Logic* 3: 154–80.

Sieg, W. (2002). "Calculations by Man and Machine: Conceptual Analysis." In W. Sieg, R. Sommer, and C. Talcott (Eds.), *Reflections on the Foundations of Mathematics (Essays in Honor of Solomon Feferman)* (390–409). Urbana, IL: Henson.

Sieg, W. (2005). "Only Two Letters: The Correspondence between Herbrand and Gödel." *Bulletin of Symbolic Logic* 11: 172–84.

Sieg, W. (2006a). "Gödel on Computability." *Philosophia Mathematica* 14: 189–207.

Sieg, W. (2006b). "On Computability." In A. Irvine (Ed.), *Handbook of the Philosophy of Science: Philosophy of Mathematics* (535–630). Amsterdam: Elsevier.

Sieg, W. (2008). "Church Without Dogma: Axioms for Computability." In S. B. Cooper, B. Löwe, and A. Sorbi (Eds.), *New Computational Paradigms: Changing Conceptions of What is Computable* (139–52). New York: Springer.

Sieg, W. and Byrnes, J. (1996). "K-graph Machines: Generalizing Turing's Machines and Arguments." In P. Hájek (Ed.), *Gödel '96* (98–119). Berlin: Springer Verlag.

Siegelmann, H. T. (1999). *Neural Networks and Analog Computation: Beyond the Turing Limit*. Boston, MA: Birkhäuser.

Siegelmann, H. T. (2003). "Neural and Super-Turing Computing." *Minds and Machines* 13(1): 103–14.

Šíma, J. and P. Orponen (2003). "General-purpose Computation with Neural Networks: A Survey of Complexity Theoretic Results." *Neural Computation* 15: 2727–78.

Siu, K.-Y., V. Roychowdhury, and T. Kailath (1995). *Discrete Neural Computation: a Theoretical Foundation*. Englewood Cliffs, NJ: Prentice Hall.

Sloman, A. (2002). "The Irrelevance of Turing Machines to Artificial Intelligence." In M. Scheutz (Ed.), *Computationalism: New Directions* (87–127). Cambridge, MA: MIT Press.

Smith, B. C. (1996). *On the Origin of Objects*. Cambridge, MA: MIT Press.

Smith, B. C. (2002). "The Foundations of Computing." In M. Scheutz (Ed.), *Computationalism: New Directions* (23–58). Cambridge, MA: MIT Press.

Smith, J. M. (2000). "The Concept of Information in Biology." *Philosophy of Science* 67(2): 177–94.

Smith, P. (2007). *Gödel's Proofs*. Cambridge: Cambridge University Press.

Smith, W. D. (2006a). "Church's Thesis Meets the N-body Problem." *Applied Mathematics and Computation* 178(1): 154–83.

Smith, W. D. (2006b). "Three Counterexamples Refuting Kieu's Plan for Quantum Adiabatic Hypercomputation; and Some Uncomputable Quantum Mechanical Tasks." *Applied Mathematics and Computation* 178(1): 184–93.

Smolensky, P. (1988). "On the Proper Treatment of Connectionism." *Behavioral and Brain Sciences* 11(1): 1–23.

Smolensky, P. and G. Legendre (2006). *The Harmonic Mind: From Neural Computation to Optimality-Theoretic Grammar. Vol. 1: Cognitive Architecture; Vol. 2: Linguistic and Philosophical Implications*. Cambridge, MA: MIT Press.

Soare, R. (1999). "The History and Concept of Computability." In E. R. Griffor (Ed.), *Handbook of Computability Theory* (3–36). New York: Elsevier.

Sommerhoff, G. (1950). *Analytical Biology*. Oxford: Oxford University Press.

Sprevak, M. (2010). "Computation, Individuation, and the Received View of Implementation." *Studies in the History and Philosophy of Science* 41: 260–70.

Sprevak, M. (2012). "Three Challenges to Chalmers on Computational Implementation." *Journal of Cognitive Science* 13(2): 107–43.

Stich, S. (1983). *From Folk Psychology to Cognitive Science*. Cambridge, MA: MIT Press.

Struhsaker, T. T. (1967). "Auditory Communication among Vervet Monkeys (Cercopithecus Aethiops)." In S.A. Altmann (Ed.), *Social Communication among Primates* (281–324). Chicago: University of Chicago Press.

Swoyer, C. (2008). "Abstract Entities." In T. Sider, J. Hawthorne, and D. W. Zimmerman (Eds.), *Contemporary Debates in Metaphysics* (1–31). Malden, MA: Wiley-Blackwell.

Toffoli, T. (1982). "Physics and Computation." *International Journal of Theoretical Physics* 21 (3–4): 165–75.

Toffoli, T. (1984). "Cellular Automata as an Alternative to (rather than an Approximation of) Differential Equations in Modeling Physics." *Physica D* 10: 117–27.

Turing, A. M. (1936–7). "On computable numbers, with an application to the Entscheidungsproblem." Reprinted in Davis 2004b, 116–54.

Turing, A. M. (1939). "Systems of Logic Based on Ordinals." *Proceedings of the London Mathematical Society, Ser. 2* 45: 161–228. Reprinted in Davis 2004b, 155–222.

Turing, A. M. (1945). "Notes on Memory." Reprinted in B. J. Copeland (Ed.), *Alan Turing's Automatic Computing Engine* (456–7). Oxford: Oxford University Press.

Turing, A. M. (1950). "Computing Machinery and Intelligence." *Mind* 59: 433–60.

van der Spiegel, J., J. F. Tau, et al. (2000). "The ENIAC: History, Operation and Reconstruction in VSLI." In R. Rojas and U. Hashagen (Eds.), *The First Computers: History and Architectures* (121–78). Cambridge, MA: MIT Press.

van der Velde, F. (1993). "Is the Brain an Effective Turing Machine of a Finite-state Machine?" *Psychological Research* 55: 71–9.

van Heijenoort, J., (Ed.) (1967). *From Frege to Gödel*. Cambridge, MA: Harvard University Press.

Vichniac, G. Y. (1984). "Simulating Physics with Cellular Automata." *Physica D* 10: 96–116.

von Neumann, J. (1945). *First Draft of a Report on the EDVAC*. Philadelphia, PA: Moore School of Electrical Engineering, University of Pennsylvania.

von Neumann, J. (1951). "The General and Logical Theory of Automata." In L. A. Jeffress (Ed.), *Cerebral Mechanisms in Behavior* (1–41). New York: Wiley.

von Neumann, J. (1958). *The Computer and the Brain*. New Haven: Yale University Press.

Wang, H. (1974). *From Mathematics to Philosophy*. New York: Humanities Press.

Weisberg, M. (2013). *Simulation and Similarity: Using Models to Understand the World*. Oxford: Oxford University Press.

Welch, P. D. (2008). "The Extent of Computation in Malament–Hogarth Spacetimes." *British Journal for the Philosophy of Science* 59: 659–74.

Werbos, P. (1974). *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. Ph.D. Dissertation. Cambridge, MA: Harvard University.

Wheeler, J. A. (1982). "The Computer and the Universe." *International Journal of Theoretical Physics* 21(6–7): 557–72.

Widrow, B. and M. E. Hoff (1960). "Adaptive Switching Circuits." *1960 IRE WESCON Convention Record* (96–104). New York: IRE. Reprinted in Anderson and Rosenfeld 1988, 126–34.

Wilkins, B. R. (1970). *Analogue and Iterative Methods in Computation, Simulation, and Control*. London: Chapman and Hall.

Wilson, R. A. (1994). "Wide Computationalism." *Mind* 103: 351–72.

Wilson, R. A. (2004). *Boundaries of the Mind: The Individual in the Fragile Sciences*. Cambridge: Cambridge University Press.

Wimsatt, W. C. (2002). "Functional Organization, Analogy, and Inference." In A. Ariew, R. Cummins, and M. Perlman (Eds.), *Functions: New Essays in the Philosophy of Psychology and Biology* (173–221). Oxford: Oxford University Press.

Winograd, S. and J. D. Cowan (1963). *Reliable Computation in the Presence of Noise*. Cambridge, MA: MIT Press.

Winsberg, E. (2010). *Science in the Age of Computer Simulation*. Chicago: University of Chicago Press.

Wolfram, S. (1985). "Undecidability and Intractability in Theoretical Physics." *Physical Review Letters* 54: 735–8.

Wolfram, S. (2002). *A New Kind of Science*. Champaign, IL: Wolfram Media.

Wouters A. (2007). "Design Explanation: Determining the Constraints on What Can Be Alive." *Erkenntnis* 67(1): 65–80.

Zuse, K. (1970). *Calculating Space*. Cambridge, MA: MIT Press.

Zuse, K. (1982). "The Computing Universe." *International Journal of Theoretical Physics* 21 (6–7): 589–600.

Index