Implementation and Interpretation: A Unified Account of Physical Computation

By

DANIELLE J. WILLIAMS
DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Philosophy

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

DocuSigned by:
Zoe Drayson
90CB5937D3A1477...

Zoe Drayson, Chair

DocuSigned by:
Rohan French
621C8A5FBDD14A4...

Rohan French

DocuSigned by:
Corey Maley
47B24262C6AC4CD...

Corey J. Maley

DocuSigned by:
William Ramsey
B2A3F04093D94BE...

William Ramsey

Committee in Charge

2023

For James

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Zoe Drayson, whom I am deeply grateful for. This project, in its current form, would not have been possible without you. Over the years you have been a sounding board for ideas, offered valuable insights into how to understand the motivations driving the literature, how to understand difficult topics, and so much more. Without you, this project would not be what it is. Your profound belief in my work and in my abilities as a philosopher has been a constant source of strength throughout this project and my entire graduate student career. Without you, I, undoubtedly, would not be the philosopher that I am today.

I would also like to thank my committee members, Elaine Landry, Rohan French, Corey Maley, William Ramsey, and Hanti Lin. It was in Elaine's mathematical structuralism seminar that I first discovered computational structuralism. Her willingness to let me explore the topic as a final seminar paper would prove to be the catalyst for this entire project; an opportunity that I am extremely grateful for. I am forever indebted to Rohan French for his extensive feedback on many, many iterations of various papers that would come to be major parts of this project. Much of what I do in this project comes from long discussions in our reading group and chats along the way. As someone whom I engage with in many places in this project, I am incredibly grateful to have come to know Corey Maley. Our first interaction involved several very long emails back and forth discussing some of his work and the mechanistic account of physical computation. At that time, I was thinking mostly about the mechanistic account and Corey's insights and discussion proved to be invaluable during that time. I would also like to thank William Ramsey for both his indirect and direct contributions to this project. I am very inspired by his work, and I am extremely grateful for his insightful comments on this project. Finally, I am indebted to Hanti Lin for taking the time to think about physical computation with me and for offering valuable comments about various topics in this project.

I am incredibly thankful to many people who have been willing to discuss this project with me over the years including Oron Shagrir, Francis Egan, Colin Klein, David Chalmers, and members of the MENTaL reading group at UC Davis. But especially to Gualtiero Piccinini whose support has been instrumental in the development of this project. I am also thankful to Felipe De Brigard and Walter Sinnott-Armstrong for hosting me at SSNAP (Summer Seminars in Neuroscience and Philosophy) at Duke University where I was able to critically engage with theories of implementation and their relevance to neuroscience.

Most importantly, I would like to thank James for his unwavering support through this process. I know that I have been successful, in part, because I have had you by my side. Thank you for always believing in me, supporting me in every way I need, and always looking forward to the future with me even when we're unsure of what it might look like. I am grateful for my mom who taught me to be independent and to work hard. She taught me that if I could manage that, then I could do anything. I am also thankful for Lorraine and Len. You've been a constant source of support, cheering me along and celebrating every milestone. Finally, thank you to my animal companions that have taken care of me at every stage. To Kittie, you'll always be here with me—we finally made it! To Jake for keeping me company no matter how late it gets. And finally, to KC for laying on my keyboard at least once a day.

ABSTRACT

My dissertation provides a framework for understanding how various theories of physical computation, which are often taken to be alternatives, can be made compatible when we specify different questions targeted by the different accounts across the debate. In this project, I identify three questions: implementation (the relation between a computational structure and a physical system), interpretation (what computation the system performs), and individuation (what distinguishes computing systems from non-computing systems). I argue that the debate is often framed in terms of the individuation question, but that most accounts address either implementation or interpretation. A consequence of framing the debate in terms of the individuation question is that views that reference semantic content and views that do not end up looking like alternative ways to address physical computation. But I argue, semantic and non-semantic views are not alternative accounts of physical computation. Instead, they answer different questions that each address a different aspect of physical computation. I clearly distinguish between two questions: implementation and interpretation, and I argue that these questions are being answered by non-semantic and semantic accounts, respectively. Once I define these questions, I offer a framework that demonstrates how the different views are complimentary. In addition, I present a path forward for how to continue engaging with questions about physical computation with clearly articulated targets that can be addressed independently or cooperatively.

TABLE OF CONTENTS

LIST OF FIGURES

INTRODUCTION

Several years ago, I read Rescorla's 2012 paper arguing for a semantic theory of computation. Part of his project argued against what he called "structuralist" theories—theories that are often referred to as "mapping views" or "non-semantic accounts." This paper struck me as doing something very strange—strange in ways that I couldn't yet pin down. For starters, why would someone give a theory about how to semantically interpret computing systems as a response to a view that said that we could identify computational systems with a mapping from formal states to physical states? The next paper I read was Sprevak's 2010 semantic view. I remember getting the distinct feeling that there was some kind of *switch* happening. I found this switch in the additional papers that I read as well. The authors identify a problem (what I now call "trivial implementations"), make the claim that it's a problem for the non-semantic account (that addresses what I call "the implementation question"), and then give a solution in terms of *semantically interpreted* states or processes. As I read more and more papers, I saw this pattern over and over again. I could not shake the thought that we only *interpret* systems that we *already know* are computers. This idea is at the heart of my dissertation. I define two main questions that are present in the literature: one that asks how to understand the relation between a formal structure and a physical system (the implementation question) and one that asks how we should understand what that physical system computes (the interpretation question). I argue that once we acknowledge that these are different, equally important questions, it becomes apparent that non-semantic accounts and semantic accounts are compatible, rather than alternative ways to address physical computation.

The idea that implementation and interpretation are compatible unifies the literature on physical computation. I've presented my project at several conferences and discussed it with many of the people whose views are present in this project. What became clear to me over the years is that there is a third question that looms in the background of many of the more recent views—one that is not always made explicit in the literature, but one that many people reference when they question whether a project like mine can work. I identify this question as the "individuation question." This question asks what the difference is between systems that compute and systems that don't. *This* question is what drives the claim that computation is necessarily a semantic process. Some think that the *very nature* of physical computation is that computation is a *semantic* process, and thus they go on to give a "semantic account." Making sense of how this question was driving the literature generated a turning point for my project. If *this* is the question some people take themselves to be answering, then no wonder why it seems like semantic and non-semantic theories are in opposition. I acknowledge the importance of this question in the debate, and I develop a place for it within my framework.

In chapter one, I lay some groundwork for later chapters. I introduce some early computing systems up to and through the development of computability theory. This is relevant for a theory of physical computation for different reasons. Some theories of physical computation might be interested in characterizing digital computers. But you might wonder if we need a more neutral theory; one that captures the nature of all types of computation. Moreover, early mapping views were characterized in terms of correspondence between transitions on a Turing machine table and physical states. Thus, understanding Turing's contribution to computation plays an important role in understanding what many accounts are doing. Additionally, if you think that computation necessarily involves information processing, we might wonder what information is in the first place. If we want the theory to capture mental computations, how does that change our notion of information? Are there computing systems that do not process information? These questions, and many others, will play a role in a theory of physical computation—especially if our goal is to capture the *nature* of physical computation that we can apply not just to computing artifacts but to natural computing systems, too.

In chapter two, I draw the relations between functionalism, computationalism, and physical computation. Functionalism and computationalism are distinct but related theses about the mind, and both play an important role in accounts that address physical computation. Functionalism is the view that what makes something a mental state of a particular type depends on how it functions or its role in the system of which it is a part, rather than what the system is made of. Computationalism is the view that the functional organization of the brain is computational. These two theses are sometimes brought together to form a view called computational functionalism. A feature both views share is that they rely on abstract descriptions of physical phenomena, and in some cases, the descriptions are computations. This means that the relationship between formal computation and physical systems will matter for functionalism, computationalism, and a theory of physical computation. This is no surprise given the triviality arguments that plagued early mapping theories of implementation stem from arguments against functionalism and sometimes computationalism.

In chapter three, I introduce my framework that distinguishes three questions in the computation literature: implementation, interpretation, and individuation. I argue that non-semantic accounts answer the implementation question while semantic accounts answer the interpretation question. I derive the implementation question from Chalmers (1995). This question asks "What are the conditions under which a physical system implements a given computation?" But the formulation of the implementation question is ambiguous between two readings. This ambiguity generates two ways of understanding the target of the question. "A computation" is sometimes understood in terms of 1) a computational structure or object or 2) a specific computational process. I argue that, in some cases, how one interprets the question will lead to either a theory of implementation or a theory of interpretation. I maintain that the first option stays true to Chalmers's formulation and should be adopted as the proper formulation of the implementation question. But the second formulation generates an additional question that needs answering. I call this the "interpretation question." The interpretation question is what semantic theories answer. In chapter four, I argue that these two formulations also track two types of triviality.

The locus of the debate is often framed in terms of non-semantic versus semantic views, but I argue that framing the debate this way is a mistake. But this is because sometimes the debate is framed in terms of the *individuation question*. I derive the individuation question explicitly from Piccinini (2015) and Shagrir (2022). They both frame the debate this way, including, sometimes, applying it retroactively to older views. I argue that this way of framing the debate supports the idea that semantic and non-semantic views (and even a mechanistic account) are alternative ways to give a theory of physical computation. Even so, we can still accept the individuation question as part of the theory. I will argue that one can maintain their *metaphysical commitments* regarding the nature of physical computation while adopting my framework that distinguishes between implementation and interpretation.

In chapter four, I argue that there are two types of triviality that are central to theories of physical computation. I identify these two types as 1) a trivialization of the implementation conditions and 2) a trivialization of the interpretation conditions. These two types of triviality can emerge independently from distinct arguments or arise from the same argument. I also show in this chapter that some writers are concerned with one type while giving reasons to dismiss or accommodate the other. This is fine within my framework because regardless of whether someone cares about one type or the other, they both need to be addressed. Toward this end, I argue that these two types of triviality should be articulated as separate problems that require their own independent answers.

In chapters five and six I argue that non-semantic accounts address the implementation question, while semantic accounts address the interpretation question. I also argue that implementation views are best equipped to handle the problem of trivial implementations while interpretation accounts are best equipped to handle the problem of trivial interpretations. In chapter five we see that non-semantic accounts that answer the implementation question are easier to handle because they claim to be addressing implementation as Chalmers formulated it. However, things get tricky with semantic views because the terminology often reflects the nature of the implementation question as I define it. So, much of the work I do in chapter six is revisionary because some people aren't answering the question that they think they are. But importantly, I argue that semantic theorists can keep their answer to the individuation question—they can maintain their commitment to the idea that computation is necessarily a semantic process.

In chapter seven I introduce an additional category of views: the mechanistic accounts. In this chapter, I distinguish between "weak" and "strong" mechanistic accounts. I also make a more subtle point (which can also be found in chapter 6 regarding one of the semantic views), that sometimes theories that are credited as a bonafide theory of physical computation, aren't actually giving a theory of physical computation—at all! But this is no surprise. If the questions aren't clearly defined, then any answer can look like a solution from the right angle. But, most of this chapter is concerned with Piccinini's 2015 mechanistic account. The mechanistic account is "strong" in that it's wholly mechanistic. Interestingly, it turns out that Piccinini's view is the *only* account that I think answers the individuation question by giving a full-blown theory rather than forking off into a theory of implementation or interpretation.

In chapter eight, I articulate several forms that my framework might take, and I settle on one that allows for the inclusion of a supporting mechanistic* account. The asterisk signals a Piccinini-*style* view because it abandons his conceptual framework. In this chapter, I build my own unified theory of physical computation showing how it's possible to incorporate one's own views on physical computation in a unified account that answers both the implementation and interpretation questions.

Chapter One: Some background on computation and computing systems

1 Introduction

Physical computers are ubiquitous in contemporary society. But, before we all carried little computers around in our pockets, computation had its defining characteristics independent of its physical counterpart. Computation during this time was mainly understood as a mathematical, formal process. It is the abstract, formal nature of computation that made it an attractive candidate for functional explanations of mental processes in philosophy of mind and later cognitive science. Before physical computers existed (less some mechanical systems that I will discuss later), mathematicians—the early computer scientists—were laying its foundations. Turing's Turing machine, an abstract theoretical device, gave us one of the first links to a mechanical computing device—a way to think about how a physical, mechanical system could perform computational manipulations. This allowed for the possibility that we could build machines to perform the computations described abstractly. The idea that there are systems in the world that might perform computation raises interesting metaphysical questions about the connection between formal mathematical computation and the physical machines—or even biological systems—that instantiate the computations. When it comes to our computing artifacts, we have a very good understanding of this relationship—after all, those are systems *we* made! But what about systems we did not construct; how should we understand the relationship between formal computation and, say, the brain? Before I get into these questions, I will guide us on a tour through some history of computation and computing devices.

I begin this chapter by giving some examples of early analog-style computing devices, including some that were built and used well before Turing and others were working on mathematical computation. This includes later devices that were electronically powered. In the next section, I discuss digital computation. Digital computation is introduced in the context of Alan Turing's work on the Turing machine and, later, physical machines built to implement Turing computations. I will introduce and explain some key computer science and information theory terminology in section three. This exercise aims to lay some groundwork for future chapters that rely on many of these concepts. When they come up in later chapters, I will do some additional explanatory work and cite this chapter when I think it might be helpful. This will allow me to sometimes speak freely about certain concepts. In the final section, I will consider some philosophical issues regarding the concepts I have discussed. For example, what is the difference between analog and digital computing systems; when it comes to computation and information processing, what does 'information' amount to; how do the concepts of 'semantics' and 'representations' influence how we think of semantics in a computational system, among others.

2 Computing machines

2.1 Analog computers

To start, we might think that analog computers store data as *continuous* physical quantities and perform computations by manipulating measures that represent numbers. This is because analog computers are oftentimes colloquially distinguished from digital computers because of their perceived continuous nature. However, analog computing was not always associated with continuous quantities. Initially, 'analog' derived from the concept of 'analogy,' where analog computers were so-called because they were used to build models that created a mapping between two physical phenomena (Care, 2006). A roadmap is a good example of an analogous system. You can trace the path on the map and calculate the distance to be traveled, but you do not *literally* travel on the map. That map serves as an *analog* for the real world.

The earliest known analog computer is the 'The Antikytherma mechanism' (dated as early as 200 B.C.E). This machine can be used to predict astronomical positions and eclipses. The inside of the device features sixty-nine gears mounted in various ways to represent the position of the sun, planets, and various moons and their features. The gears work together on different calculations for multiple purposes. From the outside of the machine, a user can turn a calendar dial to a desired point in the past, present, or future, and the gears inside move, causing pointers and rings on the surface of the machine to be displaced revealing the desired celestial positions (Freeth, 2022). The Antikytherma mechanism provides a physical *analogy* between the positions indicated on the machine and the actual celestial bodies in space.

Analog computers did not always bear this name, though. It was not until the 1800s that the concept of an 'analog computer' was developed. This concept is often retroactively applied to machines from the past, but it was not until the 19th century that the term itself came into use. Nonetheless, even the properly called analog computers work by generating an analog between some physical phenomena and the machine itself—a representation whose structure corresponds to that of which it represents. In 1814 Johan Hermann invented a wedge device to compute the area within an irregular boundary contour. He called this invention the "planimeter." Later, inspired by planimeter technology, James Thomson invented the wheel and disc integrator. The wheel and disk integrator uses the movements of a roller-driven sphere against a circular flat surface to perform the mathematical procedure known as “integration.” Integration is the summing together of a series of cognate elements resulting in a mathematically meaningful whole. The device achieves this by analoguing the magnitude of each element onto the radial distance of the ball from the center of the disk and letting the number of turns of the sphere accumulate the integral. Thomson’s disc integrator went on to become the foundation of analog computing machinery (Thomson, 1876).

William Thomson’s Tide Predictor was a popular analog computing device that traced the tidal curve for a given location by combining ten astronomical components using the disc integrator. The Tide Predictor is a hand-cranked tide predicting machine made of gears and pulleys, each analoguing one step towards the final solution. The tide predictor remained in use until the 1960’s (Gregersen, 2022). Some analog computing devices came about, in part, due to the engineering demands of the military during wartime. For example, Prince Louis designed the Battenberg Mark Course indicator. This analog computing device was used for ready reckoning the course to be steered in order to close in on a chosen vessel (given your speed) (Schliehauf, 2004). The device consists of two position bars for setting the initial and final stations, a speed bar (where the speed of the ship relative to the flagship was set), clamped at one end into the diameter grove by the speed ration clamp, and a guide bar, together with a circular disk. To use the instrument, the flagship course is first set on the guide bar. The initial and final stations are set using the position bars, and the speed ration clamp is set to show the ratio of one’s ship speed to the speed of the flagship. The course can be read off the circular disk if the pin beneath the speed ration clamp is set within the diameter grove.

Around the time that digital computers entered the scene, analog computing underwent a conceptual shift. Early devices used to solve analytical problems were defined and described using *continuous* real numbers. The shift from analogicity to having a "continuous nature" was a terminological shift brought on by engineering practices. During this time, an effort was made to develop systems with a minimal set of processing components (such as amplifiers and integrators) that ended up being well-suited for processing continuous signals (Stacewicz, 2020). To this day, analog computers are generally thought of as processing continuous signals.

In 1931, Vannevar Bush at MIT built the differential analyzer, an analog computing machine that took advantage of continuous signals (Owens 1986). The differential analyzer was the first large-scale automatic general-purpose mechanical analog computer. Like many other analog computing devices, this machine was initially based on the wheel and disc integrator requiring a skilled mechanic to set up the machine for each new job. But eventually, the wheel and disc integrator and other mechanical components were replaced by electromechanical parts, and then finally, fully electric components. The differential analyzer can best be understood as a series of 'black boxes' (Copeland, 2020). Each box performs a process such as adding, multiplying a variable by a constant, or integrating. During setup, the boxes are connected so that the desired set of processes can be executed. Because all of the boxes work in parallel, the electronic differential analyzer can solve equations very quickly.

In 1941 Claude Shannon introduced the General-Purpose Analog Computer (GPAC). The GPAC consisted of circuits with several basic units that were interconnected in order to compute some function. The GPAC was physically built in 1949 and given the name "Gypsy." To use Gypsy, parameters and initial values would be entered into the machine (potentiometers). The master switch placed in "reset" mode would control relays that applied the initial values to the integrators. Turning the switch to "operate" or "compute" mode would set the computational process in motion. Placing the switch in "hold" mode would stop the computation and stabilize the values, allowing them to be read off (MacLennan, 2009).

The earliest in-home electronic analog computers depended on vacuum tubes for their operations. An example of a vacuum-tube, electronic, analog computer is the Heathkit EC-1 (some are still up and running today!). The Heathkit is a computer from the 1960s that required the purchaser to construct the machine out of supplied parts. It features nine electron tube-based operational amplifiers, five variable resisters, various high-tolerance resistors and capacitors for operations, a multivibrator for repeatedly resenting and re-running the computations, a meter, other panel adjustments, and an oscilloscope is used to read the output. The sweeping of the oscilloscope might represent different real-world problems; if it did so accurately, the user would take a picture of the result to save for later. However, vacuum tubes are expensive, bulky, and require a lot of maintenance. Because of this, eventually they were replaced with modern components such as resistors, capacitors, inductors, transistors, etc. The electrical signal flowing through the networks or circuits experience variations in amplitude, frequency, phase, and other related properties (Schembri & Boisseau, n.d.).

In this section, I have given some examples of early analog computing devices and demonstrated the shift from analogicity to continuous processing as the defining characteristic of analog computation. While it is reasonable, given this history, to identify analog computing with continuous processing, it is contentious to draw the distinction in terms of continuous versus discrete processing. In section three, I will discuss some philosophical considerations that come up when drawing the distinction this way.

2.2 Digital computers

In his 1936 paper, Turing introduced us to an idealized typewriter—what we now famously refer to as a "Turing Machine." He describes an "automatic machine" that is supplied with a "tape" running through it which is divided into sections called "squares." Each section is capable of bearing a "symbol" that is located "in the machine." Only when a symbol is "scanned" is the machine "aware" of it, but it is possible for the machine to hold previously scanned symbols in a "memory" store. The machine's configuration determines the machine's behavior and is always dependent on the current state that is being "scanned" by the system. The machine can also write a symbol in a blank square and move sequentially left or right along the tape to scan different squares. The symbols written by the machine are the numbers being computed, while the other numbers are present to assist the memory states of the system (Turing, 1936). An *a*-machine is a machine described above whose behavior is entirely determined by its configuration. Any machine whose behavior is only partially determined by its configuration is called a "choice machine" (*c*-machine). *C*-machines require the assistance of an external operator to assist the machine under some circumstances. In addition, Turing describes a "universal machine"—what we now call a "Universal Turing Machine"—a machine that can perform the behavior of any *a*-machine. In other words, it can perform all functions that a single *a*-machine might be specified to perform; it can have *m*-configurations.

The instructions that govern the Turing machine take the form of a machine table which is understood as the *program*. The program is an 'instruction table' that includes a finite set of instructions, each calling for a single operation to be performed if the appropriate conditions are met. Recall the setup of Turing's *a*-machine above. The machine starts with an infinitely long blank tape. We can give a machine table (program) that tells the scanner to print alternating binary digits on the tape while also leaving a blank space between each digit:

State	Scanned symbol	Print	Move	Next state
a	Blank	0	Right	b
b	Blank		Right	c
c	Blank	1	Right	d
d	Blank		Right	a

Figure 1. Turing machine table

The resulting output would be the following tape (extending on endlessly):

0		1		0		1		0	

Figure 2. Turing machine tape

The above machine table is an example of the instructions given to an *a*-machine to get it to perform a task. The tape is the resulting output. To turn an *a*-machine into a *universal* Turing machine, there is an additional requirement. A universal Turing machine requires a dedicated machine table that is present no matter which program the machine follows. Call this machine table *U*. Machine table *U* allows the machine to follow instructions *printed on the tape*. This lets the machine act *as if* it were following the instructions of any machine table. If, along with its machine table *U*, another machine table were to be translated into binary digits and then written on the tape, the machine would behave according to the translated machine table. So long as the Turing machine has *U* as its program, any other program can be translated, written on the tape, and executed.

The Turing machine was remarkably simple but convincingly demonstrated that a human computer could also compute any function that a Turing machine could compute. This work developed the mathematical concept of a "mechanical procedure" by demonstrating equivalence between human and mechanical computational processes. This insight became known as "Turing's Thesis." Turing introduced this work in the context of the *Entscheidungsproblem* (Hilbert & Ackermann, 1928). The *Entscheidungsproblem* was a decision problem in first-order predicate calculus previously thought to be unsolvable. But by showing what *was* computable, Turing also showed what was *un*computable. This led to a solution that specified that there was *no* effective method for solving the *Entscheidungsproblem*. Because a Turing machine could not tell whether or not a formula was a theorem of the calculus, Turing's Thesis tells us that no human can either. Alonzo Church (1936) also came to a similar but equivalent conclusion which is why Turing's Thesis is now known as the Church-Turing Thesis.

At the time that Turing was working on this project, the modern digital computer did not physically exist. Importantly, though, Turing was not trying to develop the foundations of a physical, digital computer. Instead, he was concerned with understanding the limits of *human* computing capacities. Turing thought that his idealized machine was the most "natural" model of human computational processes. Nonetheless, because Turing was concerned with modeling the human computer, his analysis naturally led to an analysis of computation in physical systems more generally.[1] Turing's impact on theoretical computer science is substantial. Today, the Turing machine and its accompanying theory are part of the theoretical foundations of computer science and have contributed to important questions about how to understand the nature of an algorithm, how to define computation in general, and how to understand the nature of physical computation (the topic of this entire dissertation!).

In his 1945 report, Von Neumann (1993) described the structure of a *"very high speed automatic digital computing system*"[2] Within this report, he discusses the different parts found within the machine, such as wheels that can be locked into various positions in order to transmit electrical impulses that may cause other wheels to move, single or combined telegraph relays that are actuated by an electromagnet, and so on. The original input or stimulus causes these internal pieces to move and act on each other. He specifies that high speed computing devices should ideally have vacuum tube elements because they have high reliability and very fast reaction times. The vacuum tubes can be used as a current valve (a gate) and can have an all-or-none feature where a grid bias above or below a cut-off determines whether the valve will pass current or not. In addition, there can be what von Neumann calls "trigger circuits," several states in the machine where a combination of vacuum tubes maintains a constant equilibrium indefinitely without any required

[1] Gandy (1980) proposes a physical version of the Church-Turing thesis that extends Turing's analysis to discrete mechanical devices.
[2] Von Neumann's italics

inputs. Whether the tubes function as gates or triggers, they are set up to take two-valued digits as their inputs, meaning that the system is best set up using a *binary* system. Here we can see the *discrete* nature of how information is processed in the system: in a step-by-step fashion using an all-or-none binary symbolic structure (0's and 1's).

Prior to von Neumann's theoretical work on computational architecture, computers would house the program memory and data memory in distinct locations within the system. Von Neumann was the first to unite the two into a shared memory unit. This made it possible for the instructions to be arithmetically modified in the same way the data was because the program and the data were treated as the same thing within the system. This architectural shift made a big difference in terms of computing power because computations could be performed faster while also being efficient and flexible. Because the programs were stored as data in the system, the machine did not require reprogramming for each problem set. Instead, it kept multiple programs as subroutines stored within a library that could be accessed automatically.

Charles Babbage was one of the first to work on building physical, digital computing machines. Babbage's first machine, the Difference Engine, performed complex mathematical tasks to produce accurate mathematical tables. The intended purpose was to aid with astronomy and navigation by solving a practical problem: people were bad at math, and ships were regularly getting lost! The Difference Engine was to be made entirely of mechanical, non-electric components. Although Babbage never completed a full-scale Difference Engine, he provided complex plans for how to do so. In 1990, his Difference Engine 2 was built to scale and is currently on display at the London Science Museum (Bromley, 1987).

Later, Babbage proposed the Analytical Engine, a general-purpose digital computer. Plans for the Analytical Engine included a memory store and a central processing unit. It would also include decision-making abilities (conditional branching). The Analytical Engine was promised to bring more power along with the ability to perform more complicated tasks that resembled mathematical analysis. The behavior of this machine would have been controlled by a program using instructions that took the form of punch cards connected with ribbons. These punch cards were old technology borrowed from the Jacquard programmable weaving loom. Each row of the punch card corresponded to a row in the pattern, allowing the system to work automatically. The Analytical Engine was promised to be able to perform an unlimited number of computations—a physical universal Turing machine.

The earliest digital computer to include a stored program and the first physical Turing machine was built at Manchester University. This computer is now known as the "Manchester Baby" and was constructed by F.C. Williams and Tom Kilburn. The Manchester Baby performed its first calculation on June 21st, 1948. The Manchester Machine could store 1,024 bits on a cathode-ray tube—just enough space to store and run a single program on the system. The machine was built to work as a testing system for the Williams tube, the early form of computer memory. Later, the Manchester Mark I would be built, serving as the world's first commercially available general-purpose digital computer (Centre for computing history, n.d.).

3 Key terminology

3.1 Information

The most influential information theory comes from Shannon (1948) and later from Shannon & Weaver (1963).[3] Today, we call this type of information "Shannon Information." Shannon Information specifies the amount of entropy in a message. Information entropy is a mathematical concept introduced to quantify the amount of uncertainty or randomness in a message. So, what Shannon gives us is a mathematical way to measure the amount of uncertainty or unexpectedness of a message or signal contained in a set of symbols or variables. For example, imagine receiving a message that consists of a series of symbols. The amount of information the message contains can be quantified by the amount of uncertainty or unpredictability of the symbols. If the symbols are entirely random and have no pattern, then the message contains a high amount of information because it has a high amount of uncertainty. Put differently, suppose the symbols follow a predictable pattern (one familiar to you). In that case, the message contains a low amount of information because you would not be surprised by what it says. Alternatively, if someone tells you something new that you have never heard of before, you get a lot of information from the message, and if you get a message that tells you facts you already know, you have not taken in any new information, so the amount of information you have been given is very low—if you have received any at all. This notion is paired with the idea of communication when we consider that communication involves transmitting information from one party to another.

Shannon was the first to make the above idea mathematically precise. His formula calculates the minimum number of bits (a threshold called Shannon entropy) required to communicate a message. He also showed that the message would be distorted if a sender used fewer bits than the minimum required. Shannon defines information in terms of the negative log of the probability:

[3] The same Claude Shannon that built the GPAC!

$I(A) = - \log P(A)$. He showed how encoded messages could efficiently transmit information across communication channels. His account applied to communication systems, as diagramed below:

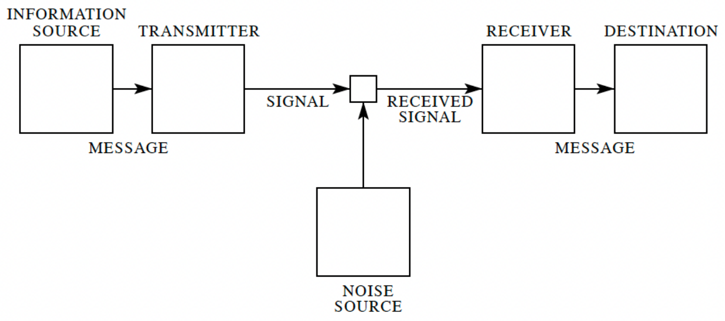

Figure 3. Schematic Diagram of a General Communication System

The communication systems that Shannon was interested in consisted of five parts. 1) an *information source* that produces a message or sequence of messages. A message source could be a telegraph, telephone, radio, television, etc. 2) a *transmitter* that operates on the message in some way to produce a signal suitable for transmission. This might include changing the sound pressure into electrical current, encoding a sequence of dots and dashes, etc. 3) A *channel*—the medium used to transmit the signal from transmitter to receiver. This could be wires, cables, frequencies, or beams of light. 4) The *receiver* generally performs the inverse operation of the transmitter: reconstructing the message from the signal. Finally, 5) the *destination*—the person (or thing) for whom the message is intended. Shannon’s goal was to specify how information was moved through systems containing these features.

Using Gibb's formula for entropy in physics, Shannon showed that the communication entropy of a system is maximal when all the messages have equal probability. The amount of information I in an individual message x is given by: $I\ (x) = -\log p_x$. This formula tells us that a message x has a probability p_x between 0 and 1 of occurring. If $p_x = 1$, then $I(x)$ will = 0 and we will receive a message that contains no information. The lower the probability of the message, the more information it contains. If two messages are unrelated the amount of information that comes through is equal to the sum of the information of the individual messages: $I(x \text{ and } y) = I(x) = I(y)$ (Adriaans, 2020).

3.2 Symbols

Shannon's goal was to define information within the context of communication. However, there is more to understanding the concept of 'information' than just having a mathematical theory like the one given above. We can ask additional questions about how to understand information in terms of its relation to computation. If symbols are (at least sometimes) understood as *carrying* information, and (at least some) computation involves symbolic structures, then a notion of what it means for a symbol to carry information is needed to understand (some) computational processes.

A symbol is an element of notation that represents an idea, concept, value, or object. When a symbol represents something, it means that the symbol has been assigned a specific meaning or interpretation, and that meaning is used to convey information or express a relationship. In mathematics, symbols are used to represent numbers, variables, and operations. The symbol 'x' might represent a variable, while the symbol '+' represents the addition operation. Here symbols are used to express mathematical expressions and equations. In computer programming symbols are used similarly to represent values, variables, and operations within a particular programming language (this is explained later in the chapter). In symbolic logic, symbols are used to represent propositions, logical operations, and relationships between propositions. For example, the symbol ‘ ^ ’ is used to represent ‘and’ while the ‘v’ symbol is used to represent ‘or.’

In addition, we also use symbolic representations in our everyday life. Consider the process of taking measurements. When we do this, we track features such as length and distance using abstract mathematical concepts such as numbers and counting. Measurement is a kind of *information processing* that uses pieces of information to track features of the world—a process that can transform otherwise meaningless features into *meaningful symbols*. For example, using a tally system to track the number of times your dog asks to go into the backyard. Each time your dog approaches the sliding door, you make a tally mark on a piece of paper. A tally mark on the piece of paper, on its own, does not have any symbolic meaning. But, in the context of tracking your dog’s trips outside, the tallies take on meaning; they *symbolize* trips outside within the context of your tracking the trips. Once we decide that marks on a paper stand in for some object or occurrence, the marks count as a symbol that *means* something. These symbols *represent* how often my dog asked to go in the backyard (which is a lot).

Different symbols can have the same meaning and be used to represent the same function. Consider the following symbolic representations of ‘3 + 2 = 5:’

$$3 + 2 = 5$$

| | | + | | = | | | | |

III + II = V

. . . + . . = ...

The above four sentences have the same truth value, but each uses a different symbolic representation. In other words, each statement is of the same *function* using different symbols. Put simply, different symbols can be used to represent the same information.

An important notion related to symbol manipulation and computation is recursion. Recursion in computer science is a programming technique where a function calls on itself to solve a problem. It is a way of solving problems by breaking them down into smaller, similar problems and then solving each in turn (Sipser, 2006). For example, we can create a recursive function that calculates the product of all the numbers from 1 to 5. First, find the base case: the simplest case that can be solved without recursion. In this example, the base case is the number 5. Then, divide the numbers by separating the base case from the remaining numbers: 1, 2, 3, 4, and then 5. Then, find the product of the first set of numbers, which is 24. Finally, multiply that number by the base case: 24 x 5. This will give you the answer: 120. This is a simple example of recursion, but it gets at the idea that we can break a complex task down into simpler tasks to solve a problem. Some examples of recursive functions are factorials (a factorial of a non-negative integer *n* is the product of all positive integers less than or equal to *n*) and Fibonacci sequences (the Fibonacci sequence is a series of numbers in which each number is the sum of the two proceeding ones). Recursive functions are abstract relations defined over the natural numbers. These functions can be computed without any reference to the physical world. But they can be *paired with* symbolic representations in the physical world. For example, when we write the steps on paper using a pencil, the pencil strokes serve as a symbolic representation of the function. The idea that we can represent recursive mathematical functions with physical symbols brings abstract mathematical structures and the physical world together.

But you might wonder, how we should understand the relationship between abstract mathematical objects and the physical world. Sometimes this relation is cashed out in terms of a mapping between mathematical objects such as numbers and functions and concrete phenomena or objects in the world. Above, I mentioned drawing tally marks on a piece of paper to keep track of every time my dog asks to go out the sliding door into the backyard. Say that he requested access to the backyard seven times that day. The seven tallies on the paper: IIIIIII map *onto* the abstract number '7.' In other words, the tallies on the paper are *isomorphic* to the mathematical number 7. In the physical world, there are many ways to perform computations—we can use an abacus, manipulate pebbles on the beach, draw marks on paper, etc. We can draw a correspondence between the class of objects in the world and the natural numbers. Turing machines help to make this clear because they operate in both realms. Recall that Turing formulated his theory in terms of an abstract machine that operates on an infinite tape using three symbols: 0, 1, and blank spaces. The data domain for a Turing machine is the set of relevant tape configurations, which can be associated with a set of binary strings made up of 0s and 1s. We can study symbol manipulation using an abstract Turing machine without saying anything about the physical world. But we can also think about *implemented* Turing machines: physical machines like the Analytical Engine or the Manchester Baby that perform computations over physical structures representing natural numbers, and we might ask, what is the relationship between a mathematical Turing machine and, say, the physical states of the Analytical Engine?

3.3 Algorithms

An algorithm is a set of step-by-step instructions for solving some problem—a sequence of operations that can be followed to produce a result or an output. Algorithms are often expressed in a high-level programming language. A high-level programming language is a type of programming language that provides a higher level of abstraction from the underlying computer hardware. It is a higher level of abstraction because algorithms are implemented in programs where programs depend, in some ways, on the computer's hardware. For example, an algorithm can be implemented in various program languages such as Python, C, C++, JavaScript, or Ruby. However, algorithms are not limited to the context of computer science—in fact, we encounter many different kinds of algorithms in our everyday life. The most common everyday life example is a cooking recipe. Recipes are algorithms for preparing food because they provide step-by-step instructions for ingredients, cooking times and temperature, and other information needed to prepare a dish. There are also mathematical algorithms. You might remember the Euclidian algorithm from algebra class—instructions to find the greatest common divisor of two numbers. Some other examples come from science, such as the Monte Carlo algorithm that can simulate physical systems and predict their behavior.

3.4 Programs

A computer program (sometimes called "software") is a set of instructions a *physical computer* can execute to perform a task or solve a problem. A computer program is written in a programming language that can be simple or very complex. A programming language is a formal language that specifies a set of instructions that can be used to create a computer program. A programming language consists of a syntax (more on this later), a set of keywords, commands, and constructs that can be used to specify the logic and control the flow of the program. Different programs have different syntax and features and some or more suited for particular tasks than others. Programs are at a lower level of abstraction than algorithms, but programs themselves can also come in different levels of abstraction. The difference between high-level and low-level programming languages is that high-level programming languages, such as Python, Ruby, and Java, are designed to be easy to read and write. On the other hand, low-level programming languages provide more control over the hardware and are typically used for performance-critical tasks or systems programming. Some examples include Assembly and C (McGrath, 2019).

A program and an algorithm are related but distinct concepts. The relation between the two is one of implementation in the *computer science sense*. When a program implements an algorithm, it means that the program expresses the steps of the algorithm in a specific programming language and performs the computation specified by the algorithm. In the practice of computer programming, programs are created by combining instructions that are meaningful within the programming language. For example, Python instructions that specify the addition of two numbers will give the following instructions: SUM = NUM1 + NUM2. When the inputs are 15 and 12, the program will execute the instructions and produce the following output: "Sum of 15 and 12 is 27."

An even lower level of abstraction is called the "machine code." The machine code is the low-level representation of the program that is executed directly by a computer's central processing unit (CPU). Machine code consists of binary instructions that correspond to the basic operations that a CPU can perform (such as moving data, performing arithmetic operations, and moving between different parts of a program). This is the bottom-most "layer" of what makes up the "software" of a computer. To run a program on a computer, it must first be compiled (translated) into machine code. This process is performed by a compiler that takes the program's source code and converts it into a series of machine instructions that the physical computer can then execute. The compiler is another program typically located on a computer's hard drive but varies depending on the operating system or programming language. After the compiler encodes the program as binary strings (bits that take the form of 0's and 1's), the bits are physically realized as voltage levels maintained by the hardware.

Ada Lovelace is credited with creating the foundations for the first ever computer program. Lovelace's program was intended to work on Babbage's Analytical Engine. Lovelace was optimistic about the possibilities of a machine like this—she envisioned it as one that could do more than simple mathematical equations. She thought that with the right program, it would be able to act on any object "whose mutual fundamental relations could be expressed by those of the abstract science of operations, and which should also be susceptible of adaptations to the action of the operating notations and mechanisms of the engine" (Lovelace, 1843). Essentially, Lovelace's vision was a machine similar to the modern-day computer.

Although Lovelace is credited with creating the first computer *program*, I think it is better to understand her project as creating the first *algorithm* specifically for use on a physical, digital computer. Here algorithm was designed to allow a machine to compute the Bernoulli numbers, a set of natural numbers that regularly appear in the sums of squares, cubes, and higher powers. Her program was written for one step of the process, the calculation of a number that she called "B7"—the 8th Bernoulli number. Her program solved the following problem, where each term represents a coefficient in the polynomial formula for the sum of integers to a particular power: $\boldsymbol{B_7 = -1(A_0 + B_1A_1 + B_3A_3 + B_5A_5)}$. Unfortunately, her program never made it beyond the stage of a list of operations specified with mathematical symbols. This is why I think it is best understood as an *algorithm*—it took the form of natural language operations that could then be later translated or *implemented* in a programming language to create a computer program. In fact, some have done the work for her using her diagram. Her program has been successfully translated into several programming languages, including C and Python.

3.5 Syntax

In computer science, syntax refers to the set of rules for constructing a correctly formatted program in a particular programming language—it specifies how instructions or statements in the program must be written so that they can be parsed and interpreted by the computer. The syntax defines the *structure* of a program, and it includes various elements such as keywords, variables, operators, and punctuation. For example, a syntax rule programmed in Python might specify that a statement must end with a period or that a variable name must start with a letter. The syntax can only contain letters, numbers, and underscores. More simply, syntax works much like grammar in structuring sentences in a natural language.

The syntax is written using natural language symbols using words and punctuation marks (although it does not read like sentences in English). For example, in Python, indentations are used to indicate blocks of code within the syntax. Each block of code within a function, loop, or conditional statement must be indented by the same amount to indicate that it belongs to the same block. Below is an example of a Python code that indents each line of code with four spaces:

```
def example_indentation():
    print
    print
```

The term ‘def’ is a keyword that is used to define a function. The phrase ‘example_indentation()’ is the function defined by ‘def,’ indicating that the text in the following lines should be indented four spaces. The ‘print’ text in the bottom two lines is standard code in Python that is used to output text to the console. This text-based interface allows the programmer to interact with a computer program by entering text commands and receiving text output. The programmer types ‘print’ to execute the function specified in the code (McGrath, 2019).

3.6 Semantics

In computer science, ‘semantics’ refers to the meaning or interpretation of the symbols and instructions in a computer program—it refers to what the program *does*. Syntax and semantics come apart and also depend on each other in different ways. A good example of how they come apart famously comes from a natural language example given by Noam Chomsky (1957):

Colorless green ideas sleep furiously.

The sentence has correct grammar—the syntax is specified correctly, but the meaning is nonsensical. Put differently, you can have a well-specified syntax that, nonetheless, lacks any meaning. However, both syntax and semantics should generally be defined together. Below is an example in the C/C++ programming language that clarifies the connection. We can specify the following syntax:

```
int x;
x = 2 ^ 3;
```

The above code is syntactically correct, we just do not know what it is *for*. What value does the variable *x* end up with? We can only solve this if we know what the carrot symbol (^) means in C/C++. This symbol is bitwise XOR – a binary operation that takes two-bit patterns of equal length and performs the logical exclusive OR operation on each pair of corresponding bits. The XOR function compares the first bit to the second bit. If both bits are the same, then the answer is '0.' If the bits are different, then the resulting answer is '1.' Now knowing what the carrot symbol means, we know that *x* is equal to 1.

The above example describes what is sometimes called an *internal* semantics in computer science. Notice that the meanings are assigned to the programming language that tells the *computer* what to do. This type of semantics concerns symbolic manipulations *inside the machine* having to do with the programming language. Because of this, it is possible for the same symbol to have different semantic interpretations across various programming languages. For example, '+' means addition in the C++ programming language (as we would expect it to). But, in Python, the '+'

symbol is used for string concatenation—the process of joining two or more strings, arrays, or data structures into a single string.

External semantics, on the other hand, is the meaning of a computer program or its parts as interpreted by a source *external* to the computer, such as a human. It is meaning that is ascribed outside of the machine's source code (the set of instructions written in the programming language as described above). The external semantics might be influenced by culture, context, or the specific domain of use. An external semantics can be understood as a human-centered view of the program's meaning outside of its source code. For example, the external semantics of Microsoft Word is that it is a word processing software used for creating and editing text-based documents. It is used to help create, format, and edit many different kinds of documents, including dissertations. Its key features include text, tables, images, templates, and the ability to collaborate with others. Contrast this with the internal semantics—the *technical* details of *how* the software is designed to solve the problem it was created to address (creating documents). This is the underlying structure and design of the software itself. Colloquially, when people refer to the 'semantic' interpretation of some computer program, they are referring to the *external* semantics—the intended use—what the program does *for us* (McGrath, 2019).

4 Some philosophical questions that arise

In the first section of this chapter, I gave the basic historical lineage of analog and digital computers, including some examples of each type. In the second section, I presented and explained some terminology for understanding some concepts from computer science. These concepts are important for understanding my dissertation project as well as some key philosophical debates that engage with these concepts. In the next section, I will briefly introduce some of the philosophical questions that arise when it comes to thinking about computing systems.

4.1 Characterizing the analog/digital distinction

At the start of this chapter, I discussed how analog computation originated with the idea of analogicity. Then, due to engineering considerations, analog computation became synonymous with *continuous* processing. Continuous processing requires continuous *representations*. In this context, the term ‘representation’ is invoked in an engineering sense meaning that information is represented in the system by means of signs or signals. Analog representations are said to be continuous because they are processed smoothly. This is contrasted with digital computation, which is said to process information in a stepwise, discrete-state style fashion. In this context, ‘continuous’ and ‘discrete’ are understood mathematically: discrete representational schemes are bijective with (a finite subset of) the natural numbers, and continuous representational schemes are bijective with the reals (Schonbein, 2013). Put differently, continuous representations vary by arbitrarily small degrees, while digital representations do not. This distinction forms the basis of the ‘received view’ regarding the difference between analog and digital. Analog computation, then, represents information by means of a signal whose information parameter is a physical quantity which may assume any value within a given range at any instant within a continuous time interval. Digital computation, on the other hand, involves the representation of information by means of a digital signal whose parameter may assume one out of a set of discrete values within a given range (Frigerio et al., 2013). The received view, then, relies on modes of representation to frame the dichotomy.

The received view is traced back to Goodman (1968), whose project involved distinguishing between types of symbols and symbolic structures. According to Goodman, how a symbol denotes depends on the system of symbolization in which the symbol is involved. Whether a symbol denotes depends on whether it meets specific syntactic and semantic requirements.[4] Roughly, for a symbol to denote, the symbol set to which it belongs must *not* be syntactically *dense*. A set of symbols is syntactically dense if it provides an infinite number of characters arranged so that there will always be a third between the two. In other words, we cannot cleanly distinguish between the symbols. Goodman explains this by drawing a comparison using analog and digital systems. Analog systems are *dense* because we cannot differentiate between their symbols. In contrast, digital systems are not dense because the symbols are discrete. A consequence of this view is that analog representations *do not* denote while digital representations do. Because part of Goodman's project draws a distinction between analog and digital, his view forms the basis of the earliest analytical argument for understanding the difference between the two. To put his view in context, Goodman aimed to give an understanding of symbols in both the sciences and ordinary life. According to Goodman, we use symbols for perceiving, understanding, and constructing the worlds of our experiences. So, understanding how symbols are used is an important part of understanding our world (Giovannelli, 2017).

Lewis (1971) engages with Goodman, arguing more specifically for a view that distinguishes between analog and digital representations of *numbers* specifically. Lewis argues that there are cases of non-dense analog representations and that differentiated representations can be analog in some circumstances. In other words, the distinction between analog and digital in terms of density

[4] Although there are some similarities to the way that Goodman is using the terms 'syntactic' and 'semantic,' we should not associate them with the syntax/semantics distinction drawn in the previous section.

only works for some cases. He gives an example of a particular circuit to demonstrate how analog representations might be differentiated. He says an analog computer might represent a number by the resistance ohms on the circuit using a switch with various positions. The switch could fix on particular ohm resister making it distinguishable from other settings. He then gives a second example of a digital machine with continuous representations. In this case, we would understand a digital representation in terms of differentiated multi-digital magnitudes. He suggests that analog representation involves relating the right sort of magnitude, in the right way, to the right sort of magnitude. Since the right sorts of magnitude need not be continuous, neither does analog representation. Lewis evolves Goodman's symbol distinction by generating a view that reframes the debate in terms of analog and digital *representation*.

Haugeland (1981) is interested in whether an analog machine can be simulated on a digital machine. Part of answering this question requires engagement with the definitions of 'digital' and 'analog'—or developing an understanding of the difference between them such that we can say that one type of machine is simulated on another type of machine. His interest is in machines in general, but his aim requires saying something about the nature of analog and digital representations in computing systems. For Haugeland, digital machines have three features: *copyability* (they can flawlessly copy and preserve), *complexity* (they are composites formed from components where the components can be understood as parts of the whole or diachronically, as steps or moves in the system), and *medium independence* (a digital machine can have equivalent structures made from different media) (213-214).[5] He adds that for a digital device to follow

[5] Piccinini (2015) uses the term 'medium independence' in the same way but attributes his usage to Garson (2003). Garson does not cite Haugeland in his 2003 paper.

procedures that involve writing and reading symbols, it must have some specification of its operating system. If it does, it can reliably follow any read-write procedure perfectly and without error.

Haugeland also describes the features of analog systems. For analog machines, the procedures for the write-read cycle are *approximation procedures* (221). They are approximation procedures because they can only "come close" to perfect success. While digital machines execute the procedures flawlessly, analog systems execute them with some margin of error. 'Approximation' replaces Goodman's claim that analog symbols are *dense*. Hougeland thinks that the idea "between any two symbols there is a third" is challenging to make sense of because "betweenness" is an ill-defined concept. Instead, he offers the concept of "approximation" to make sense of the continuous nature. He also provides the following features unique to analog machines: *smoothness* (variations are smooth or continuous without "gaps"), *sensitivity* (every difference makes a difference, e.g., the smallest rotation of a rheostat counts as changing a setting), and *dimensionality* (even with sensitivity, still only specific dimensions are relevant, e.g., slides rules can be made indifferently of metal or bamboo) (220). But it is the concept of 'approximation' that leads Haugeland to consider an interesting consequence of his view:

> "…it seems to me that there is an important digital-like character to all the standard analog devices - specifically in the dimensionality feature. Speaking freely for a moment, the essential point about (atomic) digital types is that there tend to be relatively few of them, and they are all clearly distinct and separated. Though the types of analog schemes are themselves not like this (they "blend smoothly" into one another), the dimensions along

> which they vary are relatively few and clearly distinct… the best example is a regular analog computer with its electronic adders, integrators, multipliers, inverters, and the like, each as discrete and determinate in type as any mathematical symbol, and their circuit connections as well-defined as the formation of any equation. Indeed, though the state and adjustment types of an analog computer are analog, the set-up types are perfectly digital - the component identifications and interconnections are positive and reliable" (222).

He calls the above-mentioned 'positive and reliable digital set-up types' a kind of "second order" digitalness. He goes on to say that second order digitalness seems to be a necessary condition for what enables the write and read procedures of *multidimensional* analog systems to achieve *approximation*. One-dimensional devices include simple systems like slide rules, while multidimensional systems include more complex machines that can carry out copy procedures. For a machine to carry out copy procedures sufficiently, some precision components must be present— this is the only way we can make sense of approximation. This consequence is interesting because it challenges the dichotomy in a different way. Although the distinction still exists in terms of continuous and discrete, Haugeland begins to wonder whether the distinction between analog and digital is as clear-cut as previously thought.

There has been a substantial amount of theoretical work on the analog/digital distinction that engages with the debate regarding the nature of cognitive representations (Fodor 1975, Kosslyn 1981 and 1996, Pylyshyn 1981 and 1986, among others). Some draw the distinction in terms of resemblance. Analog representations resemble their referents, while digital representations do not. However, drawing the distinction in terms of resemblance requires saying more about what it means to *resemble*. Analog representations can be understood as representing their referents by simply "standing in" for the referent without being an exact duplicate. But, whether this counts as genuinely representational in this context ends up being a substantial part of the debate. For example, mere resemblance will fail to qualify as genuinely representational if you think cognitive representations must have a propositional nature (Fodor and Pylyshyn 1988). You might also dislike this characterization if you follow Goodman's abovementioned distinction. Resemblance under this characterization would count topological diagrams as analog representations and Goodman explicitly argues against this because resemblance is "too loose" of a notion to capture anything useful (Goodman 1968, pgs. 170-171).

Much of the debate on the analog-digital distinction and mental representations engages with the nature of representations. This follows early work on the distinction that frames the difference in terms of analog having a "copy-like" format while digital does not. But there is reason to resist conceiving of the relation between analog and digital forms of representation in terms of the relation between pictorial images (analog) and linguistic propositional descriptions (digital). This seems to assume that the difference is something like resemblance versus discreteness when the question is whether this is the appropriate way to draw the distinction in the first place. As a response, some have proposed views that are aimed at separating these issues.

To address the analog-digital distinction, Katz (2008) distinguishes the *format* of representation from the *medium* of representation. To demonstrate, he asks us to consider several ways of representing the natural numbers from zero to nine.

A) A beaker filled with liquid increments that correspond to 0-9,

B) marbles partitioned such that each one represents a number 0-9,

C) a series of beakers where each holds an increment of water that corresponds to 0-9, and

D) a large beaker with marbles that are poured in equal increments that correspond to 0-9.

Intuitively, A and B are typically understood as analog and digital representations, respectively when we consider the format the representations take. However, this intuition can be challenged when we identify each system in terms of the *format* and the *medium*. Whatever stuff is used to represent is called the medium of representation, such that, A and C (water) and B and D (marbles) have the same *medium* respectively. On the other hand, call whatever structure is imposed on that medium the format of representation, such that A and D (equally poured increments) and B and C (equally distributed increments) have the same *format* respectively. From this distinction, A and D are analog while B and C are digital. Thus, according to Kats, the analog-digital distinction concerns the *format* of representation, and not the *medium*. Kats goes on to say that what drives the distinction between analog as continuous and digital as discrete (and what makes us intuitively call 'A' analog and 'B' digital) is how the representational system appears to the user. System C's representations *appear* discrete to the user while system D's representations *appear* continuous even though A and C, with the same medium of representation, invoked a different response.

Maley (2009) also pushes back on the idea that we should draw the distinction in terms of continuous versus discrete. Instead, he argues that the difference between the two concepts tracks something else. To demonstrate, he asks us to consider not just the digit, but the *base* used to interpret the relative value of the digit. He uses the following example of the number three hundred and forty-eight in base 10. ‘348’ has ‘3’ in the hundreds place, ‘4’ in the tens place, and ‘8’ in the ones place with a value of (3 x 10^2) + (4 x 10^2) + (8 x 10^0). This is much more complex than just viewing the digit on its own. When accounting for the base, he argues that we should characterize digital representations in terms of the following formula:

> A series of digits, each of which is a numeral in a specific place within the series, and
> A base, which determines the value of each digit as a function of its place, as well as the number of possible numerals that can be used for each digit. (pgs. 124-125)

The idea is that what a representation represents in a digital system cannot be read off the digit itself. For example, ‘3’ cannot be understood outside of its context. Without specifying the context, ‘3’ could represent 300, 3,000, or 3 depending on the base. The difference here is that individual representations are not by themselves discrete*;* they are discrete *within* the representational scheme that they are a part of. Thus, associating digital representations with discrete inherently misses the nuance of digital computing.

Piccinini (2015) rejects the dichotomy by dismissing the role that representations play in computation altogether. Before doing so, he briefly mentions that a system can only be said to be digital or discrete under a given mathematical description, which applies to a system at a certain level of analysis. The way to determine whether physical systems are ultimately continuous or discrete depends on the fundamentals of physics. Some people think that everything turns out to be discrete at the fundamental level. At the level of middle-sized objects, differential equations are used, so everything seems to be continuous. But ultimately, he thinks that the kinds of representations a system processes should not matter because he thinks that computers should be understood independently of what they represent or if they represent at all (199-200).

Instead, he distinguishes analog and digital computers based on their mechanistic properties. The main difference is the vehicles that they manipulate. The inputs and outputs of digital computers are strings of digits, and the inputs and outputs of analog computers are continuous or real variables (physical magnitudes that vary over time, take on a continuous range of values within certain bounds, and vary continuously over time). "Real variables" are physical magnitudes that vary over time, take a continuous range of values with certain bounds, and vary continuously over time. He cites the rate of rotation of a mechanical shaft and the voltage level in an electrical wire as examples. He says, one of the main differences here is that digital systems operate over *strings of digits,* whereas analog systems do not.

Yet, this *still* seems to characterize the difference in terms of discrete and continuous, just within a mechanistic framework. But ultimately, he argues that analog computers do not perform computations in the sense defined by the mathematical theory of computation. In other words, we cannot apply Turing's notion of a computable function to analog computers because there are no strings of digits in analog systems. So, perhaps, the best way to draw the distinction for Piccinini is in terms of whether a system can perform Turing-computable functions or not.

However, not everyone thinks the received view gets it wrong. Schonbein (2013) offers support for the dichotomy based on how it emerges from a theory of computation (in computer science) and because it allows us to explain engineering practices. Schonbein asks us to consider the Turing machine and its components—a Turing machine is defined by specifying the finite set of controller states for the machine and the rules for transitioning between states given inputs and the current contents of memory. The inputs and memory contents are the representations the system uses, and they are "*thoroughly discrete*" (p. 416). This is important, Schonbein says because the problem that Turing was trying to solve *required* it. Further, he argues that part of Turing's view was that continuous representations schemes posed a problem for describing what humans do when they calculate. Moreover, because all digital computers were engineered in accordance with Turing's account of computation, we have further evidence that digital computation necessarily requires discrete representations while analog does not (418).

Whether we should accept or reject the received view is an ongoing debate. I have given the origins of the debate insofar as how the received view was developed with ties to the work of Goodman, Lewis, and Haugeland. I have also surveyed several ways that either challenge, endorse, or reframe the distinction. There is still plenty of work to do on this topic, and the consequences of this debate will likely have strong influences across several other debates, such as the nature of mental representations, the nature of cognitive computation, and a theory of physical computation.

4.2 Computation, information, and information processing

Computation is typically used to process information, so much so that the notions 'computation' and 'information processing' are often used interchangeably. Even so, there are important differences between the two.[6] Because information plays different roles in different fields and

[6] Even if you think that computation *always* amounts to information processing, there is still reason to keep them conceptually distinct. For example, it seems that different kinds of computational systems will process different kinds

comes in different forms, we need a clear understanding of what "information" is in the first place. Floridi (2010, 2011) describes the concept as "polysemantic" because it is associated with different phenomena such as communication, knowledge, reference, meaning, and truth, to name some. In the mind sciences, information is often invoked to explain cognition and the production of behavior, as seen in Miller (1951) and Minsky (1968). As we have seen, information also plays a role in communication engineering, where information is central to the design of an efficient communication system (Shannon 1948). It is also used in biology (Godfrey-Smith 2000) and animal communication theory (Bradbury & Vehrencamp 2000). Information is also deployed in philosophical theories, which aim to provide a naturalistic grounding for the intentionality of mental states (Dretske 1981, Millikan 2004). Given the many contexts in which the concept of 'information' is deployed, we should make clear how information processing and computation fit together.

In section 2.1, I described Shannon information in detail. Although entropy has been the main tool in the analysis of the concept of information for the past seventy years or so, there were other attempts at formulating a general theory of information, such as Fisher (1935) and Li and Vitanyi (2008).

Fisher information is a concept that comes from mathematical statistics that measures the amount of information that an observed dataset contains about an unknown parameter of interest. In other words, it quantifies how much information the data provides about the value of a parameter. The

of information—natural information, non-natural information, etc., and how we understand the computational system may turn on the type of information that the system processes. Moreover, Piccinini (2015) has provided extensive argumentation worth considering for why the two concepts are distinct.

larger value of Fisher information indicates that the observed data contains more information about the parameter, and thus the parameter is more easily estimated from the data. Fisher information plays a crucial role in statistical inference, hypothesis testing, and model selection in contexts such as Frequentist Statistics and Bayesian Statistics (Ly et al., 2017).

Kolmogorov theory provides a mathematical framework for describing the probability of events in a random process (Li & Vitanyi, 2008). It defines a probability space, which consists of a set of possible outcomes, a set of events, and a probability measure that assigns probabilities to events. One of the key contributions of Kolmogorov's theory is the concept of stochastic processes, which are sequences of random variables that represent the evolution of a system over time. This theory has applications in many fields, including physics, economics, finance, and engineering. It is used to model and analyze complex systems that exhibit randomness or uncertainty, such as financial markets, weather patterns, and the behavior of particles in physical systems. A semantic theory of computation that I explore in Chapter 6 will rely on this notion of information.

But mathematical information theories such as Fisher information and Kolmogorov theory do not seem to be the theories that will help us understand what information amounts to in physical systems—especially computational systems. One reason that the idea of *processing* Shannon information is also unclear is that Shannon does not tell us what information *is*.[7] As Floridi (2004) puts it, we know that information ought to be quantifiable, additive, storable, and transmittable…but apart from this, we still do not seem to have a clear idea about the specific nature of information (p. 3). This is echoed by MacKay (1983), who says that as early as our 1950

[7] Schroeder (2004) attempts to provide an alternative to entropy in the measurement of information, but even so, this view still does not help us to understand what information is or how to understand a system as processing information.

Symposium on Information Theory, it was somewhat scandalous that the theory of information seemed to have so little working contact with such concepts as the *meaning* and *relevance* of information.

What might help us understand information processing is a *semantic* theory of information. Semantic theories of information emerged as a supplement to Shannon information meant to provide a more general theory of information that could provide a usable framework for calculating the *amount* of semantic information encoded by a sentence in a particular language. The first account comes from Carnap and Bar-Hillel (1952). For Carnap and Bar-Hillel, the amount of semantic information encoded by a sentence is inversely proportional to the likelihood of the truth of that sentence. This idea is cashed out in terms of the set of all possible worlds where given that the intention of a sentence is the set of all worlds in which the sentence is *true*, and that the content of a sentence is the set of all worlds in which the sentence is false (Sequoiah-Grayson & Floridi, 2022). But some philosophical problems arise. For example, how do we know in practice how many possible worlds there are? Also, if we are talking about the number of possible worlds with respect to all possible sentences in English, then there will be infinitely many of them which complicates things even further. [8]

The philosophical literature has contributed to the connection between semantic content and information by generating theories of meaning that turn on modes of presentation. The earliest view of this sort comes from Frege's *Sense and Reference* (1948). This view takes meaning and information to amount to pretty much the same thing, which gives rise to the following idea: the

[8] Other semantic theories of information include Floridi's (2004 & 2005) 'strongly semantic information' account.

word ‘cat’ denotes the property of being a cat, and it means *cat* because it expresses the concept CAT, and the concept CAT means *cat*. Typing meaning and information together in this way gives us a naturalistic *causal* story about information and meaning. Such information-transmitting causal relationships count as *information channels* or causal connections that facilitate the flow of information between the source of information and the receiver. Other views that build on the idea of informational semantics include Fodor (1990), Dretske (1981), Evans (1982), Recanati (2012, 2016) and Jackson (2010). These views vary in different ways, but they all characterize a theory of meaning in terms of *information channels*.

There has been some success for computational theories of cognition that understand cognitive computational systems as carrying something like naturalized information in the above sense. But it is unclear whether this idea can be extended to non-cognitive computational systems. Put differently, our computational artifacts are not the kinds of computing systems that carry natural information. Computational artifacts, if they represent at all, have something like non-natural meanings—or conventional meanings that are ascribed to the system based on context.

4.3 Drawing the syntax/semantics distinction

In sections 2.5 and 2.6, I specified a distinction between syntax and semantics in computer science. But because computation is deployed to explain cognitive processes, the distinction between syntax and semantics outside of the context of computer programming gets murky in ways that matter for a theory of physical computation. The syntax/semantics distinction and computation come together in the philosophy of mind in ways that affect the way we think about syntactic and semantic properties. In this section, I will explain how computation relates to mental processes, why the syntax/semantics distinction is important in this context, and how the distinction is framed. This section draws heavily on 'the computational theory of mind' or *computationalism*, explored in more detail in the next chapter.

In 1980 Jerry Fodor published a famous paper titled 'Methodological Solipsism Considered as a Research Strategy for Cognitive Psychology.' In this paper, Fodor outlines a theory of mind that (he thinks) can be scientifically investigated. This is the idea that the mind—including intentional states—can be understood as a formal arrangement of symbols. Simply put, intentional states are the kinds of mental states that are *about* the world—they carry information about the distal environment. For example, our beliefs count as intentional states because our belief states, whether true or false, are about features of the external world.[9] Part of Fodor's goal was to make room for intentional states in our scientific practice. To accomplish this, he relies on what he calls the "formality condition." Drawing from computational processes in computer science, the formality condition says that intentional mental states have both *formal* and *semantic* features. In this context, intentional mental states are often called mental "representations" based on the connection between the formal and semantic properties.[10] The semantic properties of a representation are the properties it has *in virtue of its relationship with the world*, properties such as being *true* or being *false*, of being a representation *of* something or being *about* something.

Formal properties, on the other hand, are the properties *of the representation itself*. Fodor defines the formal properties in terms of what they *are not*: "formal properties are the ones that can be

[9] This is explored in more detail in Chapter 2.
[10] Recall that the term 'representation' was used when drawing the distinction between analog and digital. Here, 'representation' is being invoked in a non-engineering sense. Representation in this context can be understood as shorthand for an intentional mental state that has both syntactic and semantic features.

specified without reference to such semantic properties as, for example, truth reference, and meaning" (p. 64). This implies that mental processes have access to the *formal* properties of representational states (as opposed to their semantic properties). This is where the 'methodological solipsism' comes in: "If mental processes are formal, then they have access only to the formal properties of such representations of the environment as the senses provide. Hence, they have no access to the *semantic* properties of such representations, including the property of being true, having referents, or indeed the property of being representations *of the environment*" (p. 65).[11] Fodor's solution to this solipsism is to pair computational psychology with a *naturalistic* psychology—a theory that defines the relations between the representations and the world. According to Fodor, the formal properties of representations mirror the semantic properties so that mental processes operate on formal properties which can be interpreted as having semantic content. Fodor is explicit that formal properties aren't simply "syntactic" properties, but it is reasonable to wonder, then, what the relationship between formal properties and syntactic properties are.

This draws a distinction between semantic properties and *formal* properties. But what about *syntactic* properties? Fodor tells us that 'formality' means something like 'explicitness,' and that there is no reason for thinking that 'formal' means 'syntactic,' although the "ambiguity abounds in the literature" (p. 72). This is because, Fodor says, it is possible that we could have formal processes defined over representations that do not also have a syntax. Syntax is, though, a *species* of formal operations in virtue of it *not* being semantic. Other than syntax not being a semantic

[11] The concept of "methodological solipsism" comes from Putnam (1972). (Putnam, 1975)

property, Fodor says the concept will have to remain "intuitive and metaphoric" (p. 64). But a positive account of 'syntax' is needed beyond the idea that it is just "not semantics."

One place we might look to understand what 'syntax' means is to look to a theoretical alternative to Fodor's view, namely, The Syntactic Theory (Stich 1983). The Syntactic Theory of Mind (STM) considers cognitive states as uninterpreted syntactic objects. According to Stich, the causal relations among cognitive states mirror formal relations among syntactic objects (p. 149). The thrust of this view is Stich's position that we can do psychology without appealing to the representational content of psychological states. A view of this kind, then, should make clear how to understand the non-representational features of psychological states, including what it means to be a syntactic property. Stich tells us that abstract syntactic objects are syntactic because they *lack relevant semantic properties* (p. 153). He goes on to say that abstract syntactic objects are "sets of counterfactual relations among stimuli, behavior, and other psychological states" and that the "status of a state as a token of [an abstract syntactic object] does not depend on what other cognitive states a subject currently happens to be in… it depends only on the causal interactions that the state would exhibit with stimuli, with behavior, and with other states" (p. 153). He later discusses *syntactic form* and *syntactic relations* between cognitive symbols. However, this still does not tell us how to characterize the distinction. So, we still need to look elsewhere.

Maybe we can characterize syntax by looking at linguistics. Consider Chomsky's distinction between surface and deep structures (Chomsky, 1965). The deep structure refers to a sentence's underlying meaning or semantic structure, while the surface structure refers to the actual form of the sentence. This, paired with a transformational grammar, demonstrates how the deep structure of a sentence can be transformed into different surface structures through a series of grammatical operations. Syntactic properties relate to the *structure* or *arrangement* of the words and phrases in a sentence. The study of syntax, then, has to do with understanding the rules that govern the formation of sentences and how they can convey meaning in virtue of their formation. This can be contrasted with the formal properties of language. Formal properties are the abstract properties of language independent of their meaning, such as phonology, morphology, and *syntax*. Here we have support for Fodor's claim that syntax is a species of formal properties.

Formal properties include everything but the semantics, including syntactic features. Syntactic features are those features that have to do with, roughly, the relations between the words or, more generally—between the *symbols*. These relations are governed by a grammar (or some set of causal rules) that determine how the symbols are arranged into well-formed symbol strands. So, the syntax concerns the *how* part of the arrangement of symbols. We can now say something meaningful about the distinction between syntax and semantics (with some help from the notion of formality). For clarity, I will specify the distinction in terms of symbolic structures.

Semantics: what the symbols mean or what they refer to.

Formal properties: non-semantic features, including transformational rules and the symbols' form, such as shape and size.

Syntax: the relations between the symbols (perhaps understood as causal properties).

In this section, I have distinguished syntax and semantics as understood in the philosophy of mind. To do this, I have explored how we might understand 'syntax' by giving a positive definition.

Much of this understanding draws from computer science and linguistics and is motivated by a desire to explain how contentful mental states can be transformed according to their physical properties (as opposed to their meanings). This section gives a foundation for later chapters that engage with computational theories that aim to capture semantic content in this philosophical sense. Sometimes, they do so by drawing a distinction between syntax and semantics, as it is understood in the philosophy of mind.

4.4 Internal versus external semantics

I have discussed the distinction between internal and external semantics when it comes to computer science and programming, but there is more to be said about this distinction and how it applies to a theory of computation and a computational theory of mind. Some psychologists and neuroscientists believe that minds can be explained computationally and that the computational explanation cannot be given without appealing to representations. This brings computation and theories of mind together. Moreover, it couples computation with an *external* semantics. But the distinction between an internal and external semantics must be made explicit in the context of a theory of computation, even if that theory is meant to extend to computational explanations of the mind.

An internal semantics is confined within the physical boundaries of a computer. The internal semantics is concerned with the meaning of elements as defined *within* the system itself. An external semantics has to do with how the elements of a system are related to things outside of the system, such as the real world or other systems. When it comes to computing artifacts, we define the external assignments by *interpreting* the internal states as representing something in the external world. Mental representations are an example of having external semantics: internal mental states represent the distal environment. The difference between computing artifacts and the mind, when it comes to an external semantics, is that mental representations seem to have their content assignment *naturally* (independently of some observer). In contrast, computing artifacts have them in virtue of the user or context.[12]

5 Conclusion

In this chapter, I have introduced some early computing systems up to and through the development of computability theory. This is relevant for a theory of physical computation in different ways. Some theories of physical computation might be framed in terms of digital or analog computation. You might wonder if we need a more neutral theory if we want to capture the *nature* of physical computation. It also matters how we draw the distinction in the first place, and as we saw, the difference is not straightforward. If you think that computation necessarily involves the processes of information, what does that mean? What *is* information? If we want the theory to capture mental computations, how does that change our notion of information? Are there computing systems that do not process information? These questions, and many others, will play a role in a theory of physical computation—especially if our goal is to capture the *nature* of physical computation that we can apply not just to computing artifacts but to natural computing systems, too.

[12] This difference between these two ways of having content is sometimes discussed in the context of the symbol grounding problem (Harnad 1990).

Chapter Two: Functionalism, computationalism, and physical computation

1 Introduction

Functionalism and computationalism are distinct but related theses about the mind, and both play an important role in accounts that address physical computation. Functionalism is the view that what makes something a mental state of a particular type depends on how it functions or its role in the system of which it is a part. Computationalism is the view that the functional organization of the brain is computational. These two theses are sometimes brought together to form a view called computational functionalism. An early version of computational functionalism comes from Putnam (1960, 1967a). Putnam's computational functionalism says that any creature with a mind can be regarded as a Turing machine whose operations can be fully specified by a set of instructions. This view associates mental states with states on a machine table specified in terms of their relations to inputs, outputs, and the machine's state at a given time. This provides us with a model for understanding mental processes mechanically using an abstract functional description cashed out in terms of computational states by drawing an analogy between the individuation of conditions of mental states and those of Turing machine states. This view is the basis for computationalism, which adopts the metaphysical position that the brain is a computing system.

Computationalism, the view that the mind is a computer and that mental states are computational states, is a controversial and ongoing debate in the philosophy of mind. A theory of physical computation is meant to assist with confirming or denying this thesis. If the theory of computation tells us how to pick out the brain as a computing system or it tells us which computations the brain performs, then we have confirmation for the computationalism thesis. Thus, a theory of physical computation and the computationalism thesis are closely connected. Some argue for a theory of physical computation with this motivation in mind.

In section two, I will explain functionalism and computationalism and explain the relationship between them in section two. This groundwork will give insight into the motivations for early mapping theories of implementation and the triviality arguments that challenged them (chapters 4 & 5). It will also be relevant when I discuss the differences between the mechanistic account of physical computation and how traditional theories conceptualize the relationship between computation and physical systems (chapter 7). In section three, I will discuss how computationalism relates to the methodology of cognitive science. In this section, I will carve out a place for how to understand theories of physical computation within Marr's tri-level explanatory framework.

2 Functionalism and computationalism

2.1 Functionalism

Functionalism is the view that what makes something a mental state of a particular type depends on how it functions or its role in the system of which it is a part. This view understands mental states in terms of their functional organization rather than their physical constitution. Functional organization is often cashed out in terms of causality, where the functional organization includes the causal relations between a system's internal states, inputs, and outputs. The causal notion applies to all systems with inputs, outputs, and internal states, which means that any system that has the appropriate functional organization can be said to have a mind.

Functionalism has played an important role in debates over the metaphysics of the mind. Some saw the view as a way of avoiding the problems associated with mind-brain dualism and the materialist view known as the identity theory of mind (Place 1956 and 1960, Feigl 1958 and 1967, Smart 1959, Presley 1967). Although functionalism is neutral between materialism and dualism, it is particularly attractive to materialists because it seems that the functional roles in question will be physical states. This also makes it an enticing alternative to mind-brain identity theory because it allows us to specify likely physical mental states without identifying their physical realizers. This is because functionalism accommodates the multiple realizability of mental states.

The multiple realizability of mental states is Putnam's direct response to Smart's identity theory. The identity theory espoused by Smart says there is an identity relation between psychological descriptors and mechanisms in the physical world; there is nothing over and above brain processes—they are one in the same thing. Putnam argues that very different kinds of creatures might have specific mental properties in common while having different physical properties. This means that a theory that associates mental states with specific brain states in the way that Smart does will necessarily fail to attribute mental states to organisms or artificial systems because they have a different physical constitution than humans.[13] Functionalism accounts for the multiple realizability of mental states because it defines mental states in terms of their functional role as opposed to their physical constitution. Functionalism, then, lets us describe mental states along with the processes in which they are involved, abstractly.

[13] I will flag here that this is Putnam's reading of what he calls the "mind-brain" identity theorist. It is beyond the scope of this project to argue for this position, but I think that Putnam's description the identity theorists project frames the debate incorrectly. I think that the identity theorist's position is consistent with multiple realizability and even functionalism.

Functionalism is often thought of in terms of mental "states." Still, a broader notion of functional organization can go beyond states and consider the functional states along with their relations, aggregates of states, components bearing the states, functional properties of the components, and relations between the components and their properties (Piccinini, 2010). A motivation for developing functionalism beyond its typical "state" style description is that functionalism is generally associated with computational states. As Piccinini points out, this makes it quite difficult to disentangle functionalism from its computational counterpart. The functionalism/computationalism connection comes from Putnam's "machine state functionalism," developed over a series of papers in the 1960s (1960, 1963, 1967a, 1967b, 1988). This view is often seen as a replacement for behaviorism and an alternative to the identity theory. According to machine state functionalism, any creature with a mind can be regarded as a Turing machine whose operations can be fully specified by a set of instructions (a machine table). The mental states of the organism are identified with the machine table states specified in terms of their relations to the inputs, outputs, and the state of the machine at the time.

But functionalism does not entail that functional states are computational, which means that we do not have to be committed to the idea that mental processes are computational processes in order to accept functionalism. However, Putnam gives his functionalism account in terms of probabilistic automata, which introduces a way to understand the brain as a computing machine within the functionalist program. As I said above, it is also possible to frame the view in terms of a Turing machine (as Putnam originally did). The difference between these types of computational systems is that the transitions between states are probabilistic rather than deterministic. This means that a Turing machine is just a type of probabilistic automaton with transition probabilities of 0 and 1.

Pairing functionalism with the idea of probabilistic automata gives us the following hypothesis about how to think about a psychological state as being a functional state cast in terms of computation:

1. All organisms capable of feeling pain are Probabilistic Automata
2. Every organism capable of feeling pain possesses at least one Description of a certain kind (i.e., being capable of feeling pain *is* possessing an appropriate kind of Functional organization).
3. No organism capable of feeling pain possesses a decomposition into parts which separately possess Descriptions of the kind referred to in (2).
4. For every Description of the kind referred to in (2), there exists a subset of the sensory inputs such that an organism with that Description is in pain when and only when some of its sensory inputs are in that subset (Putnam, 1967a, pgs. 162-163).

This argument is, according to Putnam himself, quite vague. None the less, machine-state functionalism forms the basis of a theory of physical computation, which aims to connect the "description" given in terms of probabilistic automata to a physical system in order to show *how* the probabilistic automata are "implemented" in the physical world.

2.2 Computationalism

The computational theory of mind (CTM), or just computationalism, is the more general view that the mind is a computer. This is compatible with functionalism, but it is its own unique thesis, as you can accept computationalism without also adopting functionalism, just as you can accept functionalism without adopting computationalism. This dissertation involves considerations that have to do with computationalism specifically. Suppose you adopt a version of functionalism that rejects computationalism. In that case, a theory of physical computation is no longer relevant to the view (although you may need some theory to reify the abstract functional characterization with the physical system it describes). So, the rest of this project concerns functionalist accounts that adopt computationalism or computationalism views that reject functionalism.

One of the primary arguments for computationalism is that it can make clear how mental states are causally relevant in the physical world. It accomplishes this by saying that mental states are syntactic entities that are computed over. Their form makes them causally relevant in the same way that form makes parts of source code in a computer causally relevant. This is closely related to the "formality condition" that says that mental processes only have access to the formal properties of the mental states (rather than their semantic properties) (Fodor 1980).

There are different versions of computationalism available. By associating machine-state functionalism with a Turing machine specifically, we end up with a closely associated view of the mind known as the "Classic Computational Theory of Mind" or CCTM. This view assumes that mental activity is akin to specifically Turing-style digital computation. Turing-style computation relies on the notion of a universal Turing machine (UTM) and accepts explicitly that the mind is a digital computing device. I say Turing "style" because this view draws an *analogy* drawn between the mind and a universal Turing machine allowing for Turing-equivalent style computation or even something like hypercomputation. This is why, even though the analogy is often drawn in terms of a UTM, CCTM only requires a commitment to Turing-*style* computation.

An allure of CCTM is that it works in conjunction with the Representational Theory of Mind (RTM)—in particular, a version called The Language of Thought Hypothesis (LOTH). Fodor (1975) was the first to link CTM with LOTH. He argued that cognitive representations are tokens of the Language of Thought and that the mind is a digital computer that operates on these tokens. The Language of Thought hypothesis proposes that thinking occurs in a mental language sometimes called "mentalese." Mentalese resembles spoken language in that it contains words that combine into meaningful sentences where each sentence's meaning depends systematically upon the meanings of its component words and the way those words are combined (Rescorla, 2019). CTM + RTM with LOTH ascribes to the formality condition, which gives a naturalized philosophical theory about how contentful mental states can be causally efficacious without relying on the meanings of the mental states to carry out the causal work.

But not all computationalism views are committed to the mind as a digital computer in a Turing-style sense, nor are they committed to a language of thought. Connectionism was offered as an early replacement to the CCTM because it seemed more "biologically plausible" given the similarities in structure between artificial neural networks (ANN's) and the neural networks of the brain. An ANN is composed of simple processors or nodes acting in parallel. Much of the parallel processing is spread over various, perhaps widely distributed, nodes, with representational data distributed similarly in a sub-symbolic way. An artificial neural network or connectionist network is sometimes preferred because it seems more biologically plausible than explicitly rule-based digital computational systems.

However, not all neural networks count as non-digital. Thus, it is possible to accept that the mind is both a neural network and a digital computer. In other words, you can accept that the mind is a digital computing system while also adopting an ANN as the computational architecture. ANNs, such as the early models proposed by McCulloch and Pitts, were both neural networks and digital systems. McCulloch and Pitts (1943) suggested that the brain's neuron operations correspond to logical connectives (later "logic gates"). Logic gates are the basic building blocks of digital computers. This brings together an ANN's architecture with a digital computer's logic gate operations. This might be a way of rejecting Turing-style computation while maintaining a digital conception of the mind embedded in a neural network structure. But one might ultimately reject this version of digital computation on the basis that it is not obviously compatible with LOTH because of the distributed nature of the representations.[14] This is because it is unclear how distributed representations can handle the language-like structure of mentalese (its compositional features).

3 Computation and cognitive science

Computationalism is generally assumed to be the central working hypothesis of cognitive science. The main aim of computational modeling in cognitive science is to explain and predict mental phenomena. Here, computationalism and functionalism reunite. Functionalism is made more robust in cognitive science by introducing Marr's hierarchy of explanatory levels. We can pair computationalism with Marr's levels by constraining the types of explanations given within the framework to computational explanations. Marr gives us a computational theory of early vision within the context of computational neuroscience. But I think there is both a broad and narrow

[14] Although there has been a recent proposal arguing for this position. (Quilty-Dunn, Porot, & Mandelbaum, 2022)

way of construing the Marrian framework. The broad way places all *models* (whether they count as information-processing or not) at the algorithmic level. Some models will have physical realizers at the implementation level, while others will not. But, because this project is concerned with what I call the narrow way of understanding Marr's levels—in terms of computational models only, I will not argue for the broader framework here. Instead, I will focus on Marr's framework and computational modeling.

Marr's explanatory framework proposes three "levels" or three distinct but related questions that a modeler should answer when giving a computational theory of an information processing system:

Computational:	What is the goal of the computation, why is it appropriate, and what is the logic of the strategy by which it can be carried out?
Algorithmic:	How can this computational theory be implemented? In particular, what is the representation for the input and output, and what is the algorithm for the transformation?
Implementation:	How can the representation and the algorithm be realized physically?

Sometimes a mathematical function is given at the computational level. This mathematical statement specifies the task being performed in terms of a functional operation or the operation that the system executes when it performs the given task. The algorithmic level involves saying how the system performs the task. This is where we would place a neural network model, or the algorithmic process takes place when the system executes the task specified at the computational level. The implementation level involves the physical realizers that carry out the task. This type of explanation is anatomical in that it identifies the physical features of the system that carry out the task—the physical realization of the process.

Many theorists interested in giving a theory of physical computation are interested in vindicating or supporting the computationalism thesis. If the brain is a computer, we need a theory of physical computation that supports the metaphysical claim that the brain is a computing system. Theories of physical computation answer different questions about physical computation. If a computational description of a physical system is *abstract*, then a theory of physical computation tells us about the relation between the abstract description and the physical system. An abstract description might be a kind of computational architecture, or it might be a computational description of which computation the system is performing. Those who are interested in a theory of physical computation to support computationalism can be interested in either question. In the next chapter, I will define the different questions that I take a theory of physical computation to address, but for now, I will briefly explore how I think these questions relate to Marr's framework in cognitive science. So, right now, I will state the questions I take a theory of physical computation to be addressing without arguing for them just yet. This exercise is meant to relate computationalism to a theory of physical computation within the context of cognitive science.

Chalmers (1995) identifies the question of how to relate a mathematical theory of computation in the abstract with concrete systems in the physical world—a bridge between the abstract and concrete domains. He says that "such a bridge is a theory of *implementation*" (p. 391). According to Chalmers, implementation is the relation that holds between an abstract computational object and a physical system. He then gives his answer in terms of relating the computational object's formal structure to the physical system's causal structure. He specifies these computational objects as those things such as Turing machines, Pascal programs, cellular automata, connectionist networks, and many others. All of these computational objects are versions of combinatorial state automata (CSA). The implementation question then relates the abstract computational structure (possibly understood as a CSA) to the physical causal structures of the physical world. So, within the context of Marr’s framework, this relates the algorithmic level to the implementation level because we will find the computational object at the algorithmic level.

It is sometimes said that the implementation relation can also be understood as a relation between the computational and implementation levels. I disagree. I push back on this understanding because it brings in the identification of a computational function. In other words, it brings in a semantic interpretation. When we relate the function to a physical system, we say which function the physical system computes—we *interpret* the system. This is an answer to what I call the “interpretation question.” An answer to the interpretation question is not an answer to the implementation question. Given this, within the context of cognitive science, a theory of implementation should be understood in terms of the relation between the algorithmic and implementation levels. And a relationship between the computational level and the implementation level corresponds to the interpretation question.

As I will argue, both implementation and interpretation are essential aspects of a theory of physical computation. Within Marr’s framework, we can understand this as relating the algorithmic level to the implementation level (implementation relation), and we can understand it as interpreting what the system computes (giving an explanation at the computational level). In the next chapter, I will define three questions about physical computation, including the implementation and interpretation questions, leaving Marr behind. Nonetheless, because computational descriptions are prevalent in cognitive science, it is worth mentioning here how my framework fits into that methodology.

3 Three questions about physical computation

1 Introduction

We regularly distinguish between physical systems that perform computations, such as computers and calculators, and physical systems that don't, such as rocks and pails of water. We generally take it for granted in our ordinary life that we can categorize objects this way. But there are other contexts where computation is a central explanatory notion, and we need additional theoretical posits to make the categorization. For example, computation is central to psychology and neuroscience (among other sciences), and, despite the claims of some, the brain is not *obviously* a computing system. According to the computational theory of cognition, cognition is a kind of computation, and the brain is a kind of physical computing system. To confirm or deny this theory depends on our computational theory about physical systems in the world: we need to understand the nature of physical computation.

As we saw in chapter one, computation can be studied by formally defining computational objects, such as algorithms, programs, and Turing machines. This is the study of computation in the *abstract*, mathematical sense. But most uses of computation in scientific practice and ordinary life involve physical (concrete) systems. So, there is a close relationship between abstract computation and physical systems that requires explanation. We often understand physical computers as *implementing* a Turing machine or performing a computational process. This relationship, the one between abstract computation and physical systems, is the target of the kinds of computational theories addressed in this project.

A key point about terminology is in order before we move on. First, "theory of computation" is understood as a "theory of *physical* computation."[15] I will sometimes just say theory of computation as shorthand for a theory of physical computation. This should not be confused with a mathematical theory of computation in computer science. As I do this, you will notice a slippage in various places between the terms "theory" and "account." Nothing hangs on this distinction. In this project, I will argue that those who offer a "theory of computation" sometimes answer different questions about physical computation. Because of this, using "theory of computation" interchangeably with "theory of implementation" is problematic. Once my framework is in place, I will argue that not all theories of computation are theories that address the implementation relation. Ultimately, I will argue that a "theory of computation" should be reserved to describe a grouping of theories that each address a different aspect of physical computation.

In section two, I will argue that there are (at least) three distinct questions at play in the literature on physical computation, the individuation question, the implementation question, and the interpretation question. These questions are sometimes acknowledged but are not distinguished as independent questions that require independent answers. In section three, I will define the connections between the questions, and I will provide a framework for how to fit the literature into it. Much of this project is revisionary because it will reclassify many accounts as answering different questions than initially intended. This part of the project will involve characterizing the

[15] There have been terminological shifts throughout the course of the literature on physical computation. As different and more detailed theories have evolved, so too has the terminology. Earliest accounts (such as Chalmers 1995) proposed a "theory of implementation," where later views offered a "theory of computation" that addressed the "implementation question," and later still, and most recently, theorists (many of the same who used the previous terminology) are now referring to their views a as theory of, or an account of the nature of "physical computation" or a "theory of physical computation." It is a mistake to view these terms as meaning one and the same thing, but there is a tight relationship between the theoretical evolution and terminological evolution of various views such that, at a very coarse-grained perspective on the literature, we can understand them as roughly meaning the same thing, or roughly aiming to answer the same question.

questions people take themselves to be answering and then recategorizing them into the questions I think they actually are answering.

2 Three questions

2.1 The implementation question

Theories of computation are sometimes taken as addressing the *implementation question*. The implementation question asks, "What are the conditions under which a physical system implements a given computation?" (Chalmers 1995). Importantly Chalmers presents this question in the context of cognitive science: "To clarify the notion of computation and its role in cognitive science, we need an account of implementation, the nexus between abstract computations and physical systems" (p. 391). But many answers to this question are given in a more general context—in the context of understanding physical computation *simpliciter* and not just in cognitive science.

However, the formulation of the implementation question is ambiguous between two readings if we do not further specify what the terms refer to. This ambiguity generates two different understandings of what an answer to the implementation question is meant to address. "A computation" can be understood in terms of 1) a computational structure or object or 2) a specific computational description of some physical process. How you interpret the question will determine the kind of theory you give. This is the first way in which answers to the "implementation question" can take different forms. I argue that because we have failed to disambiguate this question, the various forms the answers take end up being answers to different questions rather than different answers to the same question. This means that one way of disambiguating the question actually generates an entirely different question- one that was never meant to be addressed by an answer to the implementation question in the first place.

I will now argue that the first formulation of the question (option 1) stays true to the original intention behind the implementation question. In contrast, the second formulation (option 2) diverges and generates a different question altogether, what I call *the interpretation question*. To show this, we must engage Chalmers' classic 1995 paper, 'On Implementing a Computation.' In this paper, Chalmers says the following:

> The crucial element of such a bridge is a theory of implementation. Implementation is the relation that holds between an abstract computational object (a computation for short) and a physical system, such that we can say that in some sense the system "realizes" the computation, and that the computation "describes" the system. The key question that such a theory needs to answer is what are the conditions under which a physical system implements a given computation? Without an answer to this question, the foundational role of computation in cognitive science cannot be justified (p. 391).

The key to disambiguating the implementation question is to pay attention to how Chalmers originally formulated it in this passage. In this passage, he is explicit that the implementation question addresses the relation between an abstract computational *object* and a physical system. Here he shortens "computational object" to "a computation." The ambiguity arises when the question is taken out of its embedded context. As the "implementation question" has been passed around, stated, and restated, this context has dropped off, leading some to interpret the question in the second way.[16] Further evidence can be gleaned from the following passage:

> It will be noted that nothing in my account of computation and implementation invokes any semantic considerations, such as the representational content of internal states. This is precisely as it should be: computations are specified syntactically, not semantically. Although it may very well be the case that any implementations of a given computation share some kind of semantic content, this should be a consequence of an account of computation and implementation, rather than built into the definition (p. 399).

Computational *objects* are specified syntactically. This is not to deny that computational objects have semantic interpretation, just that a theory that tells how a computational object is implemented does not need to say anything about the semantic interpretation of the computational object. It may turn out that different computational objects have the same semantic interpretation or that a single computational object has several semantic interpretations. But whether it does or doesn't is a *different* question. As Chalmers says: "an account of implementation is only half the story" (p. 400). The implementation question is a *bridge*—the "nexus" between the abstract domain of computation and the concrete domain, including the domain of cognitive systems. Additional questions about the relationship between computation and cognition remain open,

[16] At the 2023 Pacific APA, I asked Chalmers what he meant when formulating the question, and he confirmed that he meant the first formulation. He also agreed that even though some people take themselves to be answering "the implementation question," they are doing something more like interpretation. He read my writing sample (as part of the Bersoff Fellowship application) that draws this distinction and told me directly that he thinks I've gotten this correct. This doesn't entail that *everyone* has to agree that I've gotten this entire project correct. Still, it at least confirms that my interpretation of Chalmers' 1995 paper and my reading of his formulation of the implementation question is correct and should be trusted.

including how we should interpret the computational objects that the brain implements. Or more generally, how we should interpret any computation system, not just the brain.

So, how should we understand computational *objects* when it comes to the implementation question? Chalmers specifies some familiar computational objects such as Turing machines, cellular automata, Pascal programs, cellular automata, and more. This is no surprise when we consider the topics presented in the previous chapter. Both functionalism and computationalism described the organization of the mind this way—as having an organization that takes the form of a Turing machine or Turing-style machine. This is a type of computational *structure*. Thus, when Chalmers gives his answer to the implementation question (which I will specify in more detail in Chapter 5), he says the following:

> The relation between an implemented computation and an implementing system is one of isomorphism between the **formal structure** of the former and the **causal structure** of the latter. In this way, we can see that as far as the theory of implementation is concerned, a computation is simply an abstract specification of causal organization (p. 396).

Thus, I formulate the implementation question in the following way:

> **The Implementation Question**: What are the conditions under which a physical system implements a computational structure?

If an account of physical computation gives us an answer that relates the formal computational structure to the causal structure of the physical system, then we can say that it answers 'the implementation question' and provides a *theory of implementation*. If it does not do this, we should say that it answers another question—that it provides some other kind of theory.

2.2 The interpretation question

Depending on your theoretical starting place or your ultimate explanatory goals, you may interpret "a given computation" as meaning some *specific computational description*. This means that the question becomes one about identifying which computation (possibly understood as a computational *process)* the physical system performs (or could perform). This leads to some people asking, "Which computation does this system implement?" This question is a natural extension of the second reading of the implementation question. But using the term 'implement' when asking questions about interpretation should be avoided until implementation and interpretation are clearly distinguished in the literature. Without a clear distinction, asking the interpretation question in the above way further ambiguates what is meant by the implementation question. Thus, I formulate the interpretation question in the following way:

> **The Interpretation Question**: What computational process does the physical system perform?

The interpretation question asks to identify the computational process that the physical system performs. It might also ask for an interpretation of the computational states of the physical computing system. Either way, this is a question about *what* the system computes in terms of states or processes. Nothing hinges on how we apply the interpretation for my purposes. We might ask the interpretation question because we want to know whether some physical computing system is computing addition or multiplication. Or we might ask if it performs the 'and' function or the 'or' function. When it comes to the brain, we might wonder whether it's performing one function or another—we might want a theory that tells us which function we should attribute to Marr's computational level. My claim is that theories that answer questions about how to interpret the states or identify the computational processes of a computing system are answering 'the interpretation question,' and they provide a *theory of interpretation*.

This is our first indication that a theory of implementation and a theory of interpretation are actually *compatible*. We can see how these answers *complement* each other by addressing two aspects of physical computation.

2.3 The individuation question

The implementation question is thought to be closely related to another question and is sometimes used interchangeably with it. I call this question the "individuation question." The individuation question asks for a way to distinguish computing systems from non-computing systems and originates from the work of Gualtiero Piccinini and Oron Shagrir. Some engaged in the debate about physical computation have adopted this question and take themselves to be providing an answer to it. To further complicate matters, some (like Mark Sprevak) have associated this question with the implementation question. This further confuses the questions that a theory of physical computation is meant to address. Now, I will argue that we should distinguish the individuation question from *both* a theory of implementation and a theory of individuation.

The first reason we should distinguish the individuation question from the interpretation question is that it is only when we take *all* views to answer this question that we end up with the result that theories of implementation and theories of interpretation are incompatible. If they are all seen as answering the same question, then it is a mistake to think they go together. But, when we distinguish the individuation question from the other two questions, we can have disagreeing answers to the individuation question while maintaining the compatibility of implementation and interpretation views.

An answer to the individuation question generally takes two forms.[17] One answer involves a reference to semantic content. This answer ascribes to the idea that there is "no computation without representation"—that computation is necessarily a semantic process. So, any theory of computation should refer to the semantic content of the computing system. When we take all "theories of computation" to address the individuation question, it becomes clear why some accounts that bring in a semantic interpretation are thought of as alternatives to those that don't. But, as I have argued, theories that involve semantic interpretation are answering the *interpretation* question. Another way to say this is that a theory of interpretation may *stem from* an answer to the individuation question, but it is not, in itself, an answer to the individuation question. I argue that we should understand an answer to the individuation question as stating a metaphysical *commitment* about the nature of physical computation. I formulate the individuation question in the following way:

[17] It sometimes takes a third form that involves individuating computational systems mechanistically. I set this view aside and address it in chapter six.

The Individuation Question: How are computing systems distinguished from non-computing systems?

The other answer to the individuation question says that we can individuate computing systems from non-computing systems *without* referencing their semantic properties. This is not to *deny* that computational systems have representational properties. Instead, it is a claim that we do not need them to give a theory of computation. But again, notice how this relies on bringing individuation and implementation together. If you are inclined to give a theory of implementation and understand it in terms of implementing a computational *structure* (like Chalmers), then it is no surprise that you answer the individuation question this way. But when we account for the different questions, an answer to the individuation question does not necessarily have to be the *denial* that representational properties play a role in computation. You might answer the individuation question this way while thinking that computation requires semantic properties—you just don't think they need to factor into a theory of implementation! This is a further reason why we should eliminate the individuation question as the target of a theory of computation. Moreover, it is not clear that anyone giving a theory of implementation takes themselves to be answering *this* question.

Because a theory of computation is sometimes, by some people, framed as answering the individuation question, the false dichotomy between theories of implementation and theories of interpretation persists in the literature. Often this is framed in terms of a debate between non-semantic accounts and semantic accounts of physical computation. So, theories that rely on semantic interpretation are often seen as alternative theories of computation to the ones that focus directly on specifying a non-semantic implementation relation. However, once we distinguish the implementation question from the interpretation and individuation questions, we can see that the views are *compatible* because they answer *different* questions that each address different aspects of physical computation. This also means that we need to eliminate the individuation question as the *target* of a theory of computation.

3 What does a theory of computation target?

So far, I've defined three distinct questions:

i. **The Implementation Question**: What are the conditions under which a physical system implements a computational structure?

ii. **The Interpretation Question**: Which computational process(es) does the physical system perform?

iii. **The Individuation Question:** How are computing systems distinguished from non-computing systems?

Given these three questions, we can now ask: what is the target of a theory of computation—which of these questions does a "theory of computation answer?" My answer is that a theory of computation should address (i) and (ii), with the option of answering (iii). But an answer to (i) or (ii) does not have to hinge on how (iii) is answered.

But those engaged in the literature do not take themselves to be answering *all* of these questions. This is fine because I don't think anyone addressing physical computation must do so. Instead, we can use a divide-and-conquer strategy. Some people can answer the implementation question, and some can answer the interpretation question. They can also adopt whatever answer they want to the individuation question (or even leave it out altogether). My position is that a "theory of computation" will involve bringing these aspects *together* into a unified account.

Many of the people engaged in the debate also do not take themselves to be answering either the implementation or interpretation questions in the ways that I have formulated them. As I have argued, the implementation question has an ambiguous reading when it is removed from its original context. But, this is prior to disambiguating the different readings of the implementation question. After disambiguating the question, we are in a position to reclassify the views.

Given the above discussion, theorists ought to be taking themselves to be answering either the implementation question or the interpretation question, as I have defined them. This means that some people who think they are answering the implementation question should take themselves to be answering the interpretation question. This also means that some people who take themselves to be providing an answer to the individuation question should be taking themselves to be giving an answer plus either a theory of implementation or a theory of interpretation, depending on how their theory is developed. As it turns out, the literature is already set up quite nicely to adopt this framework, which I develop in chapters 5 and 6. Still, I will briefly introduce it before making a pitstop with triviality arguments that have also influenced how the literature has been developed.

Non-semantic accounts that address the nature of physical computation are always given in terms of the implementation relation according to the first reading I specify in section 2.1. This means that non-semantic accounts actually *do* provide a theory of implementation. Thus, we *should* take non-semantic accounts to be addressing the implementation question. Because this aligns with how non-semantic theorists would categorize their views, so only a little revision is required here. The revision, then, is that those who offer a non-semantic account should take their account to be *compatible* with a theory of interpretation which requires recognizing the distinction that I have drawn and accepting that they are only addressing one aspect of physical computation.

Semantic accounts, on the other hand, often take themselves to be answering either the implementation question, the interpretation question, or possibly both. This is possibly because they take on the second reading of the implementation question. I imagine that other reasons can be added to my diagnosis for why semantic accounts are developed as a response to the implementation question or why the second reading of the implementation question was adopted in the first place. Still, nothing about my framework hinges on *why* authors come to give the theory that they do. I've made a charitable diagnosis for why this might have occurred in the literature, but I don't see why any other reason would be a barrier to the success of my framework. Those who give a semantic account can maintain their answer to the individuation question, but according to my framework, they should take themselves to be answering the interpretation question and not the implementation question. This also means that, like the non-semantic theorist, they would need to accept that their view is compatible with a non-semantic theory of implementation.

In the following chapters, I will show how to place non-semantic and semantic theories into my framework. But first, there is an additional reason why adopting my distinctions benefits the literature on physical computation. It turns out that triviality arguments can be mapped onto both implementation and interpretation. In other words, we can identify a triviality of the implementation relation and a triviality of the interpretation conditions. And, even more, a non-semantic account has the theoretical tools to address trivial implementations, while a theory of interpretation has the theoretical tools to address trivial interpretations, but no single approach can handle both. I will show this in the next chapter.

4 Implementation, interpretation, and triviality

1 Introduction

In the previous chapter, I argued that there are three questions at play in the computation literature: implementation, interpretation, and individuation. I argued that we should understand the implementation question as addressing the relation between formal computation and physical systems, a connection between abstract computational objects and physical systems. I argued that theories that address how to interpret computational states or processes are not answering the implementation question. Instead, they are answering what I call the interpretation question. I also identified a third question that is sometimes described as the target of a theory of computation, the individuation question. I argued that this question could be answered independently as a metaphysical claim about the nature of physical computation and that it could be addressed separately from a theory of implementation and interpretation. I diagnosed the literature as suffering from two readings of Chalmers' original formulation of the implementation question, which is what leads many theorists to propose what *seems* to be alternative ways to respond to the implementation question when really, they are addressing different aspects of physical computation based on their understanding of the original question.

In this chapter, I will argue that there is an additional thing that influences how theories regarding physical computation are developed. Some of the first views that addressed the implementation relation were challenged by triviality arguments that aimed to show that the conditions defined in the relation allow for cases where non-paradigmatic computing systems could be shown to implement a computational structure. As these triviality arguments were being developed, a second way of trivializing physical computation came about. This second way involved superfluous *interpretations*. Not only was it possible to show that non-computing systems could meet the implementation relation, but it would also generate an additional problem; it would *also* be indeterminate which computation the system was performing. As I pointed out in the last chapter, the same computational structure can be interpreted in many ways. In this chapter, I will argue that a divergence between theories that address the implementation question and theories that address the interpretation question can also be traced back to which version of triviality the theorist aims to resolve. I will argue that those who aim to resolve the first type end up giving a theory of implementation, while those who aim to resolve the second type end up giving a theory of interpretation.

Because I developed the implementation and interpretation questions in the previous chapter, I will use them freely moving forward. When I mention one of these questions by name, I am using the in accordance with how I've defined them. This also means that when I use the terms 'implementation' and 'interpretation,' I use them based on the definitions I have given. In section two, I will introduce triviality arguments. Part of this task includes describing some of the early mapping views that address the implementation relation. The next chapter develops these views in detail, so I will briefly introduce them with only enough detail to show how triviality arguments apply to them. Early in this section, I will identify two types of triviality and pair them with their corresponding target questions before moving on to more examples of triviality arguments that exhibit the same two features. In section three, I will show how some people interested in physical computation accept either one or the other type of triviality. This indicates that not everyone intends to address both types. This fits nicely within my framework, which identifies two kinds of triviality and maps to them the type of theory that is the best fit for solving the problem.

2 Triviality arguments

Triviality arguments can be raised against functionalism, computationalism, or a theory of physical computation. The earliest triviality arguments attacked functionalism directly. However, triviality arguments against functionalism often extend directly to a broader context, including any view that ascribes a computational description to a physical system. In this section, I will follow the progression of triviality arguments. This will require a brief introduction to early theories of physical computation.

2.1 The 'simple mapping account'

The earliest iteration of a theory of computation can be derived from Putnam (1967). Putnam argued that anything accurately described by a computational description is a computing system that implements that description. He gave his theory in terms of a Turing machine with a corresponding machine table. For Putnam, a machine table describes a machine if the device has internal states that correspond to the columns of the table. It obeys the machine table when there is a correspondence between the transitions of the read-write head of the Turing machine and the transitions of the states of the physical system that is being described. Putnam's early view is visually depicted by the following graphic where a physical system *S* performs a computation defined by description *C* just in case (i) there is a mapping from the states ascribed to *S* by a physical description to the states *C*, such that (ii) the state transitions between the physical states:

$$\begin{tikzcd}
S_1 \arrow[r] \arrow[d, dash, dashed] & S_2 \arrow[d, dash, dashed] \\
C_1 \arrow[r] & C_2
\end{tikzcd}$$

Figure 4. Simple Mapping View Diagram

As explained in Chapter 2, this account is part of the development of what we now know as machine-functionalism. Because the view is characterized in terms of a specific type of machine, *a computing machine*, this view ends up being one of the first recognized implementation accounts that we now know as the "simple mapping account" (Godfrey-Smith, 2009). It is dubbed "simple" because it relies on a straightforward mapping understood in terms of a mirroring relation between the physical state transitions and the description of the abstract, computational structure (the transitions described by the machine table). We can formalize the simple mapping account in the following way:

> A physical system X implements a formal computation Y if there is a mapping f that maps physical states of X to abstract states of the formal computation Y, such that: for every step-wise evolution $S \rightarrow S'$. of the formalism Y, the following conditional holds: if X is in physical state s where $f(s) = S$, then X will enter physical state s' such that $f(s') = S'$ (Sprevak 2018).

The first triviality arguments challenged this idea. Thus, the first triviality arguments attacked a theory of implementation (as I define it) specifically. In the following sections, I will give a taxonomy of some of the most popular triviality arguments, and I will show that while they were initially generated to attack a theory of implementation, they led to the development of a second problem that I diagnose as contributing to the generation of theories that address the interpretation conditions.

2.2 Hinckfuss's Pail

Attributed to Ian Hinckfuss, Lycan (1981) gives us the following triviality argument that I call "PAIL":

> Suppose a transparent plastic pail of spring water is sitting in the sun. At the micro level, a vast seething complexity of things are going on: convection currents, frantic breeding of bacteria and other minuscule life forms, and so on. These things in turn require even more frantic activity at the molecular level to sustain them. Now is all this activity not complex enough that, simply by chance, it might realize a human program for a brief period (given suitable correlations between certain micro-events and the requisite input-, output-, and state-symbols of the program)? And if so, must the functionalist not conclude that the water in the pail briefly constitutes the body of a conscious being, and has thoughts and feelings and so on? Indeed, virtually any physical object under any conditions has enough activity going on within it at the molecular level that, if Hinckfuss is right about the pail of water, the functionalist quickly slips into a panpsychism that does seem obviously absurd . . . (Lycan 1981, p. 39).

PAIL does not mention triviality, and it is only later identified as a triviality argument against functionalism and then a theory of physical computation. However, because it gives a case where the simple mapping relation supposedly holds, we end up with a triviality result that targets the implementation relation specified by the simple mapping account. According to PAIL, a bucket of water sitting in the sun has so much causal complexity that, via the suitable categorization of states, it can be taken to realize the functional organization of a human agent. This undesirable result ends up trivializing the simple mapping account.[18]

This makes trouble for a theory of physical computation because if we translate the functional description in terms of computational states, then it turns out that the bucket of water has the appropriate *computational* structure. This is a problem for a theory of implementation because it attributes a computational structure to physical systems that are not computational. This type of result trivializes the very notion of computational implementation and calls for theory revision.[19]

2.3 Putnam's inputless FSA

Putnam (1988) argues against his mapping view and concludes that it is possible to attribute a computational structure to a non-computing system and that those systems might be said to perform *any* computation at one time or *every* computation at one time. This second problem is a problem that was not present in PAIL. Putnam argues that any machine table can describe any system and that the description will always be correct. He gives us the following triviality argument that I call "FSA:"

[18] Some disagree that PAIL gives a triviality result or that even if it does that it's not unacceptable. For example, Sprevak (2018) argues that what follows with the argument is a version of panpsychism that is not an untenable position. His main point, though, is that it's not clear that PAIL generates a triviality result. He thinks that what is imagined is the *possibility* of an unusual implementation and that we need a reason to think that such an implementation is *actual* in order to have a triviality result. Absent a reason, we should not think that computational implementation is trivial in this sense (p. 7).

[19] Piccinini (2015) and Piccinini & Maley (2021) call this "limited pancomputationalism"—the idea that every physical system performs at least one computation.

> Imagine a physical system with defined boundaries, no inputs, and no outputs, and that has state transitions that take place in real time intervals-- say in a 7-minute interval from 12:00pm - 12:07pm. We can imagine such a system exists because of the laws of physics: the principle of continuity and the principle of noncyclical behavior (guarantees the disjointness of the physical states). Any machine table with any sequence of states can be paired up with the physical system where the system realizes the given machine table during the interval specified (Putnam 1988, p. 122).

The trivial implementation comes from the implementation of the FSA structure. This is the same type of triviality we glean from PAIL. The additional type of triviality comes from Putnam's inclusion of the idea that "any machine table" can be paired with the physical system. The same FSA can have many different corresponding machine tables that specify a number of different processes. So, the problem is that any of these machine tables will be compatible with the implementation. When we interpret the states of the machine table, we learn what the FSA *does*—which computation it is performing.[20] Because it is indeterminate which computation the system is performing based on the implementation alone, the interpretation is trivialized—the system could be understood as performing any or all computations unless we specify an interpretation of the machine table.[21]

2.4 Two types of triviality

[20] See sections 1.2 and 2.4 of Chapter 1.

[21] Piccinini (2015) and Piccinini & Maley (2021) call this "unlimited pancomputationalism" – the idea that every physical system performs every computation.

FSA introduces an additional type of triviality into the literature, a trivialization of the interpretation conditions. This gives us two types of triviality when it comes to physical computation, one regarding the implementation relation and one involving the interpretation of the computational states or processes:

Trivial implementation: Too many or all physical systems can be seen as implementing a computational structure.

Trivial interpretation: A physical system can be interpreted as performing every computational process.

The literature sometimes recognizes these two types of triviality. For example, the problem of trivial implementations is sometimes called "limited pancomputationalism," while trivial interpretation is sometimes called "unlimited pancomputationalism" (Piccinini, 2015 and Piccinini & Maley, 2021). I do not think that these labels do justice to how we should understand their place in the literature. I provide a new labeling scheme that maps each triviality type onto the aspect of physical computation in which they correspond. I also frame the problems in a way that aligns with my development of the implementation and interpretation questions. Once this is done, we can see how the triviality types neatly map onto the questions:

The Implementation Question: What are the conditions under which a physical system implements a computational structure?

Trivial implementation: Too many or all physical systems can be seen as implementing some computational structure.

The Interpretation Question: Which computational process(es) does the physical system perform?

Trivial interpretation: A physical system can be interpreted as performing every computational process.

These pairings show how a divide-and-conquer strategy could play out in a way that addresses both types of triviality. No theory of implementation or interpretation, on its own, will manage both at the same time. However, as I will explore at the end of this chapter, some people reject that one or the other type of triviality is a problem. Thus, they sometimes set one type aside and move on with providing their view that addresses the other. I want to be clear that I am not demanding that everyone take both types of triviality seriously. There are genuine reasons people give for thinking that one (or maybe both) are not threatening. However, unless we all agree on which ones should stay or go, we should make room for them both in a theory of computation. My framework carves out a place for both and a clear strategy for addressing wach of them. Those who provide a theory of implementation address trivial implementations, while those who provide a theory of interpretation address trivial interpretations.

In the following sub-sections of this chapter, I will present several more triviality arguments cited in the literature to highlight additional places where the two types of triviality also emerge.

2.5 Searle's wall

Before I get into Searle's (1990, 1992) triviality argument, some explanation is required. Searle argues against the claim that "the mind is a program and the brain the hardware of a computational system" (p. 21). His starting assumption is that the mind is *not* a computer program. He then argues against the claim that the brain is a digital computer on the basis of the idea that all there is to a computer is syntax and that the mind requires semantic understanding. So, given this, the brain cannot be a digital computer. In Searle's triviality argument, he introduces the concept of a "program." When he discusses a program and implementation, he says that a computer "implements the steps of a program." He defines the program in terms of the instructions provided to the Turing machine, which specifies the movement of the read-write head. In other words, the program is the transitions specified by the machine table. He argues that anything can count as a following a program if following a machine table's steps is all there is to computation. Thus, we have his triviality argument that I call "WALL:"

> For any program there is some sufficiently complex object such that there is some description of the object under which it is implementing the program. Thus, for example the wall behind my back is right now implementing the WordStar program, because there is some pattern of molecule movements which is isomorphic with the formal structure of WordStar. But if the wall is implementing WordStar, then if it is a big enough wall, it is implementing any program, including any program implemented in the brain (Searle 1990, p. 27).

When introducing the WordStar program, he assigns a semantic interpretation to the machine table he is referencing. This is where the second type of triviality comes into his argument. Inside the wall are many microscopic physical changes—so many changes that the physical activity inside his wall has at least one structurally isomorphic pattern to a machine table—one that is sufficiently complex enough that it could be responsible for generating WordStar. This brings structure (the machine table) together with an interpretation (WordStar). Moreover, according to Searle, if the wall is complex enough, it could implement any program, not just WordStar. In the WALL argument, a trivialization of the implementation relation and a trivialization of the interpretation conditions are present.

2.6 Putnam's rock

A common feature of PAIL and WALL is that we are not given a way to understand the relevant mappings—we are just told that there is one. Putnam (1988), on the other hand, provides a method for finding the relevant mapping with "ROCK" (as described by Sprevak 2018).

> Pick any open physical system (say, a rock) and any time interval, t_0 to t_n. Consider the 'phase space' of the rock over this time interval. The phase space is a representation of every one of the rock's physical parameters, including the physical parameters of the rock's constituent atoms, molecules, and other microscopic parts. Over time, the rock will trace a path through its phase space as its physical parameters change. The rock's physical parameters will change owing to endogenous physical causes (its atoms changing state, vibrations, atomic decay, etc.), and because of external causal influences (gravitational, electromagnetic, vibrational, etc.). Putnam argues that some external influences are certain to play the role of 'clocks' for the rock: due to these influences the rock will not return to precisely the same set of values of its physical parameters in the time interval. Putnam calls

this the ‘Principle of Noncyclical Behavior’. He argues that this principle is likely to be true of any open physical system (p. 9-10).

The phase space reveals that the rock undergoes multiple changes in its physical state type during the time interval, including $r_1 \rightarrow r_2 \rightarrow r_3 \rightarrow r_4$. Additional changes include the following disjunctions: $r_1 \vee r_3 \rightarrow r_2 \vee r_4 \rightarrow r_1 \vee r_3 \rightarrow r_2 \vee r_4$. This means that the rock travels through regions of phase space while oscillating between disjointed regions of phase space. Putnam maps $r_1 \vee r_3$ to computational state *A* and $r_2 \vee r_4$ to computational state *B* giving an isomorphism between the physical states and transitions of the rock and the formal state transitions of the FSA. This same reasoning can be extended to other physical systems, and therefore, every open physical system implements every inputless FSA.

Putnam’s ROCK is an example of pairing a triviality argument with the specifics of a theory of implementation so that we can see how an argument that claims trivial implementation can be understood in terms of what a theory of implementation might specify. Because Putnam specifies his argument in terms of implementation only, then this argument only targets the implementation relation.

2.7 Chalmers’ clock and dial

Some have argued that we can avoid trivializing the implementation relation by strengthening the relations between the physical states of the implementing system. One way to do this is to argue that the physical state transitions should support certain counterfactuals. Nevertheless, Chalmers (1996) shows that a triviality argument can be given for even strengthened mapping views such as the counterfactual account. Chalmers gives us a case involving a clock and dial; call this "CLOCK." Chalmers describes a 'clock' as a component of a physical system that reliably transits through a sequence of physical states over a time interval. He describes a 'dial' as a physical component of the system with an arbitrary number of physical states such that if it is put into one of those states, it states in that state during the time interval. Chalmers claims that every physical system containing a clock and dial will implement every inputless FSA. Chalmers's argument is very similar to Putnam's. However, he characterizes an FSA that includes inputs and outputs and not just a possible trajectory over a phase space but an actual one.

> Label the physical states of the system $i - j$ where i corresponds to a clock state and j to a dial state. If the system starts in state [1, j], it will reliably transit through states [2, j], [3, j], and so on… Now, assume the physical system on the actual run has a dial state 1. The initial states of the system will be [1, 1]; we associate this state with an initial state of the FSA. We associate states [2, 1], [3, 1] and so on with the subsequent states of the FSA in the obvious way. If, at the end of this process, some FSA states have not come up, choose an unmanifested state *P*, and associate state [1, 2] with it. We associate states [2, 3], [3, 2] and so on with the states that follow *P* in the evolution of the FSA… eventually all of the states of the FSA will be exhausted. Now, for each abstract state of the FSA, we have a non-empty set of associated physical states {[i_1, j_1], [i_2, j_2] …, [i_n, j_n]}. Map the disjunction of these states to each FSA state. The resulting mapping between physical and formal states satisfies the counterfactually strengthened version (p. 318).

Chalmers argues that almost all open physical systems have a clock and dial, and if not, we can attach one to the physical system. But, if a trivial implementation can be achieved by adding a watch, the implementation account should be rejected. Notice that CLOCK also only targets implementation. This is no wonder since Chalmers keeps implementation and interpretation distinct.

2.8 Godfrey-Smith's transducer

Godfrey-Smith (2009) provides a triviality argument meant to build off of Chalmers' CLOCK. He argues that any sufficiently complex physical system can be made into a behavioral duplicate of an intelligent agent via a change to the "transducer layer of that system." He draws on early mapping views to develop his argument and concludes that functionalism combined with a simple mapping account collapses into triviality. He begins with the idea that any functionally characterized physical system whose operations link it with its environment can be broken down into a *transducer layer* and a *control system*. When it comes to inputs, the transducer layer responds to physical impacts in a form that the rest of the system can use for further processing (something like a perceptual system). When it comes to outputs, the transducer layer, like muscle fibers, allows it to respond to various inputs. The control system is used for everything functionally important to the system other than the transducer layer. Locating the transducer layer is a matter of locating the boundary between the system and its environment (p. 284-285). I call this argument "TRANSDUCER;"

> The transducer layer can be seen as the input-output devise. that maps one set of physical magnitudes to another, or many to one without constraint. And a change to the transducer

> layer may result in a change to the formal properties of the mapping as well as which physical magnitudes are involved. Now, consider an actual human agent, A, with non-marginal mental properties. If functionalism is true, then this agent has its mental properties in virtue of its functional organization. This functional organization is labeled S and is specified in the form of some combinatorial state automata (CSA). Then we take a complex physical system, B, that has interactions with its environment—B may be a pail of water such as in PAIL. There will be a possible transducer layer that can be added to B that will give it the input-output profile associates with S. B, with its added transducer layer would be a realization of S in the functional sense (p. 287).

This case generates the following argument:

> 1. For any sufficiently complex system B, there is a possible system that differs internally from B only in its transducer layer, and that has the input-output properties of a human agent with non-marginal mental properties.
> 2. Any sufficiently complex system with the input-output properties of a human agent is a functional duplicate of that agent. (Simple mapping criterion for realization.)
> 3. Functional duplicates share all their mental properties (functionalism).
> 4. Two systems that differ only in their transducer layers must either both have, or both lack, non-marginal mental properties.
> 5. Therefore, any sufficiently complex system has non-marginal mental properties (p. 288).

Godfrey-Smith targets functionalism, but we can extend the argument to a theory of physical computation. TRANSDUCER, like CLOCK, shows that any sufficiently complex system can be given a transducer layer, and if this is done, it can be shown to implement some CSA. Godfrey-Smith concludes that the mapping relation is too weak and that views of this kind collapse into triviality. TRANSDUCER, then, is an argument that trivializes the implementation relation.

In this section, I have shown how some arguments target implementation, interpretation, or both. One reason someone might target only one and not the other is that they may find only one type threatening.

3 Accepting some triviality

Some people take the stance that one or possibly both types of triviality are permissible. This is also a motivating factor for why someone might be satisfied with addressing either the implementation or interpretation questions and not both. For example, suppose you think that triviality when it comes to an interpretation of the computational system is permissible. In that case, we can address physical computation by focusing solely on the implementation relation. On the other hand, if you think there is nothing concerning about the trivial implementations, then you might think we can address physical computation by focusing on interpretation conditions. However, whether you think only one type of triviality matters should not affect whether we need a theory of computation that addresses *both* implementation and interpretation. It is possible to deny that a triviality argument threatens the adequacy of either a theory of implementation or a theory of interpretation while accepting that both implementation and interpretation are essential aspects of understanding computation in physical systems. After all, a metaphysical account is meant to explain the *nature* of some phenomenon, not just the aspects of that phenomenon that we care about. In this section, I will show how some people are okay with triviality in some respects. This might be understood as motivation for focusing on either a theory of implementation or a theory of interpretation.

Chrisley (1995) argues that instances of Putnam-style triviality may occur in a reduced sense. Given the natural complexity of the physical world, there is much room for indeterminacy. So, even if every system does not instantiate every automaton, it might be that every ordinary macroscopic system (like the brain) instantiates an infinite number of automata. This is a reduced Putnam-style argument because only *some* systems will have multiple interpretations instead of *every* system being interpreted as performing every computational process. This is an acceptance that some systems might be given multiple interpretations. Chrisley is specifically interested in the sufficiency of the computational theory of mind when he argues that we can accept some triviality—he says that "for computational states to be ontologically sound, one does not have to show that there is only *one*, *unique* computational characterization that applies to a given system" (p. 417). In other words, it is possible that the same computational system (perhaps the brain) can be understood as performing many different computational processes.

Rescorla (2013) argues that trivial interpretations are harmless because, in some cases, facts about a system's physical properties and the surrounding computational practice may not fully determine whether the system performs a particular computation. He draws an analogy with language to support his point. He argues that the term 'bank' can mean either a *financial institution* or a *river* bank and that whether it means one or the other is fixed by the speaker's intentions and expectations

(p. 702). Similarly, the context will fix the computation that the system performs. In other words, facts about computational interpretations will depend on the context in which the computation is being performed. In this way, Rescorla accepts some cases of trivial interpretations.

Schweizer (2016) argues that trivial implementations are not something that we should necessarily avoid. He thinks that trivial implementations pair with anti-realism about physical computation and whether any number of abstract mappings exists across various systems in a *purely mathematical sense* is not a problem. He contrasts this with the idea that there is any number of mappings from the set of positive integers to collections of physical objects and particles, but whether we care about the mapping is a matter of context. So, whether we count the mapping as implementing a computation will depend on the context in which the mapping occurs. This is an explicit acceptance of trivial implementations on the basis of the idea that we can resolve triviality within whatever context we are applying the description.

Dewhurst (2018) argues that we should accept that many (or any) physical system can be understood as performing some computation. He thinks that we can avoid trivial implementations because when we adopt, what he calls *an explanatory perspective*, the range of functional attributions (including computation) will be constrained by the underlying physical structure of the system we are interested in. But he argues that this permits each system to be seen as performing multiple computations simultaneously. He thinks that the explanatory perspective will help pick out which computation is being performed, but nonetheless, it is possible to view the system as susceptible to trivial interpretations.

In this section, I have shown that not everyone is moved by both types of triviality. I gave examples of some people who accept trivial implementations or interpretations as permissible. This is compatible with my framework. My framework does not require that everyone address both types of triviality. But I have shown that there are two types that map cleanly onto the questions I identify. Thus, two types of triviality concerns map onto the two aspects of physical computation I identify.

5 Non-semantic theories of implementation

1 Introduction

In this chapter, I will present what are sometimes categorized as "non-semantic" theories of computation. Non-semantic accounts address physical computation without specifying how to interpret the computational system semantically. I classify non-semantic accounts as theories of implementation because they all, in different ways, address the implementation relation. Most people who give a non-semantic account characterize themselves as providing a theory of implementation, a theory of computation, or both. Because I reserve 'theory of computation' for a unified account that addresses both implementation and interpretation (at least), I classify non-semantic accounts in terms of a theory of implementation only, as on their own, they are not giving a theory of computation. I am *not* arguing for the claim that to answer the implementation question that you *must* provide a non-semantic account. It is an artifact of the literature that all theories that address the implementation relation happen to be non-semantic.[22] Not all non-semantic accounts answer (or recognize) the individuation question. This is fine because I argued that we could take or leave that question; it is not an essential part of a theory of computation.

[22] This chapter categorizes accounts that are already developed, so it might be possible to give a semantic account of implementation in the future. But I am suspicious of a project that claims to do this. As we will see in the next chapter, many accounts addressing interpretation are characterized as theories of implementation. One way someone might go about this is by bringing in an internal semantics and then calling their view semantic in virtue of this move. However, the only way that I think this can be pulled off without causing additional problems for the literature is to adopt a framework like mine first. This is because semantics accounts are semantic in virtue of referencing an *external* semantics. All non-semantic accounts accept that computation requires an internal semantics. So, explicitly referencing it doesn't make the view semantic in the way that semantic views are semantic. I say that we would need to adopt a framework like mine because it gets rid of the "semantic" versus "non-semantic" debate and re-classifies the views as either a theory of implementation or interpretation. Once we adopt a framework like mine, we could reintroduce semantic and non-semantic in a more nuanced way. So, a non-semantic implementation account tacitly accepts an internal semantics while a semantic implementation account explicitly it, for example.

There are two argumentative components to this chapter. The first is that we have good reason to distinguish between the implementation and interpretation questions. This benefits the literature because it clearly shows how views often thought of as alternatives are compatible. This means that the literature should not be understood in terms of non-semantic versus semantic accounts. Instead, we should understand different accounts as addressing either implementation or interpretation and that these views are *complimentary* because they each address a different aspect of physical computation. This is not adequately addressed in the literature and is often entirely obscured by the liberal use of the term "implementation" across theories. The second component is revisionary. Adopting my framework requires accepting a re-structurization of the literature in terms of implementation and interpretation rather than semantic and non-semantic. This means that people will need to accept that they were not answering the question that they initially thought they were. Another revisionary component is that no single account can address both types of triviality. Some people who incorporate an answer to the individuation question into their theory think that doing so helps to ward off trivial implementations. For example, suppose physical computation is necessarily a semantic process. In that case, this rules out trivial implementations such as pails of water, walls, etc., because those are not systems that semantically represent their environment. However, because I distinguish an answer to the individuation question from a theory that addresses physical computation and then pair triviality results with either implementation or interpretation, an answer to the individuation question can no longer serve as a bonafide response to triviality.

In section two, I will introduce a variety of non-semantic accounts. I will maintain that these views address the implementation relation and therefore count as a theory of implementation. I will also point out when these views are meant to address trivialization. Taxonomizing the views will be primarily a descriptive project because people who give these views take themselves to be giving a theory of implementation as I define it. The revisionary part is that I will argue that these views only address implementation and are only equipped to handle trivial implementations meaning they only address *one aspect* of a theory of physical computation. I pair implementation theories with the problem of trivial implementations in the following way:

The Implementation Question: What are the conditions under which a physical system implements a computational structure?

Trivial implementation: Too many or all physical systems can be seen as implementing some computational structure.

2 From a non-semantic theory of computation to a "theory of implementation"

As we saw in Chapter Three, what is now known as the 'simple mapping' account is one of the earliest and most influential theories of implementation (Putnam 1960, 1967, 1975). According to the simple mapping account, a physical system *S* performs a computation defined by description *C* just in case (i) there is a mapping from the states ascribed to *S* by a physical description to the states defined by computational description *C*, such that (ii) the state transitions between the physical states mirror the state transitions between the computational states (Piccinini & Maley 2021). To see how the view works, suppose the physical system *S* has physical states $p_1 - p_2$ that map onto states $s_1 - s_2$ of the computational description C for any computational state transitions of the form $s_1 \rightarrow s_2$, S transitions from p_1 to p_2 whenever it goes into state s_1.

The simple mapping account is a theory of implementation that addresses the implementation question because it draws a relation between a formal computational structure and the states of a physical system. The physical system implements a computational structure if the specified conditions are met. Putnam initially gives his account in terms of Turing machines, meaning if the conditions were met, the physical system would implement a Turing machine. But eventually, Putnam abandoned the Turing machine requirement and specified the view in terms of a physical description of the system. This means that the view can be generalized to computational structures other than a Turing machine.[23]

The simple mapping account faced immediate challenges because acceptable mappings under this specification are quite easy to find in the world. As we saw in the previous chapter, PAIL initially demonstrated how liberal the simple mapping account is. While Putnam's view was never meant to address *trivial* implementations (instead, it seems to generate the concern), it is still a theory of implementation.

2.1 Specifying causation

A common way to address the trivial implementations permitted by a simple mapping is to provide constraints on the mapping relation. Some do this by introducing causal requirements into the theory. There are different ways to understand causation. In this section, I will address the different

[23] Piccinini (2015) challenges this move, but we can let it slide.

ways that people have defined causal relations in order to prevent trivial implementations. Adding causal requirements begins with Putnam's (1988) expansion of the simple mapping view.

Putnam specifies a physical-causal structure that must obtain for a physical system to implement a computational structure. This is an improvement on his original mapping account (that he ends up rejecting anyway). In order to strengthen the simple mapping account, Putnam argues that the state transitions in the physical system must "stand in the appropriate causal relations to one another and to the inputs and the outputs" (p. 124). The appropriate causal relation, in this case, has to do with the "type that commonly obtains in mathematical physics… that is, if a program says, as it might be, that 'state A is always followed by state B'… in classical physics this would be determined by either the Hamiltonian or the Lagrangian equations of the system… in familiar language, this is to say that a mathematically omniscient being of the kind once envisioned by Laplace… could *predict* that the system *X* would go into state B at the relevant time given the information that it was in state A at the earlier time given the boundary conditions" (p. 96).

By adding this causal requirement, the states will be maximally compatible with each other in a way that demands that they progress in the order they do. Putnam's strengthened account still addresses the implementation relation, but now it also addresses trivial implementations by strengthening the causal relations between the physical states in the system.

2.2 Causation and counterfactuals

Chalmers (1995, 1996) argues for a theory of implementation that builds on Putnam's causal requirement. Putnam ultimately rejects his strengthened mapping view when he argues for ROCK.

Chalmers thinks the improved mapping view is still susceptible to trivial implementations because the specified physical system does not satisfy the right state-transition conditionals. Chalmers argues for the following causal relation:

> The conditionals involved in the definition of implementation are not ordinary material conditionals, saying that on all those occasions in which the system happens to be in state *p* in the given time period, state *q* follows. Rather, these conditionals have modal force, and in particular are required to support counterfactuals: *if* the system were to be in state *p*, then it would transit into state q. expresses the requirement that the connection between connected states must be reliable or lawful, and not simply a matter of happenstance. It is required that however the system comes to be in state *p*, it transits into state *q* (perhaps with some restriction ruling out extraordinary environmental circumstances; perhaps not). We can call this sort of conditional a strong conditional (Chalmers, 1996, p. 312-313).

Chalmers concludes that it is not clear from Putnam's description that his conditionals have this sort of modal force. In any case, Chalmers thinks that Putnam's view does not satisfy the strong conditionals in the way that implementation requires. In addition, Chalmers does not think that Putnam includes all of the necessary state transitions in the mapping. As a solution, Chalmers specifies his counterfactual account in the following way:

> A physical system implements an inputless FSA in a given time-period if there is a mapping *f* from physical states of the system onto formal states of the FSA such that: for every

formal state-transition $P \to Q$ in the specification of the FSA, if the physical system is in a state p such that $f(p) = P$, this causes it to transit into a state q such that $f(q) = Q$ (p. 315).

Chalmers gives a strengthened mapping account that requires counterfactual supporting transitions and the requirement that the unmanifested state transitions are reflected in the physical structure of the machine. This account specifies the implementation relation and addresses trivial implementations.

Chrisley briefly discusses a consequence of causal views that is relevant to this dissertation's project and worth noting here. In his discussion regarding Searle's triviality argument (WALL), he says that "if one adopts a causal notion of computation, then every system will *not* realize every computation, but every system *will* realize multiple (perhaps infinitely many) computations simultaneously" (Chrisley 1996, p. 404). I interpret Chrisley as saying that a theory of implementation can only address one type of triviality: trivial implementations. This leaves the problem of trivial interpretations on the table, which is why we must have another way of addressing the problem, namely, a theory of interpretation. As I have said before, some are concerned with a trivialization of the interpretation conditions, but other are not. Chrisley is in the latter camp. He thinks that cognitive science does not require a unique computational description for a system. Thus, the possibility of ascribing multiple computational processes to the same system is not of concern to him.

Like Chalmers, Chrisley (1995) argues that Putnam's notion of causation is too liberal. To strengthen the relation, he requires proper embedding conditions and a stronger specification of causation. Chrisley considers embedding conditions in terms of the signals received and produced via various inputs and outputs—how the system *interacts* with its environment. For computers, this might include a mouse, keyboard, monitor, video camera, etc. (p. 414). He then argues that to be true to computation, the inputs and outputs of the system must be characterized in terms of their intrinsic properties. This requires us to fix the definition of an input and output to a system before drawing the mapping relation. For example, a digital desktop computer might have as its output a video display. Notice that output is understood in terms of an architectural feature of the system, not an interpreted state or process. Chrisley thinks that if the outputs are fixed, it eliminates systems like walls being shown to implement computations because they cannot produce the required output—they do not have a video display.

Chrisley also specifies something called “full causation.” He thinks Putnam's notion of causation is much too liberal in that it allows events outside the context of mathematical physics that we would *not* consider causally related (p. 414). Chrisley mainly provides an argument against Putnam’s use of causation rather than providing an original account of causation, so I cannot give it here. However, he does say that he thinks that Putnam’s account needs a stronger version of causation.

Chrisley’s project gives us a theory of implementation that includes a reference to the inputs and outputs of the physical system in terms of the structures that allow the system to *take in* and *produce* inputs and outputs. This is what prevents Searle’s WALL from implementing a Turing machine:

> Consider an animated display of a Turing Machine on a computer screen. Since, *ex hypothesi*, there is a one-to-one correspondence between the states of the display screen and the states of some Turing Machine, Searle and Putnam would apparently claim that the screen realizes the Turing Machine, if anything does. But it seems clear that we would say that the screen *depicts* a Turing Machine, but is not itself one. One reason why we would deny it computational status is because the state of the screen that corresponds, in the putative interpretation function, to a computational state *A* does not produce, as a causal effect, the screen state that corresponds to the successor computational state *B*, even though the Turing Machine depicted does make a transition from state *A* to state *B*. Computational states must be able to cause other computational states to come about.

Chrisley thinks that trivializing the interpretation conditions is not a problem for a theory of computation—that it is acceptable for a multitude of semantic interpretations to be assigned to the computational system. What Chrisley adds to a theory of implementation is the idea that the inputs and outputs, as he characterizes them, should be defined in advance, and this prevents trivial implementations. Chrisley offers a theory of implementation that addresses a trivialization of the implementation conditions.

Copeland (1996) argues that to say a physical system computes is to say that there exists a modeling relationship of a certain kind between it and a formal specification of an algorithm and supporting architecture. For Copeland, a formal specification of an algorithm is something like a machine table, and architecture is a physical structure. An algorithm is understood as a finite list of machine-executable instructions such that anyone or anything that correctly follows the instructions in the specified order is certain to achieve the result in question (337). To say that an algorithm *p* is specific to an architecture is to say that a machine with this architecture can run *p* and that each instruction on *p* calls for the performance of some sequence of the primitive operations made available in the architecture. 'Primitive operations' are those procedures for which the architecture is specifically enabled. For example, a program calling for multiplications can only run on such an architecture because the compiler (the program that converts instructions into machine code) replaces each multiplication instruction in the program with a series of addition instructions (337). These primitive operations will vary from architecture to architecture.

To see how the view works, call the formal specification of the architecture and algorithm in question "SPEC." Let SPEC take the form of a set of axioms.[24] SPEC is connected with some physical system, *e*. The bridge between *e* and SPEC is carried out in terms of a labeling scheme for *e*.[25] A labeling scheme for an entity consists of two parts: (1) the designation of certain parts of the entity as label-bearers, and (2) the method for specifying the label borne by each label-bearing part at any given time (338). Copeland thinks that this can help eliminate trivial implementations. He considers Fodor's example involving the solar system and Kepler's Law. Fodor, when discussing rule following in psychology, says, "If you are willing to attribute regularities in the behavior of organisms to rules that they unconsciously follow, why don't you say… that the planets 'follow' Kepler's law in pursuit of their orbits around the sun" (Fodor 1975, p. 74). Copeland argues that acting in accordance with Kepler's law is not the same as executing an algorithm

[24] Copeland notes that nothing in his view turns on using the axiomatic method as opposed to some other style of formalization.
[25] Labels can be understood in different ways. Some examples given by Copeland included the following: eight binary digits associated with groupings of subdevices where the possible states (maybe measured in volts) are divided into two mutually exclusive classes and a state is labelled '0' if it falls into the first and '1' if it falls into the second (338).

because Kepler's law is not an algorithm. Even if it were, Copeland thinks that we cannot say whether the solar system computes without a description of the solar system's computational architecture.

Copeland's view includes a second part, explaining the behavior and function of the labeled entity. He says to explain the system's behavior is to interpret the labeling scheme. Importantly though, Copeland is *not* thinking of interpretation in terms of ascribing external semantic content to the system. Instead, he is thinking more along the lines of *internal semantics*. A specification of an internal semantics maintains Copeland's view as a non-semantic theory.[26] Copeland describes the interpretation of the labeling scheme of system *M* in terms of implementing an algorithm in a program.[27] The axioms for *M* are specified in terms of the instructions in the machine code. For example,

$Ax1$ If $I = 00000001$ ACTIONS-IS ($A \Rightarrow D$)

$Ax2$ If $I = 00000010$ ACTIONS-IS ($A \Rightarrow A + D$)

The intended interpretation of a statement of the form 'if *X* ACTION IS *Y*' is that the occurrence of *X* produces or brings about the action of *Y* with the same logical strength as the statement that *X* causes *Y* in a counterfactually supporting way (p. 341). Here Copeland adds an additional constraint on the physical system: counterfactual supporting transitions between the physical states. This is what makes Copeland's view a counterfactual implementation view. His account is

[26] See Chapter One, sections 2.5 and 2.6.
[27] See Chapter One, section 2.4.

a theory of implementation that strengthens the physical requirements of the implementing system by requiring that it have the appropriate type of architecture such that it is capable of implementing an algorithm, and the physical state transitions must have the appropriate kind of causal structure.

Melnyk (1996) also gives us an implementation view. Melnyk's target is Searle's argument against Strong AI, and, in particular, his goal is to respond in terms of the Robot Reply. To do this, he strengthens a causal mapping view. He thinks Searle's thought experiment involving placing himself inside a room with a rule book does not meet implementation requirements because it fails to include an embodiment condition. Here Melnyk characterizes Searle's thought experiment as an example of a trivial implementation. To avoid this kind of trivial implementation, the embodiment condition causally connects the state transitions inside the room to the outside world appropriately (p. 406). Melnyk proposes that we might think of the connections as something like nomological dependence. While Melnyk's goal is not to specifically provide a theory of computation, he does give a theory of implementation that addresses the problem of trivial implementations.[28]

2.3 The dispositional account

[28] Melnyk's project *overlaps* with the computation literature, but there are some differences in the dialectic. Melnyk can be best understood as contributing to the debate about the nature of physical realization (see Francescotti 2002 for a reply to Melnyk and Kim 1998 & 1999 for a similar view, and Papineau 1995 for an explanation of what is at stake in the debate). In some places, this debate overlaps with debates about functionalism and sometimes computational functionalism specifically. Whenever the debate moves toward computational functionalism, we end up with an interesting overlap with the computational literature. So, while many of Melnyk's interlocutors are not engaging in the debate about computational implementation, his view fits in quite nicely with other implementation theories because he engages with the relation between computational architecture and a physical system. Note, though, that just because an author describes their view in terms of a "realization function" does not mean they are engaging in the same debate as Melnyk (for example, Scheutz 1999).

Klein (2008) offers a dispositional theory of implementation to reply to Maudlin's 1989 conclusion that conscious states can neither supervene on computational states nor be explained by an appeal to computation. Klein calls this problem the "superfluous structure problem," where in order to count as an implementation, it looks like a system must have more structure than is strictly active during a particular computational process.[29] To solve the problem, Klein gives us a view that requires that the computational states map onto the dispositions of the implementing system. He defines dispositions as things that are or supervene on objects' intrinsic properties, giving them conditional causal powers. An object's disposition is picked out by specifying the conditions under which the disposition would manifest and the effect that this manifestation would have—meaning they support counterfactuals. The dispositional account strengthens the causal relations by requiring a dispositional relation between the physical states where for any computational state transition of the form $s_1 \rightarrow s_2$, if the system is in the physical state p_1 that maps onto s_1, the system manifests a disposition whose manifestation is the transition from p_1 to the physical state p_2 that maps onto s_2 (Piccinini & Maley 2021).[30] This view is a theory of implementation that addresses trivial implementations.

2.4 The computational complexity accounts

In this section, I will introduce a category of theories I call "computational complexity accounts." These accounts go beyond strengthening causal relations in various ways by introducing a series of complexity requirements of varying types.

[29] This can be seen as a consequence of Searle's Chinese Room argument. If you can't get something like "conscious understanding" from the physical-causal computational manipulations, then there must be some required structure that exists over and above the computational structure—a superfluous structure.

[30] Bishop (2009) cites an additional paper from Colin Klein (2004) titled 'Maudlin on Computation' as a "working paper." I asked Colin for a copy, and he said that the 2008 paper I cite here is what ended up as the final, published copy of the paper that Bishops cites.

Schuetz (2021) gives a theory of implementation in terms of bisimulation.[31] This view defines the mapping between the computational structure and the physical system. According to this view, in order for a physical system to implement some computational structure, the computation and the physical system must share one crucial aspect: the state transitional structure of the computation is at most as complex as the causal transitional structure of the implementing physical system with respect to the computational sequences (p. 560). Scheutz calls the similarity between the complexity of the transitional structures a "bisimulation." This notion of implementation is meant to *relax* the conditions established by mapping views that relate computational states to physical states. Instead of requiring a mapping between states, this view requires only a correspondence between computational and causal *paths*—every computational *sequence* corresponds to a causal *sequence,* and every causal to a computational sequence.[32] Scheutz's bisimulation view addresses the implementation relation by incorporating a causal complexity requirement.

Millhouse (2019) gives what he calls a "simplicity criterion for physical computation." He aims to provide a formal criterion for physical computation that allows us to distinguish between competing computational interpretations of a physical system objectively. Millhouse describes his view as distinguishing between "interpretations," but he takes a 'computational interpretation' to be a description of a physical system as "an abstract discrete state machine" (160). Contrary to the terms he chooses, Millhouse provides a theory for how we are to "interpret" some physical system

[31] He also discusses bisimulation in his 1999 paper.
[32] Scheutz (2001) is classified as a causal view in the SEP article on physical computation, but it's not clear to me that it actually counts as a causal view in the ways that makes the previous views "causal." Scheutz also does not seem to categorize his own view as causal either, but instead, as one that focuses on "computational complexity."

as *implementing* a discrete state machine. So, “interpret” here ends up being misleading. What Millhouse is addressing is the implementation relation.

To address triviality, Millhouse relies on Kolmogorov complexity to show how information distance might help address triviality.[33] Kolmogorov complexity is the measure of information contained in a string of symbols or data where complexity is defined as the length of the shortest possible description that can produce the desired output. Information distance is the measure of similarity between two pieces of data and their shortest possible descriptions. So far, I have described trivial implementations as an overgeneralization problem: too many systems implement computational structures. When Millhouse characterizes trivial implementations, he does so in terms of being unable to adjudicate between the computational structures that a *single* system implements. This is an interesting deviation from how trivial implementations are typically presented, but it is still a type of trivialization of the implementation relation.

Millhouse gives his view in terms of *P,* an abstract discrete state machine, *M,* an ordered pair of functions mapping states *P* to *M*. The interpretation is *I* and its component function *S* and *C*. To address implementation, Millhouse says that a physical system *P* implements an abstract discrete state machine *M*, if there is an ordered pair of functions mapping (i) the states of *P* to the states of *M*, and (ii) inputs to *M* to possible interventions in *P*. In addition, he requires that the mapping support the counterfactuals specified by the abstract machine’s transition function (160-161). So far, his view is similar to Chalmers's (1996). What he adds to the view is a way to evaluate whether

[33] See Chapter One, Section 3.2 for more on Kolmogorov information.

the computational description is appropriate for the given machine—which helps eliminate trivial implementations.

He argues that to prevent trivial implementations, interpretations (in terms of Kolmogorov Complexity) assigned a smaller value should count as better than those assigned a higher value where for any physical system P, and any pair of abstract machines, M_1 and M_2, if the optimal interpretation of P as M_1 is better than P as M_2, then P more genuinely implements M_1 than it does M_2. The Kolmogorov complexity of I, $K(I)$ is the length of the shortest program that computes S and C. This gives the simplicity criterion for choosing between implementations. Essentially, the criterion allows us to quantify the minimum amount of information required to use a physical system as a particular type of computing machine (163). Although Millhouse's view is strange in some ways, it still follows the basic pattern of giving a theory of implementation that address trivial implementations.

3 Conclusion

The influence of philosophy of mind on implementation theories should be evident based on the abovementioned views. Many were interested in justifying the computationalism paradigm because Computationalism is silent on how the brain realizes the computational states. Putnam's earliest account, now known as the "simple mapping" view, was one of the first attempts to make good on the computationalist claims by giving a theory of how we might understand the brain as implementing a computational structure. So, what we now know as the literature on computation and implementation has its foundations not in computer science but in the philosophy of mind. However, we can easily see how this extends beyond the context of cognition when we consider that any metaphysical account that addresses the nature of the implementation relation should necessarily get it right about *all* computing systems. In other words, it should ensure that only the right systems implement computational structures.

In this chapter, I have demonstrated that non-semantic theories address the implementation relation in various ways and that they address the problem of trivial implementations. Although the terminology may differ across the various views in some ways, all theorists are interested in relating a computational structure to a physical system.

6 Semantic theories of interpretation

1 Introduction

In the previous chapter, I argued that non-semantic theories of computation naturally lend themselves to the implementation question, as I have defined it (and as most of them define it). While these views vary across different dimensions, what they have in common is that they directly address questions about computational implementation. In doing so, they address the problem of trivial implementations. What these theories do not address is the interpretation question. This is by design; they do not *intend* to address questions beyond implementation. But, by not addressing the interpretation question, they leave the problem of trivial interpretations on the table. Whether authors of these views take the problem of trivial interpretations seriously or not does not alter the result that their view only addresses one aspect of physical computation.

Views that address interpretation provide a way to understand how to attribute a particular computational process, description, etc. (sometimes just called a "function") to a physical system.[34] Some who offer a theory of implementation do not think that trivial interpretations threaten a theory of computation for various reasons, some of which include the claim that what a system computes is a matter of *perspective*, so we do not need to *fix* the interpretation (Chalmers 1996). But this reason is not strong enough to eliminate the need for a theory of interpretation. For

[34] In Chapter Two, I mentioned that a function is placed at Marr's computational level. Some people who give a semantic view are interested in ascribing the function to a physical system, which means that in the context of Marr's framework, they want to relate the computational level to the implementation level. This is different from implementation views that relate the algorithmic level to the implementation level. This is another place where we can see the difference between implementation and interpretation. So, my framework has benefits for understanding the relationship between Marr's levels. It seems to me that when we relate the computational level to the implementation level and when we relate the algorithmic level to the implementation level, we are doing *different* things, especially if the levels are meant to capture *different* kinds of explanations. My framework helps make sense of this.

example, if you think that what a system computes is a matter of perspective, you still have not said anything about how the perspectival stance *works* or which *contexts* are appropriate. Without an accompanying theory, it is an empty claim. Moreover, some who offer a theory of interpretation *also* take this stance.[35] So, just because you think that attributing computational descriptions is a matter of perspective (or context), you have not dispelled the need for a theory that explains which perspectives and when are appropriate—there must be *some* constraints on how we make our ascriptions—*this* is what a theory of interpretation does for us and what interpretation theorists take to be an essential aspect of physical computation. Nevertheless, as I have also argued, theories of implementation are not responsible for the problem of trivial interpretation because a theory of interpretation has the resources to handle the issue. In this chapter, we meet views that pair with a theory of implementation, and together we can see them as addressing two important aspects of physical computation in a complimentary way.

In this chapter, I will argue that "semantic accounts" address a second aspect of physical computation, namely, interpretation. The individuation question is most obviously derived from semantic accounts because some involved in the debate answer it explicitly. Semantic theorists offer a semantic account because they take on the metaphysical commitment that computation necessarily (or essentially) involves the manipulation of representations (Piccinini & Maley 2021, Sprevak 2010, Shagrir 2022). This sometimes takes its cue from Fodor's famous claim that there is "no computation without representation" (Fodor 1981, p. 122). This metaphysical claim is sometimes given as a response to the perceived failure of non-semantic accounts (what we now know as theories of implementation). So, sometimes those who provide a semantic account take

[35] For example, Rescorla 2013 and Shagrir 2022.

themselves to be giving a theory of physical computation that answers the individuation question. My framework distinguishes the individuation question from the implementation and interpretation questions. This means that semantic theorists may provide an answer to the individuation question in the form of a metaphysical commitment, but their *theory* is an answer to the interpretation question that addresses the problem of trivial interpretations.

However, a second way of answering the individuation question adopts a weaker stance. This weaker stance says that our *explanations* regarding computational systems should reference what the system represents. This weaker version is compatible with the idea that computation may not involve the manipulation of representations in some cases. I identify this second, weaker answer to the individuation question because, as we will see, some theorists that propose a theory of interpretation do so because they think that our *explanations* should reference representations in some cases. If we do not acknowledge this weaker notion, then some accounts that require reference to semantic content will not be properly categorized as semantic accounts, generating an extraneous third category of "middle" views.[36] So, as long as the view references semantic content in some way, regardless of the metaphysical commitments regarding the relationship between the content and computation, I will refer to the view as a "semantic account." The idea that semantic accounts involve answers to the individuation question that vary in strength is no surprise, given my argument that the individuation question is distinct from the implementation and interpretation questions.

[36] For example, Rescorla (2013) maintains that his view is neither semantic nor non-semantic because he thinks that we only *sometimes* need to appeal to semantic interpretation, but as we will see later in the paper, his view is clearly a semantic account—it just adopts the weaker claim.

In this chapter, I will argue that semantic accounts address the interpretation question, and in virtue of doing so, they address the problem of trivial interpretations:

> **The Interpretation Question**: Which computational process(es) does the physical system perform?

> **Trivial interpretation**: A physical system can be interpreted as performing every computational process.

This chapter is far more revisionary than the previous chapter. Before I get started, I will flag that semantic accounts are quite messy when it comes to figuring out how they all fit together. Some people take themselves to be answering the individuation question while others do not. Some people refer to their view as answering the “implementation question,” and then they frame the question in several different ways. Some claim to address “triviality” but only tackle one type. This is sometimes because they think that an answer to one *implies* an answer to another, but not always. When it comes to implementation views, even the more complex ones, we can see that they all generally aim at a similar target. In contrast, it is not so clear that semantic views do. I think this is the source of many of the problems in the literature and a very good reason why a framework like mine is needed.

Once we re-classify semantic accounts as theories of interpretation, we place distance between an answer to the individuation question and a theory of interpretation. Within my framework, it does not matter how interpretation theorists answer the individuation question—they can be committed to whatever computation/representation combination they want. A consequence of this is that it makes a variety of implementation and interpretation views *compatible*. You can answer the individuation question however you would like while adopting whatever pair of theories you think best explains the various aspects of physical computation. I show this in Chapter Eight.

2 Breaking up with Jerry Fodor

Jerry Fodor's work on computation and representation has been widely influential in the philosophy of mind and the philosophy of computation. His famous quote, "There is no computation without representation" (Fodor 1981), is often cited as support for the claim that any computational theory should necessarily involve a reference to semantic properties—and sometimes even further that computation is the manipulation of *representational* symbols.[37] One of Fodor's main contributions to cognitive science and philosophy of mind was pairing the computationalism thesis with the representational theory of mind. This pairing produced Fodor's famous Language of Thought Hypothesis (Fodor 1975). The representational theory of mind combines Turing's work on computation with intentional realism—the idea that there are lawful relations between intentional states that are physically realized. These representational states have both syntactic structure and a compositional semantics leading to the hypothesis that thinking is a process that takes place in an internal language of thought.

When Fodor's statement that there is "no computation without representation" is invoked in various philosophical contexts, "representation" is understood as a mental state bearing a

[37] Pylyshyn also references this slogan in his 1984 book 'Computation and Cognition: Toward a Foundation for Cognitive Science' on page 62.

semantic/representational relation to the external world—that their truth conditions are determined by how the external world is.[38] This is because Fodor was giving a theory of mind and *not* a theory of computation. It makes sense that when Fodor discusses a computational theory of mind that he thinks that mental computations represent—that they are *intentional*. After all—this is exactly what his theory is meant to explain. However, it is a mistake to extend this quote to a theory of physical computation and claim that for some physical process to count as a computation requires that it necessarily has semantically interpreted physical states. Put differently, using this quote to answer the individuation question misappropriates the context in which it was given in the first place.

There is reason to think Fodor would not endorse this kind of answer to the individuation question anyway. Consider the entire context in which the quote is embedded: "To think of a system (such as the nervous system) as a computer is to raise questions about the nature of the code in which it computes and the semantic properties of the symbol in the code. In fact, the analogy between minds and computers actually implies the postulation of mental symbols. There is no computation without representation" (Fodor 1981, p. 122). But Fodor is talking about neural computations and not computation *simpliciter*. So, what is the nature of the code involved with *neural* computation? The semantic properties involved with neural computations are supposed to be mind-independent and fixed by the causal relations to the external world. There is a tight connection between the semantic and formal properties of neural computation. Still, the *necessity* of the external semantics is a feature of neural computations and not computing artifacts. So, to assume this stance assumes too much of computation simpliciter. Computers have an external semantics, but not in a mind-

[38] For examples of this see: Piccinini (2006).

independent way. Moreover, some people who give a semantic account take a perspectival stance toward the semantic ascription—they think of the representations as mind-dependent.

Sometimes Fodor is characterized as giving a semantic account of computation—the kind meant to be a contender for how we should understand the nature of physical computation in general. For example, Piccinini (2015) argues that Fodor (1975) and Pylyshyn (1984) take computational vehicles to require structures that have a language-like combinatorial syntax and semantics, and if the vehicles do not have this structure, then they don't count as computations (p. 29). But this is a serious overstatement of their view. Fodor and Pylyshyn were giving a computational *theory of mind*. They thought that mental computations required a combinatorial syntax and semantics to account for the systematicity and productivity of thought—a reason to reject neural networks as a sufficient model of mental processes. They were not arguing that neural networks weren't computational, just that they were the wrong kind of computational *architecture* to account for *mental* computations (Fodor and Pylyshyn 1988).

Before importing Fodor's quote into the literature on physical computation, we should ask whether his analogy between neural computation and physical computation, in general, holds within the context and whether his project genuinely fits in with the goals of a theory of physical computation. I contend that it does not. Just because some theoretical posits and arguments from the functionalism/computationalism area of philosophy import nicely into the computation literature does not mean everything does. So, I propose that we break up with Jerry Fodor when it comes to our metaphysical commitments about the nature of physical computation.

3 From a semantic account of computation to a "theory of interpretation"

In this section, I will taxonomize various semantic accounts that have been offered throughout the literature. This will include accounts that accept the strong and weak ways of answering the individuation question. I will then classify them as theories of interpretation that address the problem of trivial interpretations.

3.1 Semantic theories and philosophy of mind

In section two, I argued that we should reject Fodor's slogan as motivation for giving a strong semantic response to the individuation question. But it is still possible to offer a theory motivated by the desire to understand the mind or brain as a computing system. Many theorists in the philosophy of mind give accounts of the nature of physical computation by addressing computation in cognitive psychology.

Pylyshyn (1984) tells us that a computer "is a physical object whose properties vary over time in accordance with the laws of physics" (p. 55) and later that "to count as computation rather than simply any functionally described physical system… it must contain symbols that are interpreted" (p. 62).[39] I consider this to be Pylyshyn's answer to the individuation question if he were to give one. He says, "Because an unlimited number of physical properties and their combinations can be specified in the physical description, there is, in fact, an unlimited number of such sequences" (p. 55). Pylyshyn notes how mappings from physical states to computational states might not even be possible and that "not only can… computational sequences be realized in any digital computer ever made, or that will ever be made… they can be realized in devices operating in any imaginable

[39] After this statement, he gives Fodor's famous quote "no computation without representation" in a parenthetical.

media—mechanical… organic—even a group of pigeons trained to peck as a Turing machine" (p. 57). In this passage, I understand Pylyshyn as describing the problem of trivial implementations. Pylyshyn is saying that we shouldn't characterize computing systems in terms of some mapping because it will end up trivializing which systems count as computing systems.

As a response to the problem of trivial implementations, Pylyshyn offers a semantic account that tells us something about the computational process in the machine (p. 55). He says, "One of the most important similarities between cognition and computation… is suggested by the observation that the explanation of how computation… proceeds must make reference to what is represented by the various intermediate states of the process just as the explanation of cognitive processes must make reference to the content of mental states" (p. 57). To address the problem of trivial implementations, I argue that Pylyshyn gives us a theory of *interpretation*. He is concerned, explicitly, with the question "what computation is being performed" (p. 57). Although his motivation to develop his view was the problem of trivial implementations, he gives a view in terms of *interpretation*.

A quick detour is in order. This switch happens with many of the semantic views. But it is not by accident. Many people who give a semantic account think their view addresses trivial implementations because of how they answer the individuation question or because if computation is necessarily a semantic process, it rules out any system that cannot represent in the correct way. So, a semantic account *implies* an answer to trivial implementations. First, *implying* an answer is not a theory. What we need is a theory for how to understand when a physical system implements a computational structure in a way that *prevents* trivial implementations. Second, this has consequences for an account of physical computation that not all semanticists would be inclined to take on. If you think that what eliminates trivial implementations is the fact that only computing systems represent *and* you think that a semantic interpretation is a matter of perspective, then *ipso facto*, you must also accept that the property of *being a computer* is a matter of perspective. This dissolves the need for a theory of computation in the first place. Perhaps we could make do with the appropriate kind of perspectival realism. I don't intend to argue for this any further, but I hope that my point is made that this position has downstream consequences. Moreover, it does not resolve the problem of trivial implementations. Returning to Pylyshyn, it still doesn't solve the problem of how a group of trained pigeons can implement a Turing machine. If pigeons are a stand-in for computational states and Pigeons represent (they do), then it looks like being a representational system does *not* prevent trivial implementations after all. Something more needs to be said.

Returning to Pylyshyn, his semantic account unfolds in the following way. First, he thinks that if we want to answer questions about what computation is being performed, we need to refer to the meanings of the symbols in both the expression and the printout of the output. This incorporates the internal semantics (expression) and the external semantics (the printout). To explain why the machine prints the numeral "5" when it's provided with the expression "(PLUS 2 3), we must refer to the meanings of the symbols in both expression and on the printout. To start, Pylyshyn gives us a numerical example.

> Let us say that the expressions consist of the atomic symbols o and x arranged in strings of arbitrary length. In this example the states of the memory registers are designated by such

> expressions as o, x, ox, xo, xx, oox… and so on; each expression designates some possible state of each machine's memory registers. Let us further suppose that when a certain pattern (designated by the symbol "⊕") occurs in a portion of the machine called its "instruction register," the machine's memory registers change states according to a certain, specifiable regularity… (p. 59)

The specifiable regularity can represent different functions, such as addition, given the adoption of the appropriate semantic function and specific requirements. The semantic function maps the strings of o's and x's onto numbers and requires regularities in the computer's state transitions to correspond to the mathematical operations defined over the interpretations of the intended symbols. In other words, the state transitions must preserve the intended interpretation function (p. 60-61).[40] This gives us a way to understand the preservation of the semantic content ascribed to the computational states. This is an important feature of the theory because, ultimately, Pylyshyn is interested in giving an account of cognition.

Pylyshyn's answer to the individuation question involves the idea that computations must contain interpreted symbols—symbols that represent numbers, letters, or words, etc. But he also says these interpretations are not "wired into the machine…the designation is provided by a person who takes the symbols to be about something—that is, the person gives the symbols a semantic interpretation" (p. 62). Pylyshyn's stance here is that the interpretation is a matter of perspective.

[40] This concept also underpins the Language of Thought. One way to describe what's happening here is that the syntactic manipulations *preserve* the semantic interpretations. This is a critical aspect of the Language of Thought: the syntactic properties are causal, but in such a way that the semantic content is preserved in a rule-governed way. This is the formalist's motto: "If you take care of the syntax, the semantics will take care of itself" – the beauty of a physical symbol system (Haugeland 1985, p. 106).

And his view is that a theory of interpretation that has the conceptual resources to handle trivial interpretations. Although he targets trivial implementations, he does not provide a solution to the problem.

Dietrich (1989) also offers a semantic theory in the context of cognitive psychology.[41] According to Dietrich, computationalism requires that we posit and interpret mental states and structures. He calls this the "computational strategy." He extends this claim to computers by adopting the view that computers are systems that manipulate representational (interpreted) symbols. To take on the computational strategy is to attribute to a subject or physical system *S*, the computation of a certain function, *F*. Fixing *F* for a particular system "is seeing *S* as doing something regular. Therefore, fixing *F* grossly, but not trivially, answer the question *What is S doing*?" (p. 126-127).[42] Specifying the question of what the system is doing is an indication that he is addressing the interpretation question. He briefly mentions triviality but does not give an example of what type he responds to. Regardless, as soon as he begins to provide a theory of what the system is "doing," he is well on his way to providing a theory of interpretation. Dietrich thinks that semantic attribution connects *S*'s state changes within something we can easily understand. He gives us an example of how to apply the computational strategy by describing a four-function calculator:

> Consider something very different from us: an ordinary, four-function calculator, a device that can add, subtract, multiply, and divide. For our purposes, assume that we are given a

[41] Dietrich's target is Stich's syntactic theory of mind (1983).
[42] My italics.

> four-function calculator which uses a brand new architecture, and that our task is to figure out what this new architecture is (p. 128-129).

After hypothesizing how it works based on the outputs that it gives, we might ask the following questions and come to the following hypotheses:

> Why does the input have to have the argument and operations in the order that it does (post-fix order as opposed to the more natural infix form)? Specifically, we want to explain the way our calculator seems to 'remember' and 'forget' numbers, and that when it gives the output we want it seems to 'remember' two numbers, operate on them, and 'remember' the result, which can then be used as one of the arguments for the very next operation. ... Our hypothesis then is that our calculator is a stack-based machine... Stacks have two associated operations: push and pop. Data items are pushed onto the stack as they are stored and popped off from the 'top' of the stack as they are used. We hypothesize that our calculator pushes numbers onto its stack as they are input, and pops them when an operation is input. If "22+" is the input, the first "2" is pushed onto the stack, then the second "2" is pushed. When "+" is received, the two "2"'s are popped off the stack, added together, and "4" is pushed back onto the stack (p. 129-130).

Dietrich is interested in the way that we reason about the architecture in the example. First, we assume what it *does*, and this assumption is based on the *context* that in our culture calculators are used to compute instances of functions like addition. Second, we explained how the calculator performed addition by analyzing its addition into a series of stack operations. Third, we drew a correspondence between the states of the calculator and the states of the stack, which required an interpretation of the states of that calculator as being the states of the stack, and we interpreted the changes in the stack as the computations of the function push and pop. For Dietrich, semantic attribution comes from the context in which the attribution is made. This relativizes our scientific explanation to our goals and views of the world (p. 131). The only requirement is that the attributed contents explain observed behavior (p. 132). Dietrich has given us a theory of interpretation that we can use to address the problem of trivial interpretations. He doesn't seem to think that representations are essential to computations but that an interpretation of the computational states is necessary for a sufficient *explanation* of the computational system—one we can use and one we can understand. This is evident when he tells us that his project is epistemological; he is concerned with how we can know what a computer is *doing*. So, Dietrich ultimately gives provides the weaker answer to the individuation question while giving a theory of interpretation.

Smith (1982a, 1982b) tells us that semantic attribution lies at the core of the notion of computation. He says, "What distinguishes an abacus, a calculator, and even a full-scale computer, from other rule-governed complex artifacts like steam plants and food processors, is that the best explanation of [their] behavior is formulated in the domain of *interpretation*… computers…. are… just those devices whose functional architecture we understand in terms of external semantical attribution."[43] Smith (2002) summarizes what he calls his "long-term investigation of the foundations of computer science" and claims to give a theory of physical computation that is meant to do justice to computer science, artificial intelligence, and computational cognitive science (p. 24).

[43] This quote was reported by Dietrich, who pulled it from an unpublished manuscript: Smith (1982a). I have also come across some information leading me to believe that these claims can also be found in Smith's dissertation: Smith (1982b).

Smith still holds tight to his answer to the individuation question we saw developing in the early 80s. He tells us that he takes "computation to be semantic (not just by assumption, but as an unavoidable lesson from empirical investigation)" (p. 34).[44] Smith's view is both strong and revisionary. What we have from Smith is an answer to the individuation question that could lead to a theory of interpretation. But we never actually get a theory out of Smith. Instead, we end up with the conclusion that computers do not form a distinct category. This pessimistic conclusion says that we'll *never* be able to say anything about what counts as a computer and what doesn't. Within my framework, I interpret this conclusion in terms of *theory*. I take it that Smith's conclusion is ultimately that we cannot give a proper theory of implementation nor a theory of interpretation because even though we know that computation is a semantic process (his position), we aren't going to be able to specify the nature of the various aspects of computing systems such that we can lay down a theory that allows us to pick those features out in the world.

Smith gives us an answer to the individuation question but nothing more. This detail gets missed because the individuation question is run into the implementation question, which is then run into the interpretation question. When these questions aren't pulled apart, an answer to *any* of them starts to look like a "theory of computation." So, even though Smith does not give us a theory *at all*, he still gets credited with providing a semantic account of physical computation simply in

[44] An interesting part of Smith's approach is that he thinks that a theory of effective computability is a theory about the physical world and that recursion theory is a theory about marks (p. 41). This turns the way we generally understand these theories upside down. Generally, effective computability and recursion theory belong to the formal side of computer science—the *mathematics*. But Smith understands them instead as issues regarding the nature of the physical world. Especially given this insight, we can see that Smith's concern with physical computation results from dropping the mathematical side of computation into the physical world. So, when he gives us a theory of physical computation, he's actually giving us a theory of mathematical computation (if we don't accept the flip). Because of this, it's not clear to me that Smith belongs in the grouping of semantic views or even that he belongs in this project. But he's often cited as giving an account of physical computation, and I wonder if it's because this flip is being ignored.

virtue of answering the individuation question by claiming that computation is necessarily a semantic process. I've included him in this list to demonstrate how not distinguishing the questions in the way I do can give the illusion of a theory of physical computation when in actuality, no theory has been given—just a claim—an answer to the individuation question.

In this section, I've given several accounts from the philosophy of mind. These views (except Smith's pessimistic conclusion) all answer the individuation question by requiring some reference to semantic content by adopting either the stronger or weaker version. But a semantic view cannot solve the problem of trivial implementations because it does not provide a theory of implementation. So, instead, what these authors end up doing is giving a theory of interpretation, one that is perhaps cut out to help resolve the problem of trivial interpretations.

3.2 Semantic accounts in the context of a "theory of computation."

In this section, I will taxonomize views that fit into the computation literature proper. These views directly engage with the individuation question while attempting to resolve triviality problems in some way. In this section, I will argue that these views only address one aspect of physical computation—namely, interpretation—and thus, they should be classified as theories of interpretation that directly address the problem of trivial interpretations (regardless of whether they think that their answer to the individuation question implies a solution to trivial implementations). I deliberately start this section with Sprevak's semantic account. His account and presentation of the literature are demonstrably the most "sticky." I hope the work I've done so far will illuminate how he's running all three questions together. Consider this our first test for future applications of my framework. After Sprevak is concluded, I will move on to other semantic accounts.

Sprevak (2010) argues that representational content plays a critical role in determining computational identity. He adopts the 'received view' about computation: that computations must involve representational content.[45] He thinks that representational content is a necessary condition that does crucial work in determining computational *identity* and that representation does much of the hard work in answering the questions about *individuation*. He also uses the phrases 'computational process,' 'physical computation,' and 'computational system' interchangeably and as all meaning 'the implementation of a computation by a physical system.' It is difficult to discern what these terms are tracking in the broader context of the literature, but let's see if we can figure it out. Consider the following passage where I've emphasized some terms in bold:

> Where no restriction is placed on the type of representational content involved, e.g. no ban on distal objects serving as content. Egan and Piccinini accept that computations often do involve representational content, but they argue that such features are accidental to a system's computational nature, and have no bearing on its computational **identity**. My claim is that such representational content is a necessary condition that does crucial work in determining computational **identity.** It is worth emphasising that even on this view, representation would still only be one condition on computational **implementation**: there are further conditions that **a physical computation** should satisfy, and additional properties that **differentiate physical computations**. However, representation does much of the hard work in answering the questions about **individuation** that motivate an account

[45] Sprevak has recently cast some doubt on this, however. At the workshop on "The Nature of Physical Computation" hosted by Oron Shagrir on June 27, 2022, Mark expressed that he thinks that this commitment is far too strong because it's difficult to see how computation "all the way down"—or unconscious computational processes at the microphysical level could be representational in this way.

> of physical computation. Consequently, it is a feature of physical computation that should be of special interest (p. 3).

His use of the term ‘identity’ when discussing representational content gives the impression that he is using the term to signal the identity of a computational *process* or *state*. This can be corroborated by referencing the 2008 Piccinini paper he cites. In this paper, Piccinini argues that “computational states are individuated at least in part by their semantic properties” (p. 205). If we follow Piccinini’s paper, the term ‘individuation’ tracks the idea that we can individuate computational states semantically.[46] Sprevak argues in this paper that non-semantic views are threatened by universal realization. This *shifts* the discussion from the way that we individuate computational states to one about “realizing” computational states and then to the problem of trivial implementations. So, when Sprevak invokes the term “implementation,” he uses it to describe a realization function—or a version of the implementation relation related to the problem of trivial implementations. So, he thinks that we should individuate computational states semantically to avoid trivial implementations. Hopefully, we can begin to see the mismatch blooming.

In this paper, Sprevak gives us an argument for why we should semantically interpret computational states and then says that this is a solution to the problem of trivial implementations. What makes this project even more difficult is that he opens the paper with three questions and

[46] In this paper, Piccinini argues for a view that he calls the “functional view of computational individuation.” According to this view, computational **states** are individuated by their functional properties, and their functional properties are specified by a mechanistic explanation in a way that need not refer to any semantic properties. This view can be seen as part of Piccinini’s account of physical computation (which he thinks answers the implementation question as he defines it), but it’s not the view itself.

claims they are “among the key questions that a philosophical theory of physical computation should answer” (p. 1).

> What makes a physical process a computation?
>
> What is the difference between a computation and any other process?
>
> Under what conditions are two computations the same or different?

The fact that Sprevak engages these questions at the start of his paper makes the following debate even more confusing. Given this, the Piccinini and Egan papers cited no longer seem to be the appropriate targets for his view. Because of this, I find this entire project to be very confusing. Using Egan’s view to target trivial implementations seems misplaced. She is targeting computational explanation in cognitive science. Moreover, Piccinini is not giving a theory of physical computation in the target paper. So why does Sprevak target these papers? What is he doing? Why is he doing it this way? Sprevak is by far the worst offender when it comes to running questions of implementation, interpretation, and individuation together. But I think this only becomes obvious when we disambiguate the formulation of the implementation question and then clarify the implementation and interpretation questions as distinct.

Luckily my framework can help. It is clear to me that, charitably interpreted, he is interested in the individuation question as I have defined it. He argues that computation essentially involves representational content (p. 3). This, combined with some other claims he makes, lets us formulate his answer as follows: what distinguishes computing systems from non-computing systems is that computing systems essentially manipulate representational content. In addition, Sprevak mentions that he is interested in computational *processes* and not just computational *states*. This indicates that he is interested in understanding what the computing system is *doing*—what process it is performing. So, he wants an answer to the interpretation question: which computational process is the system performing? In addition, when Sprevak says, "the implementation of a computation by a physical system," we can disambiguate how he reads "a computation" and define it in terms of a computational *process*. This means that whatever theory he gives will address interpretation and *not* implementation. So, when he describes the problem of "universal realizations" as "a mapping between a mathematical computation and a physical system, and such mappings are cheap" (p. 10), we know that he takes himself to be arguing against implementation views that are, as we've seen *non-semantic*.

Sprevak gives his theory by presenting some paradigmatic cases of computation involving representation. One example involves an electronic computer that takes electrical signals as inputs and yields electrical signals as outputs. The input and the output represent (typically) 0's and 1's. The strings of 0's and 1's can represent many things: an email message or even a picture displayed on the screen.[47] Until we interpret the strings of symbols, we cannot say which computational process is taking place. The fact that the strings can represent many things is the problem of trivial interpretations. Until we interpret the system, it could be doing anything! Here we can see that Sprevak is specifying the problem of trivial interpretations—*a switch* (that I endorse).

[47] There is a clue here about why a theory of interpretation does not give a theory of implementation. Implementation views are closely related to the individuation question in that they help us to identify which systems implement computational structures—this helps us to pick out computing systems from non-computing systems. The examples here are of systems that we already know are computers—they are paradigmatic computing systems. But they are used as examples for *why* we need a theory that involves reference to representational content. So, one way to see this is that a theory of interpretation comes *after* a theory of implementation. Once we know that the state transitions or the physical processes implement a computational structure, then we can move onto interpreting the system. When we only think about systems that we already know implement a computational structure, the implementation question slips away into the background. But note that it's **never** actually answered.

Sprevak characterizes his view in terms of input-output (I/O) equivalence. Recall that he asks, "Under what conditions are two computations the same or different?" This is where he directly answers this question. This is part of the interpretation question—sometimes we want to know whether different systems with different architectures or structures are implementing the same computational process.[48] He asks us to imagine two I/O equivalent systems that are made out of different physical materials (these systems aren't computational, but it helps generate the intuition that Sprevak is pushing on).

> One system is made of silicon and takes electrical signals as inputs and outputs, the other is made out of tin cans and string and takes marbles as inputs and outputs. Suppose the two systems are computationally I/O equivalent. What could their I/O equivalence consist in? The respective inputs and outputs of the two systems are different and may be so different as to not have any physical or functional properties in common. The only answer seems to be that their respective inputs and outputs *represent the same thing* (p. 22).

The best way to understand Sprevak is to categorize him as giving a theory of interpretation that answers the interpretation question. He thinks this helps to identify which computational process

[48] This gives us another clue as to why we should distinguish between implementation and individuation. It is possible that the same computational process can be realized by a variety of computational architectures. For example, I'm using Word on my Mac, but I can also use Word on my PC. Macs and PCs have different operating systems and architectures (that's why you specify which one you are buying the program for!). If this is possible, we can consider the computational architectures of Macs distinct from the computational architecture of a PC without having to say anything about whether I'm using Word. We want to know whether the physical hardware is implementing which structure. Sprevak doesn't care about the structure; he cares about the process—the program.

a system performs because otherwise, it can be seen as performing any computational process. This is also a solution to the problem of trivial interpretations.

It is worth visiting Sprevak (2018) as well. In this paper, he says the following:

> A theory of implementation aims to describe the conditions under which a physical system does and does not implement a computation… a theory of implementation tells us, for a physical system *X*, and abstract computation, *Y*, the conditions under which '*X* implements *Y*' is true or false… a theory of implementation says what we mean by our talk of computational implementation and explains how it reduces to… conditions regarding abstract computations in physical systems… sometimes a theory of implementation is also described as a theory of the computational *relation*. This relation is envisioned as a metaphysical bridge between the abstract realm of mathematical entities… and the concrete realm of physical systems… (p. 2-3).

Untangling these descriptions, we see both the implementation and interpretation questions being posed. He then goes on to say that a theory of implementation should answer two questions:

1. Under which conditions is it true/false that a physical system implements a computation?
2. Under which conditions is it true/false that a physical system implements one computation rather than another?

I hope that it's clear at this point that Sprevak understands "a computation" in a very specific way: in terms of the identification of a computational process. I take it that when he reads statements regarding computational implementation, he *always* takes on this reading of the phrase. This is the only way to make sense of how he can clearly state the implementation question in relational terms (*a la* Chalmers) while also understanding the task of a theory of physical computation as requiring an interpretation of the computational process. After we distinguish the individuation question from the implementation question and then from the interpretation question, we can make sense of Sprevak's characterization of what a theory of computation is meant to do.[49]

Rescorla (2014) gives us what he calls a *descriptivist* theory of what it means for a physical system to implement an abstract computational model. This is a follow-up to his previous 2013 paper where he argued against what he calls "structuralist" theories or what we've discussed as non-semantic mapping theories of implementation like Chalmers' view. In these new iterations, Rescorla's target still includes mapping views from Chalmers and Putnam. He also thinks that his view can address triviality arguments. Specifically, Rescorla's paper is titled 'A theory of implementation,' which I take it means that he intends to answer the implementation question. He formulates the question in several ways:

1. What is it for a physical system to realize a computational model?
2. When does a concrete physical entity—such as a desktop computer, a robot, or a brain—implement a given computation?

[49] Sprevak's work gave me the most trouble initially. But in a lot of ways, it demonstrated the importance of clearly articulating what a theory of computation should address. If we don't distinguish the questions properly, descriptions of what's at stake for a theory of computation get run together and many people engaged in the debate end up very clearly talking past each other.

Immediately, we can recognize these questions as ones that address the interpretation question. A "given" computation is a *specific* computation—which computation? He takes a computational model to be an abstract description of a system that reliably conforms to a finite set of mechanical instructions where the instructions dictate how to transition between the states. So, a physical system "*implements"* the computational model when the system reliably conforms to the instructions. Thus,

> Physical system *P* realizes/implements computational model *M*
>
> Just in case
>
> Computational model *M* describes physical system *P* (p. 1278)

According to Rescorla, 'mechanical instructions' are something like a recipe, music score, or blueprint, as well as formal algorithms like the Euclidean algorithm. For Rescorla, the computational model is *interpreted*—it describes what the system is doing, and this is why his view can be classified as a semantic account.

To make his view more precise, Rescorla tells us what it means for a computational model *M* to accurately describe some physical system *P*. *M* accurately describes *P* just in case *P* reliably moves through "state space" according to the mechanical instructions encoded by *M*. A "state space" (canonical state space description) is understood as an ordered-quadruple [S, I, Ω, s_0] (stated simply):

S:	the state space where S is a possible computational sate
I:	the set of inputs
Ω:	the transition function which encapsulates how the current computational state determines the next.
S_0:	the privileged initial state of the system

The state space accurately describes physical system P just in case:

1. For each state in the state space, s is a possible state of P.
2. For each set of inputs, i is a possible input to P.
3. P reliably conforms to the transition function Ω (counterfactually).
4. Absent malfunction, P always begins computing in state s_0.

State space descriptions do not have to be computational in any sense. The idea is that a computational model is translated into a canonical state space and then is said to be realized by the physical system if it adheres to the rules above.

What Rescorla is doing here is changing the nature of how we understand what it means to be a computational model. He is taking computational models to be something like a recipe—interpreted instructions that describe the transitions of physical states in a system: behavioral descriptions! He is trying to formalize the notion of "realizing" a behavioral description, but he calls the behavior description a "computational model." Despite a heavy dose of formalization, Rescorla is basically giving us a way to understand when some description of a system in terms of *what the system is doing* is appropriately ascribed to the system. This is a way of addressing the interpretation question: identifying which computation the system performs.

Rescorla says this is the appropriate way to address questions about physical computation because scientists *use* abstract computational models to *describe* physical systems and that a model, *as used within current descriptivist practice,* places conditions on which physical systems implement the model.[50] So, Rescorla is using the term "implements" in regard to a specific interpreted model, not a computational structure. Despite using the term "implement," he offers a theory of interpretation. He compares his view to what he calls "structuralism about computational implementation"—these are our mapping theories of implementation. This is the major shift between his 2012 and 2014 paper. In the newest iteration, he says that descriptivism is *compatible* with structuralism, but ultimately, he thinks that we should still reject mapping accounts.[51] He thinks we should reject structuralism because, for similar reasons present in 2012, he does not think that a mirroring relation is always sufficient for implementation. But, if Rescorla wants to reject structuralism, he will still need an actual theory of implementation to replace it. Moreover, when Rescorla discusses his view as an implementation view, I've argued that he's mistaken. Instead, his view is a theory of interpretation and nothing more.

I've characterized Rescorla's view as a semantic account, but ultimately, he rejects those views, too, and refrains from identifying himself as such. But really, what he's rejecting is their answer to the individuation question. He thinks that there are cases where we do not need to appeal to

[50] His italics.
[51] As opposed to 2012 where he argues explicitly against structuralist accounts in favor of his descriptivist view because sometimes implementation requires instantiating some representational properties. This paper is titled: 'Against structuralist theories of computational implementation.'

semantic content when describing what a computational system is doing. Rescorla rejecting both non-semantic "structuralist" views and semantic accounts demonstrates further a need to pull these questions apart and address them individually. As I mention at the start of this chapter, Rescorla does still give us an answer to the individuation question, but he does so by deploying a weaker answer than some: he thinks that representations are *sometimes* necessary for identifying computational processes. And he's giving us a theory of interpretation.

Rescorla also addresses triviality. He thinks that because his view incorporates counterfactual conditionals (step 3 above), he avoids trivial implementations. But, as we've seen, his view is not a theory of implementation. So, even though he thinks that the physical states should be counterfactually supporting, it doesn't mean he solves the problem of trivial implementations. He also thinks that his view solves the problem of trivial interpretations. Whether it does or not is up to the next counterexample that pops up in the literature, but because this *is* the target of a theory of interpretation, we can grant that his view accomplishes this task.[52]

The final account in this chapter is Shagrir's semantic account. Shagrir (2022) has the most well-developed semantic account on the market.[53] He also *seems to* implicitly adopt my distinction between a theory of implementation and a theory of interpretation, although, in some areas, it's not clear. Shagrir's "master argument"—that semantic properties play an essential role in the *individuation* of physical computing systems such as laptops and brains—can be understood as his

[52] Rescorla doesn't cleanly distinguish between the two types of triviality that I define, but he does mention both types indirectly and argues that his view addresses them both.
[53] Shagrir's view has been in development for over a decade and has recently come together in his 2022 book titled 'The Nature of Physical Computation.' For papers leading up to his seminal project, see Shagrir 2001, 2006, 2012, 2020, to name several.

answer to the individuation question. I derive the individuation question from Shagrir (2022) and Piccinini (2015). Shagrir says:

> I offer an extended argument for a variant of the semantic view of computation. This view states that semantic properties are involved in the nature of computing systems. Thus, laptops, smartphones, and nervous systems compute because they have certain semantic properties. Stomachs, hurricanes, and rocks, for instance, which do not have semantic properties, do not compute (p. 1).

This answers the individuation question but with a slight variation. Here he says "compute" rather than what counts as a *computer*. I think this signals that he is interested in what the system computes. He makes it clear later when he says that "architectural accounts" give a different response:

> *Architectural accounts*, as I call them, share the view that possessing the right kind of architectural profile is a necessary condition for computing; therefore, systems that lack this select profile do not compute… I argue against the necessity of architectural profiles. These profiles play a minimal role, if any, in distinguishing computing from non-computing physical systems" (p. 2-3).

But notice here that when he presents the individuation question again, he does so in terms of "computing from non-computing systems." I think this is a clue that they are answering different questions. The last sentence in the quote is the individuation question, as I have defined it. But recall that theories of implementation don't answer the individuation question in any direct way. Shagrir gives this answer *on their behalf*! This is important to recognize because it frames the debate in terms of the individuation question specifically. This frames the in terms of non-semantic versus semantic. "Architectural accounts" are implementation theories, but when the implementation question isn't distinguished from the interpretation question, Shagrir is licensed to rename them. But to be clear, people who give a theory of implementation are **not** framing the debate this way. This comes from Shagrir, and it only causes more confusion.

Shagrir gives his answer to the individuation question as a response to trivial implementations: some physical systems simultaneously implement different automata at the same time, in the same space, and even in the very same physical properties ("simultaneous implementations") (Shagrir 2020). Using my framework to restructure this, we can see that Shagrir is concerned with trivial implementations and intends to provide a semantic theory that overcomes this problem. But we know that theories of interpretation don't provide a solution to trivial implementations, and Shagrir's view is no different.

But Shagrir does acknowledge that the idea that computation often operates on entities that carry representational content is something that can be accepted by both those who propose a non-semantic account. This is consistent with what I've argued so far. In several places, Shagrir states that his semantic account is consistent with any non-semantic theory of implementation. He says, "First, the semantic view of computation is consistent with a non-semantic view about implementation. A non-semantic view about implementation asserts that the relation of implementing an automaton by a physical system does not involve semantic properties" (Shagrir 2020, p. 4088). The key insight, though, comes from the following passage:

> Of course, if you think that computation is (nothing but) the implementation of an automaton, then you cannot hold a semantic view about computation and a non-semantic view about implementation. But the semantic view of computation is not committed to the identification of computation with implementation. A proponent of the semantic view of computation is free to hold that the implementing of an automaton (or some formal structure more generally) is necessary for computation and yet is non-semantic. *She can say that counting the implemented automaton as a computational structure—and counting the implementing physical system as a computing system—essentially involves semantic properties* (p. 4088).[54]

Let's unpack this. "If you think that computation is nothing but the implementation of an automaton" seems to reference what he calls "architectural accounts." So, if you hold an architectural account, *"then you cannot hold a semantic view about computation and a non-semantic view about implementation*." Who holds the architectural account, though? When he describes architectural accounts, they seem to be our non-semantic theories of implementation. But, as I've argued, implementation views do not answer the individuation question. Shagrir asks *and answers* it on their behalf. So, this first part may be true, but it is not true of our implementation views. This second half makes more sense. You can accept that implementation is non-semantic

[54] My italics.

while answering the individuation question semantically! This is one of the main conclusions of this project and my framework.

So far, we have seen that Shagrir gives us a very explicit answer to the individuation question. But, despite what he says in the above passage, an implementation view is still needed. He gives a theory of interpretation while targeting trivial interpretations. Take his master argument:

> 1. A physical system might simultaneously implement several different automata $S_1, S_2, S_3, \ldots$
> 2. The contents of the system's states determine (at least partly) which of the implemented automata, Si, is relevant for computational individuation.
>
> Conclusion: The computational individuation of a physical system is essentially affected by content (p. 4089).

(1) describes the implementation of a formal structure. Shagrir acknowledges that this is a non-semantic process. (2) is where the interpretation question starts to take form—we identify which computation is implemented in the system by reference to the semantic content of the states of the system. This is a claim about computational vehicles—that the *identity conditions* of the computational vehicles essentially involve semantic properties.

Shagrir argues that his view overcomes the problem of simultaneous implementation—the problem of the same token physical system implementing different automata at the same time, in the same space, and even with the very same physical properties. But he's understanding the automata as interpreted, not as purely structural. This is shown when he describes some physical system implementing both an AND gate and an XOR gate simultaneously. Under a mapping relation, the system can be seen as implementing both gates (p. 4091). Shagrir argues that simultaneous implementation is a different kind of triviality than the Putnam-Searle kind of triviality we say in Chapter Three. He says that the difference is that simultaneous implementation is achieved through the very same physical properties of the system (its voltages). In addition, the mapping is held fixed while the voltages are coupled in different ways. This makes it so that the same voltages implement different automata at different times. Recall that we can derive both types of triviality from Putnam and Searle's arguments: we can see how there is both a trivialization of the implementation relation and a trivialization of the interpretation conditions that arise from their arguments. When Shagrir holds the mapping fixed but couples the physical states in different ways such that the states implement the same automata differently, he's generating a triviality result in terms of interpretation only. In this respect, Shagrir does target the appropriate type of triviality.

Shagrir then goes on to give his theory of interpretation. According to him, whether a system performs one computation or another depends on the environment in which the system is embedded. It is possible that the AND function is implemented in one context while the OR function is implemented in another context. What fixes the interpretation is the context in which the computation occurs (p. 4095).

Shagrir's project would benefit from separating the individuation question from the interpretation question and decoupling the debate from one about how we should answer it—that is *not* the locus of the debate because the debate should not be about semantic versus non-semantic accounts. Regardless, he offers us a theory of interpretation that addresses trivial interpretations.

4 Conclusion

In this chapter, I have argued that semantic accounts address the interpretation question, and by doing so, they address the problem of trivial interpretations. I began by arguing that we should abandon Fodor as a reason for answering the individuation question in terms of representational content. I have also shown that there needs to be more clarity when specifying the target of a theory of physical computation. We've seen that those who generate a semantic view frame the implementation question in many unhelpful ways, leading to confusion about what they're trying to do and what we want out of a theory of computation. I've given a framework that allows us to restructure semantic accounts cleanly as views that provide a theory of interpretation: one aspect of physical computation. I have also argued that these views are equipped to address trivial interpretations. Hopefully, this has given some insight into how theories of implementation and theories of interpretation can be compatible. But, if you can't see it quite yet, I will propose a way in the final chapter.

Next, I will engage with mechanistic accounts of physical computation. These views have always been around, but they have taken center stage in recent years. In the next chapter, I will consider how they can be included in my framework, what questions they might be addressing, and which aspects of physical computation they may target.

7 Mechanistic accounts and a theory of computation

1 Introduction

In the previous two chapters, I argued that non-semantic theories of computation answer the implementation question and are equipped to handle trivial implementations and that semantic accounts answer the interpretation question and are equipped to handle trivial interpretations. This restructures the literature into two categories: implementation views and interpretation views. I also motivated the idea that calling them "non-semantic" and "semantic," along with not distinguishing the questions, gives the false impression that they are *alternatives* to a theory of computation when they actually address different aspects of it. There are other ways that some authors have contributed to the impression that they are opposing theories, but I think that rebranding views in terms of implementation and interpretation and clearly defining the questions they address is enough to make some progress toward consistency in the literature. In this chapter, I will present three accounts classified in the literature as "mechanistic." But, this third category has mostly come into prominence as an alternative to both semantic and non-semantic accounts due to the work of Gualtiero Piccinini. This third category emerges *only* if you think semantic and non-semantic accounts are *alternatives*. However, because implementation and interpretation are *compatible*, mechanistic accounts no longer qualify as a bonafide alternative. This chapter explores a place for them in the debate.

The three views I will present in this chapter come from Fresco (2014, 2019), Milkowski (2013), and finally Piccinini (2015). I distinguish between "weak" and "strong" mechanistic accounts. Weak accounts answer the implementation question and avoid trivial implementations by arguing that the physical system should meet some mechanistic requirement. Thus, weak mechanistic accounts should be categorized as implementation views. Milkowski is an example of a view that does this. I explore his view to show the differences between mechanistic accounts, but ultimately, his view should be adopted as an implementation view. I also explore Fresco's account to show that sometimes a theory of physical computation turns out not to be one at all. I presented a semantic account (Smith 2002, 1982a, 1982b) in the last chapter that had this same problem. In the same way that some people take themselves to be giving a theory of implementation when they are giving a theory of interpretation, some people take themselves to be addressing the nature of physical computation when they are actually addressing something else. I think this is a symptom of the more significant problem I identify in this project. Without a framework that clearly specifies target questions, views that address neither can freely enter the debate.

On the other hand, strong mechanistic accounts are *wholly* mechanistic in that they do not address implementation or interpretation; instead, they form a third category. Piccinini's view is a strong mechanistic account. It turns out that Piccinini's view is the *only* account that I think answers the individuation question by giving a full-blown theory rather than forking off into a theory of implementation or individuation (I explore this option in the next chapter). But ultimately, I will argue that it will only be *compatible* with implementation and interpretation views if we reject some of his underpinning assumptions about the debate. In other words, we would need to adopt 'The Mechanistic Account'*. Piccinini introduces his view based on the perceived failure of both non-semantic and semantic accounts. He thinks that his view can *replace* both views. But, unless his view addresses the implementation and interpretation questions, it cannot replace either.

In section two, I will provide some background information on how to understand mechanisms. This will help with later sections in the paper. In section three, I will present Milkowski’s weak mechanistic account, followed by Fresco’s view. Finally, I will evaluate Piccinini's strong mechanistic account in what makes up the bulk of this chapter. I will argue that he builds three main conceptual shifts into his view. I will also argue that these shifts are too strong, and ultimately, throwing out a framework because there is confusion in the literature is not the best way to move forward. Finally, I will present several views that engage with the mechanistic account. I will show that two of them do not recognize the conceptual shifts and argue against what I call the mechanistic account*. The last view I present argues directly against the mechanistic account targeting its concepts.

2 Thinking about mechanisms

Before we get started on the “mechanistic account,” it will be helpful to review the concept of a mechanism in the philosophy of science, particularly by the new mechanists. The new mechanists offer qualitative descriptions of mechanisms designed to capture how scientists use the term and how the concept is used in experimental and inferential practices. There are different ways to characterize mechanisms, but all characterizations contain four basic features: a phenomenon, parts, causings, and organization.

The phenomenon is the behavior of a mechanism as a whole. All mechanisms are mechanisms *of* some phenomenon, so in the case of a neuron, the mechanism of the action potential generates action potentials. The boundaries of a mechanism are fixed by reference to the phenomenon that the mechanism explains. The components in a mechanism are components by virtue of being relevant to the phenomenon (Craver & Tabery, 2019). New mechanists understand mechanisms either as producing, underlying, or maintaining the *phenomenon*. Production is a kind of causal sequence that terminates by producing some end product. If a mechanism underlies a phenomenon, it is understood as a capacity or behavior of the mechanism as a whole. For a mechanism to maintain a phenomenon, we understand the phenomenon as a state of affairs that is held in place by the mechanism.

Defining the *parts* of a mechanism has been notoriously difficult given the part/whole relation and difficulty with mereological theories. For example, Piccinini (2020) takes a “composition” approach to define the relation between wholes and their parts. For him, parts compose the whole, and a whole decomposes into parts where the relation between the two is that higher-level properties are realized by lower-level properties, and lower-level properties realize higher-level properties (Piccinini, p. 7). There is much more to be said here, but hopefully, we can see the overlap with mereological theories.

Mechanists understand *causation* in different ways. One example is to derive causation from the concept of a mechanism so that causal claims are claims about the existence of a mechanism. So, what makes a causal claim true is that there is a mechanism at some lower level. An activity-based account says that causation should be understood in terms of productive activities. So, according to this view, activities are a kind of cause. Views like this don’t say anything about the nature of causation. Instead, they rely on science to say what activities are and what features they might have.

The *organization* of a mechanism can also be understood in different ways. This is related to issues regarding the *parts* of the mechanism and how they come together. Organization can be characterized spatially, temporally, or in terms of patterns. The spatial organization includes location, size, shape, position, and orientation, while temporal organization includes the component activities' order, rate, and duration. Examples of patterns include patterns of organization and repeated patterns. Others define organization in terms of modules, jointness, or levels (this is how Piccinini understands organization). When I discuss the mechanistic account, I will give Piccinini’s take on a multi-level explanation of the organization of a mechanism.

3 Mechanisms and physical computation

In this section, I will give two views categorized in the literature as “mechanistic accounts” of physical computation. The first is from Milkowski, and I will argue that he gives us a weak mechanistic account that ends up being an answer to the implementation question. Because of this, we should categorize it as an implementation view. Next, I will present Fresco’s account and argue that it does not give a theory of physical computation, even though it’s categorized as such across the literature.

3.1 The weak mechanistic account

Milkowski (2012) argues against mapping theories of implementation. These are the theories of implementation that I introduced in Chapter Five. Milkowski focuses on the inner complexity of the physical system, arguing that more than a mapping is required to satisfy the implementation relation. From the start, we can see that he is directly targeting implementation views, and, like those views, he places further constraints on the mapping to overcome trivial implementations. He discusses WALL and argues that it lacks the kind of *inner complexity* that a computer requires and that this is why it does not run Wordstar. He thinks a mapping view allows for this kind of implementation because it overlooks the need fo inner complexity. His solution is that when drawing the relation, one must account for *all* relations between the subsequent steps of the code to ensure that all relations hold between the steps in the code and the states in the physical system (p. 222). Because Milkowski argues that we should consider all states of the system (and not just some arbitrary mapping), we should only consider causal chains *relative to* the isolated system. From this, he specifies two additional requirements for a theory of implementation: 1) the complexity criterion; and 2) the relative isolation criterion.[55] These additional features make Milkowski's view "mechanistic," but as we saw in Chapter Five, other non-mechanistic accounts also introduce a causal complexity criterion. So, rather than calling this view a mechanistic account, it should be classified as a theory of implementation that addresses trivial implementations.

My concern is not that Milkowski's view is being called "mechanistic" *per se*. The problem is that it is treated as belonging to its own class within the context of a "theory of computation" when instead, it should be understood as a *sub-class* within the available options for a theory of implementation. So, Milkowski rightfully belongs with implementation theories, perhaps as a causal complexity account.

Milkowski (2013) expands on his view. In this book, he suggests that the best way to give a theory of computation is to rely on the causal structures of the world. What brings Milkowski's view into

[55] See also Milkowski (2007).

the mechanistic category is that he wants to give a mechanistic explanation of cognition by rejecting functionalism. He is a proponent of computationalism but not one that relies "on a Cartesian gulf between either software and hardware or mind and brain" (p. ix).[56] In this project, Milkowski elaborates on the complexity and relative isolation criterion. Most of his argument for these two criteria involves giving a negative exposition on how the various non-semantic and semantic theories fail for various reasons. But we can gather some of his position about the nature of physical computation by looking at what he rejects and accepts from the various views.

When addressing counterfactual accounts, Milkowski argues that counterfactuality is almost always a vacuous constraint if we don't specify a specific kind of structure of both the physical and formal systems (p. 34). But ultimately, he rejects mappings because "it escapes me in what way [formal modes of implementation] would be an improvement over simple talk of physical processes and their causal structure" (p. 35). This second claim develops this account beyond his 2012 project, where he accepted a mapping relation. But, as I will argue, dropping the mapping will not de-classify his view as a theory of implementation.

Milkowski's view is non-semantic because he thinks that information processed by a computer need not refer to or be about anything in the distal environment to be the inputs or outputs of a computation. In this respect, he aligns himself with Fresco (2010) and Piccinini (2006). But he continues on that physically realized inputs and output states trivially carry semantic information as part of the causal nexus. But all that is required to count as information is that the computational

[56] In this book, Milkowski is giving a computational theory of mind specifically. But, in the process he analyzes "what it is for a physical process to realize or implement a computation" (p. 25).

vehicle has at least one degree of freedom (p. 46). He takes his view to accommodate any kind of computation, provided the computation is specified in terms of relations between input and output information states. But, at the threat of being too broad, he specifies that a complete specification of the mathematical model of computation involved is required for implementation.

Milkowski appeals to functional and structural notions such as "mechanism" and "level of organization." *This* addition is what makes his view more mechanistic than his 2012 project. When describing a formal computation, he says, "Computations are performed by mechanisms, which is to say that computational functions are realized by the organization of the mechanism's parts" (52). Here Milkowski defines computation as something that can be realized by a functional mechanism. He says, "Computations occur only in highly organized entities and cannot float freely without any realizing machinery" (p. 52). This connects formal computation with the physical world. Computations do not exist in some abstract realm. Instead, computation is a physical process. He specifies that a mechanistically adequate *model* of computation is implemented when the input and output information streams are causally linked and that this link, along with the specific structure of information processing, is completely specified. Because he thinks that computation is a physical process, the (formal) computational model must have the appropriate kind of structure such that it can be linked with the causal structure of the world.

This characterizes computation as a physical process rather than "bridging" formal computation with the physical world. Milkowski argues that the computational descriptions can be understood in terms of levels of composition where the mechanisms have three levels of organization:

1. Constitutive: The lowest level
2. Isolated: The level at which the parts of the mechanism are specified along with their interactions (activities and operations)
3. Contextual: The level at which the function the mechanism performs is seen in a broader context (the embedding conditions) (p. 55)

The constitutive level is causal—it constitutes the causal structure that matches the computational characteristics of the entire mechanism in such a way that it contributes to the functioning of the isolated level. Which levels of organization a given description should include requires reference to the explanatory goals of a particular research project. Milkowski says the isolated level corresponds to the "traditional" focus of philosophical theories of implementation because it is the only level at which the abstract specification of computation is to reflect the causal organization.

Instead of adding a constraint on the causal relations in the physical system to strengthen the mapping, Milkowski's mechanistic account requires that the computational description at the isolated level not be completely independent of the constitutive level. This is also what escapes the problem of trivial implementations because it allows us to distinguish "real" computing system from a "mere hodgepodge of physical states… such as tornadoes, piles of sand, or planetary systems… that normally do not realize any information-processing capacity" (p. 57). Notice that Milkowski brings the relation between the abstract descriptions and the physical system back together; he thinks we can't analyze them independently! So, after all this, we *still* have a theory of implementation.

Milkowski is interested in showing how we can find non-trivial cases of physical mechanisms that implement computational architectures. For him, a correct computational description will specify the parts of the mechanism and its organization where the organization is understood in terms of "architecture" (p. 60). Different models of computation will require different mechanisms. So, to realize a Turing machine architecture or an FSA, the specification of the organization will be different. This also addresses trivial implementations because the abstract organization corresponds to the computational architecture in a functional way where the functional role of a component is one of its causal roles, such that it contributes to the system behavior of the mechanisms.

3.2 Fresco on digital computation

Fresco's (2014) view specifies *digital* computation as realized in physical systems to specifically understand the relationship between computation and cognition. He thinks that the best way to get insight into the computational theory of mind presented in psychology, neuroscience, and cognitive science, is to examine possible answers to the problem of what concrete digital computation is.[57] Fresco gives a view within the context of classical physics, dealing with physical features such as force, motion, heat, entropy, and electricity. "Using classical physics, the hardware of conventional computing systems can be analyzed, for example, at the macroscopic level at which registers change their stored data or at the level at which logic gates operation on various input line" (p. 6).

[57] Although he does think that computationalism does not have to be committed to digital computation specifically.

An essential part of Fresco's view is that he extends digital computation to connectionist networks. This is because he relaxes the requirement that digital computation requires a physical symbol system. By relaxing this requirement, the scope of digital computation is broadened, and connectionist networks are captured. According to Fresco, broad digital computational systems like connectionist networks implement a task analysis without decomposing the overall operation into subtasks performed by individual components. Instead, they build their networks as mechanistic models. Thus, Fresco takes *connectionist models* to be mechanistic explanations – mechanism *sketches*.[58] They are sketches of mechanisms because they explain cognitive phenomena without relying on the decomposition into subtasks. Instead, each subtask is *distributed* across the network layers without any straightforward localization. According to Fresco, this counts as mechanistic because the instructions for the manipulation of data in the system demonstrate an attempt to account for the decomposition of tasks into simpler subtasks.

Fresco's account is unique because he relies on connectionist models as the implementing mechanisms. However, this is no surprise because Fresco wants to give a theory of *cognitive* computation in digital terms! This means that if he wants to think of cognitive architecture as a connectionist network, then he has to eliminate the need for symbolic structures as understood in the classic digital way. For Fresco's view to be an account of concrete computation, we also have to accept that connectionist models count as implementations in the same sense that computers are physical.[59] You might think that neural networks are implementations of symbolic systems, but

[58] Another way to put this is that Fresco is giving an argument that connectionist networks are *implemented* digital computers. This is contentious though. See Weiskopf (2011) for an argument against connectionism as a physical implementation.

[59] His account is best placed within the debate about cognitive architecture, and we can best describe his position in the following way: "a connectionist thesis is that cognition is sub-symbolic computation. Accordingly, cognition should be explained by connectionist network activity, in a more generic sense than the association between stimuli

neural networks are still abstract, mathematical structures. So, even though Fresco discusses physical mechanisms and implementations, it does not mean that he's given a theory of physical computation.

4 The strong mechanistic account

In this section, I will present what I call the "strong" mechanistic account of physical computation. Piccinini's view is generally called "the mechanistic account." As I progress in this section, I will refer to the view this way. I categorize the mechanistic account as "strong" because it does not address implementation or interpretation. Instead, it's *wholly* mechanistic. This view does not answer questions about implementation or interpretation because it dissolves the implementation relation and because it does not tell us how to interpret the system. To dissolve the implementation relation, the mechanistic account reconceptualizes the nature of what it means to give a theory of computation. Explaining how the reconceptualization happens is the purpose of this section. I will also argue that we can best understand this view as providing a full-blown theoretical answer to the individuation question. This is no surprise, though, because the individuation question enters the literature through his work (and was later adopted by Shagrir). Finally, I will argue that the mechanistic account is incompatible with theories of implementation and interpretation because of the conceptual shifts that underpin it. But it might still be possible to include the view if we adopt a version that rejects the conceptual shifts and adopt the mechanistic account*.

Piccinini argues that computing mechanisms manipulate vehicles according to a rule defined over the vehicles and, possibly, certain internal states of the mechanism (p. 1). Over the course of the

and response… on this view, it is implicit that connectionist networks broadly are capable of computations that are not limited to discrete manipulations of symbolic representations… (p. 211).

development of the mechanistic account, Piccinini makes several conceptual and terminological shifts that obfuscate the different aspects of computation that non-semantic and semantic accounts seek to capture. In the next section, I will address these conceptual shifts in turn. As I address these shifts, I will simultaneously formulate the mechanistic account.

4.1 Functional analysis as a mechanism sketch

Piccinini's formulation of the mechanistic account stems partly from the claim that functionalism and computational explanation have been conflated in the literature (Piccinini 2004). He blames Fodor's (1968) use of Turing Machines tables as generating the conflation. He argues that this conflation has led some philosophers to believe that to do functional analysis is to provide a computational explanation. To de-conflate functional analysis and computational explanation, Piccinini argues that we should understand functional analysis as *an explanation that describes a system in terms of complex capacities that are made up of more basic capacities*. To do this is to give a mechanism sketch of a system where the explanation becomes a full-blown mechanistic explanation when relevant additional details are filled in. This is Piccinini's first shift:

I. To do functional analysis is to provide a mechanism sketch.

Remember that implementation views are an extension of the functionalist paradigm. It was a way to connect the abstract descriptions with the physical systems they described. Already we can see a push to eliminate the relation between formal descriptions and the physical world. According to Piccinini, functional analysis, in terms of a mechanistic explanation, eliminates the problem that functional analysis *assumes* a computational explanation. But why think that we to eliminate functional analysis within the functionalist framework? I think that Piccinini's argument goes too far. The conclusion should be that not all functional analyses require computational explanations, not that we should eliminate functional analysis in terms of computational ascriptions altogether. This seems strange, given how profitable functionalism has been when it comes to explanations in cognitive science.

Piccinini thinks functionalism and computationalism have been conflated historically and into contemporary literature. He goes even further when he argues that because of the "conflation," many take functionalism to *entail* computationalism. However, Piccinini's argument that they have been "conflated" is too strong, and the abundant use of computational descriptions does not mean an assumed entailment relation exists. In Chapter Two, I discussed the close connection between functionalism and computationalism and showed that they are related but distinct frameworks. Functionalism commonly relies on computational explanations, especially in cognitive science, because computational descriptions of cognitive processes help us to explain and model cognitive processes. Piccinini throws the baby out with the bathwater because he doesn't like some of the consequences of the framework.

Piccinini argues that we can avoid these issues by thinking of functional analysis in terms of giving a mechanism sketch and that we can understand computational ascriptions in functionalism as task analysis. "Task analysis explains a capacity of a system as the execution of a sequence of operations of finitely many steps by that system, which need not be analyzed into components and their functions" (Piccinini, 2004, p. 825). According to Piccinini, Fodor "firmly inserted into the philosophical literature a thesis that begged the question of whether all psychological capacities could be explained by program execution: the thesis that psychological theories were traditionally formulated as lists of instructions for producing behavior (Piccinini, 2004, p. 826).” According to Piccinini, this formulation of functional analysis left no room for other formulations that explained mental capacities in terms of types of components and their functions to take hold.[60] Because early on, Fodor (and Putnam) formulated their psychological theories in a way that related minds to Turing machines and program execution, it became the mainstream way of approaching an explanation of the mind while leaving little room for any other kind of explanation to emerge. So, he ultimately wants to shift the explanatory paradigm from functionalism to a mechanistic explanation. This is how the implementation relation is dissolved. If we do functional analysis in terms of a mechanistic explanation, then we no longer need to relate the formal, computational descriptions to the physical world.

But, even if Piccinini wants to eliminate functional analysis as it is done within the functionalism framework, it is not going anywhere anytime soon, which means that we will still need a theory of implementation to account for the connections between abstract computational descriptions and the world—we still need a theory of implementation. To provide a theory of implementation is to extend beyond the functionalist paradigm and ask for a theory of how the physical system implements the abstract mathematical formalism. This links the abstract description with the physical particulars, a project beyond the scope of functionalism, beyond the scope of task analysis by computational explanation, and distinct from mechanistic explanation.

[60] See Deutsch (1960) for an early mechanistic account.

Piccinini's first shift aims to recharacterize functional analysis as a mechanism sketch. In some ways, this is uncontroversial-- functional analysis if approached by providing a mechanistic explanation quite often. But functional analysis within the functionalist paradigm that deploys computational descriptions to give a deliberately abstract characterization of a physical system is not the result of a conceptual mistake. Moreover, it has contributed substantially to the traditional formulation of the implementation question.

4.2 Abstraction and medium independence

Piccinini maintains that a computational explanation should be understood as a mechanism sketch and that a full-blown mechanistic explanation should include a description of the relationship between medium-independent vehicles and the physical particulars of the system. Medium independence is introduced as a replacement for multiple realizability, although, as I will argue, it is a very different concept. Because Piccinini's view does not refer to mathematical formalisms, multiple realization is characterized in terms of abstraction. This frames Piccinini's second shift:

1. A physical system performs a generic computation if it has medium-independent vehicles that are processed according to a rule.

The mechanistic view is a theory of what generic computation amounts to.[61] He argues that concrete computations can be defined without referencing all of the structural properties of the physical system. Instead, we only need to identify the computational properties relevant to the rules defining the computation. Thus, they are medium-independent because concrete

[61] Generic computation is explained in detail below.

computations and their vehicles can be defined independently of the physical media that instantiates them (Piccinini 2015).

Introducing medium independence as a replacement for multiple realizability is further evidence that the implementation relation is dissolved. To show this, more explanation of what medium independence, vehicles, a rule, and generic computation amount to is needed. Medium independence is developed from Craver's (2001) account of mechanisms. According to Craver, mechanisms are composed of entities and activities. Entities are the things that engage in various activities and activities are the producers of change (Machamar *et al*., 2000). The organization of the entities and activities determines the ways in which the mechanism produces a phenomenon. Functions are the roles played by entities and activities in mechanisms. Machamar *et al*. describe the procedure that scientists use to provide a mechanistic explanation of a physical process.

1) They begin with an idealized description of the set-up conditions. Scientists idealize the set-up conditions by defining them in terms of static time slices. The start conditions include relevant entities and their properties, including structural properties, spatial relations, orientations, and various relevant enabling conditions. Most of these features are parts of the mechanism itself.
2) They provide the termination conditions. These are idealized states or parameters describing a privileged endpoint.
3) They specify the intermediate activities that take place in the mechanism. This is a characterization of the intervening entities and activities that produce the end stage from the beginning.

This three-step process provides a mechanistic explanation in a nested hierarchy. Within this hierarchy, lower-level entities, properties, and activities are components of the mechanisms that produce higher-level phenomena.

Scientists do not always provide a complete mechanistic explanation of this kind. They also provide mechanism schemata and *sketches*. A mechanism schemata is an abstracted description of a type of mechanism at varying degrees of abstraction depending on how much detail can be included. In contrast, a mechanism sketch is an abstraction for which bottom-out entities and activities cannot (yet) be supplied or which contains gaps in its stages. These gaps are present because there are missing pieces that scientists do not yet know how to fill in. Because there are missing pieces in the explanation, sketches are meant to indicate the further work that needs to be done in order to understand the mechanism in its entirety. Because they function as placeholders in this way, they are often abandoned for schemata once more information is obtained. This concept of mechanism sketch as an *abstraction* is what underpins Piccinini's concept of medium independence. To be clear, mechanism sketches are *abstractions*: characterizations of physical systems with some details omitted. Medium independence relies on this notion of a mechanism sketch that draws on scientific practice in biology and neuroscience.

Piccinini argues that there is a similarity between medium independence and multiple realizability because, in both cases, structural constraints determine whether the function can be realized in the physical system. For the mechanistic view, those constraints amount to possessing certain degrees of freedom. But the differences between the concepts are important. Multiple realizability comes from the idea that psychological processes are shared across multiple systems or kinds of systems and that those psychological processes produce distinct cognitive behaviors (Polger, Shapiro, 2016). Functionalism is an approach that was introduced to accommodate the multiple realizability of mental kinds. The functional organization is characterized abstractly, sometimes in terms of some mathematical formalism (CSA, FSA, Turing Machine, Algorithm, machine table, etc.). But functionalism is silent on how to connect the formalism with the physical world that it describes. This means that we need a theory of implementation. Of course, we want to understand implementation beyond cognitive computational descriptions, but the idea remains the same. But a switch to medium independence follows from Piccinini's rejection of functional analysis within the functionalism paradigm because he no longer needs the relation. So, he can rely on the concept of abstraction rather than explaining the relation between an abstract structure and the physical world. His theory instead, is firmly planted in the physical world.

Piccinini uses the concept of medium independence to develop his concept of a 'vehicle.' He uses the term 'vehicle' to refer to one of two things. It can refer to a variable or a state that can take different values and change over time. Or it can refer to a specific value of such a variable. He states that sometimes variables can be defined as purely mathematical, as symbolic entities, or as physical states that can change over time. But, for the mechanistic view, variables are just a way of referring to a *physical* variable (Piccinini 2015). Mechanistic vehicles have spatiotemporal parts. He points to a string of digits to exemplify what spatiotemporal parts are. A string of digits is a kind of vehicle made out of digits concatenated together. Digits are a portion of a vehicle that can be either binary (take one of two states) or continuous (a variable that changes continuously over time and can take on an infinite number of states).

Within the mechanistic framework, a rule is a mapping from inputs (and possible internal states) to outputs. Piccinini argues that the rule does not need to be represented within the computing system or spelled out in terms of any algorithm or by listing the inputs, outputs, or internal states. All that is required to give a rule is to specify the relation that ought to be obtained between inputs and outputs associated with a set of functionally relevant degrees of freedom. This notion of a rule is deliberately and substantially broad. Because Piccinini aims to capture *generic* computation, his notion of a rule is meant to accommodate computing systems that are not rule-based in any algorithmic sense. This is no surprise because this way of understanding a rule is developed by Horgan and Tienson (1989), who propose a syntactic argument for cognition, supporting the idea that computation is possible *without* rules.

The mechanistic view captures what Piccinini calls 'generic computation' in physical systems. Generic computation amounts to the processing of medium independent vehicles by a functional mechanism according to rules that are sensitive solely to differences between the spatiotemporal parts of the vehicles (Piccinini 2015). This description shows us that the implementation relation is completely gone. But not because it has been rejected. Instead, it has been completely dissolved.

4.3 Generic computation

Piccinini argues that we should abandon the idea that computation always involves information processing because the concept of 'information' has been confused in the literature. Instead, he thinks we should take on a theory-neutral understanding of computation that does not rely on information processing. Below is the third shift:

2. A theory of computation should not frame computation in terms of information processing.

Generic computation is meant to apply to all media and capture all relevant uses of computation, including classic computation, algorithmic computation, computation of Turing-computable functions, digital computation, analog computation, and neural computation.

Piccinini argues that the term ‘computation’ comprises different notions and that the same is true of information. Because information plays different roles in different fields, we should clearly distinguish between two central notions: Shannon information and natural and non-natural semantic information. He characterizes each in the following way. Shannon information originates from communication theory. Under this type, any device that produces messages in a stochastic manner can count as an information source or destination (Piccinini & Scarantino 2010). Semantic information pertains to what a specific signal broadcast means. Natural semantic information follows Dretske (1981). It is characterized in terms of correlations between event types where an event *A* carries natural information about an event token *b* of type *B* just in case *A* reliably correlates with *B*. Non-natural semantic information is characterized by following Grice’s (1957) non-natural meaning where a sentence is true even if the reference of the sentence does not obtain. For example, ‘those three rings on the bell (of the bus) mean that the bus is full’ is true even if the bus is not full.

Piccinini argues that each of these ways of understanding 'information' makes independent contributions to a theory of cognition, and the cost of conflating them results in a "mongrel" concept that is too messy to be useful because it obstructs any consensus on whether cognition involves computation or information processing. Further, he argues that the conflation of computation with information processes contributes to bad arguments because the notion of computation being used is never specified. And, because the notions are not specified, we are not justified in concluding whether the brain performs computations. As a solution, Piccinini argues for 'generic computation' because it does not rely on information or any specific type of computation.[62]

A view of this kind eliminates both internal and external semantics. This is much stronger than the non-semantic theories of implementation that implicitly accept the internal semantics posited by computability theory (or recursion theory). Internal semantics amount to a set of instructions that assert what the execution of a process accomplishes within the computer and are determined independently of any *external* features of the system. When a computer executes a function, it compares strings of digits within the system to produce the output. This process is self-referential; nothing within the system itself references anything external to the system to compute. Piccinini's mechanistic account is a strong non-semantic account because it explicitly denies that even an internal semantics is necessary for computation. This is, in part, a reason why the mechanistic account can accommodate computation without any information at all. An internal semantics is needed to process information computationally; thus, internal semantics can also be eliminated if the information is not required.

[62] There go two more babies.

Putting these ideas together, according to the mechanistic view, there are two reasons why neural processes count as computations. "First, the variables that are functionally most relevant to neural processing are spike frequency and timing; both appear to be medium-independent (Piccinini, 2018, p. 3)." Because nothing specific about the biophysics of the spikes themselves matters for the computation, they are medium-independent. "Spikes" are action potentials of neurons. Unpacking the concepts of spike frequency and timing will help make clear how to understand medium independence in this sense and then contrast it with the abstract entities that traditional theories refer to. The cell membrane of a neuron has a resting potential (the voltage of a neuron is approx. -70 mV). In addition, the neuron has three basic ions: potassium, calcium, and sodium, which exist inside and outside the cell membrane. When the neuron is resting, each of these ions has a particular concentration on the inside versus the outside of the membrane. When a neuron receives input and the threshold increases, the cell membrane depolarizes, lowering the voltage within the membrane. Ion channels open, the concentrations of the ions shift, and an action potential occurs. The depolarization spreads down the axon towards the axon terminals, where the synapse occurs. The action potential or firing of the neuron is propagated to the next neuron via the synapse, giving way to communication between the neurons. This firing is often called a *spike*. The spike frequency has to do with the stimulus the neuron receives. When stimulated with a constant stimulus, many neurons respond with a high spike frequency that then decays down to a lower steady-state frequency. Spike timing is a biological process that adjusts the strength of connections between neurons in the brain.[63] The process adjusts the connection strengths based on the relative timing of a particular neuron's output and input action potentials (or spikes).

Spike frequency and timing are medium-independent features of neurons because the frequency of the spike to the incoming stimulus has nothing to do with the physical features of the neuron itself in a way that appears to be relevant to producing the behavior in the neuron. This is the same for spike timing: spike timing does not depend on physical features of the neuron itself. Because spike frequency and timing function as medium-independent vehicles that are processed in accordance with a consistent input-output function, they are an instance of generic computation.

The above application is meant to tell us why neural processes count as computations; what makes them computational. But the story tells us nothing about whether the brain implements a computational structure. Moreover, it does not tell us which computation the brain performs. Thus, we do not have an answer to the implementation question, nor do we have an answer to the individuation questions.

4.4 Marr's tri-level hypothesis

Computational ascriptions are made from two directions: top-down or bottom-up. A bottom-up approach starts with data from the physical system, and then, from there, an abstract characterization of the system's behavior in terms of a computational model is generated. A top-down approach starts with an abstract model of a process within a physical system. In either case,

[63] Spike trains are obtained by detecting intra- or extracellularly the action potentials but preserving only the time instant at which they occur. They are obtained with multisite recordings from different septotemporal regions of the brain.

the abstract computational model might be used to describe what the system is doing in a literal sense. But whether we accept that computational ascriptions are true or not will have to do with our theory of physical computation. The mechanistic account, though, cannot answer these questions for us. But this is not a problem for Piccinini because he thinks we should understand Marr's levels mechanistically. This is the third shift:

3. Marr's tri-level approach should be understood as providing a mechanistic explanation where each 'level' is a mechanism sketch.

According to this interpretation of Marr, the levels are different aspects of the same mechanistic explanation. This interpretation of the Marrian framework integrates mechanistic functional analysis with a multi-level explanation.

Marr provides three levels to analyze an information processing system: computational, algorithmic, and implementation. The computational level states the goal of the computation. The algorithmic level gives a representation of the input-output and what algorithm is responsible for the transformation. The implementation or hardware level characterizes how the representation and algorithm are physically realized (Marr 2010). The computational and algorithmic levels are abstract in that they say nothing of the implementing mechanisms. However, Piccinini re-frames each level as a mechanism sketch. This is another place where the implementation relation is dissolved. Piccinini does this because he thinks "the computational level is a description of the mechanisms 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" (Piccinini 2015, p. 98).

He frames Marr this way because he thinks that Marr implicitly recognizes that each level is a different aspect of a mechanistic explanation based on "the way that he intermingled considerations from different levels in his theory of visual shape detection" (Piccinini, 2018). He argues that "this is how traditional computational explanations also work: that they [abstract] away from many of the implementing details… and that's how mechanistic explanation generally works; it focuses on the mechanistic level most relevant to explaining the capacity while abstracting away from the mechanistic levels below" (p. 98). But notice that there is a difference between abstraction in terms of a mechanism sketch and giving an abstract *description*.

Marr was providing a framework for how to provide explanations of complex *information-processing* systems. He never intended for the "levels" to be autonomous. He intermingled considerations from different levels in his theory of visual shape detection because, although some explanations will only be given at one or two levels, others will be given at all three to provide a complete explanation. This does not mean that he intended for the levels to be understood as a nested hierarchy in a mechanistic explanation. According to Marr, the relation between the levels includes logical and causal relations where structural considerations of the physical system will matter for which algorithmic explanation we provide. But these constraints do not amount to a mechanistic explanation at varying levels of abstraction. Marr characterizes mechanisms at the *implementation* level explicitly: we may approach the implementation level at different levels of *abstraction*, but we do not, therefore, shift to the higher level in the framework. When characterizing the visual system, Marr argues that "while algorithms and mechanisms are empirically more accessible, it is the top level, the level of computational theory, which is critically important from an information-processing point of view" (Marr 2010, p. 27).

He goes on to distinguish between *what* a system does and *how* it does it and argues that ignoring this distinction hampers progress between different fields. When discussing Chomsky's theory of linguistic usage, he says that "finding algorithms by which Chomsky's theory may be implemented is a completely different endeavor from formulating the theory itself… it is a study at a different level, and both tasks have to be done" (p. 28). Here Marr is arguing that we need to formulate the theory (provide an explanation at the computational level) and then provide a theory, an algorithm, for *how* that task is done (provide an explanation at the algorithmic level) and that both of these explanations are distinct from the mechanisms at the implementation level.

4.5 Concluding remarks

Ultimately, this new framework dissolves the implementation question and does not answer the interpretation question. Because of this, it cannot be a replacement for either view. But I don't think that this consequence is acknowledged in the literature. Piccinini's mechanistic account is always lumped in with Milkowski and Fresco's mechanistic accounts:

> "… the claim that ITTM computes is consistent with most accounts of computation—including… the mechanistic account (Milkowski 2013, Fresco 2014, Piccinini 2015)" (Shagrir 2022, p. 58).

> "Others require the implementation relation to have some *modal force*, meaning that it supports relevant *counterfactuals*. Even more, restrictions are placed by the *dispositional accounts*, and the *mechanistic account* (see Piccinini 2007 and 2015 & Milkowski 2013)" (Shagrir 2022, p. 129)

> "The mechanistic account (Piccinini 2007, 2015… Milkowski 2013, Fresco 2014…) avoids appealing to both syntax and semantics" (Piccinini & Maley, 2021).

> The attempt to use the tools provided by New Mechanism to account for computation in physical systems has been developed most forcefully by Piccinini (2007, 2015), Milkowski (2013), and Fresco (2014)" (Coelho Mollo 2018).

As we have seen, Milkowski's view and Piccinini's view are *very* different. Milkowski does not dissolve the implementation relation or eliminate an internal semantics. Fresco, on the other hand, does not provide a theory of physical computation in the first place.[64]

5 Engaging with the mechanistic account

Piccinini's mechanistic account is well known and has generated various replies. But, in the process, views sometimes characterize the mechanistic account as giving a theory of

[64] I think that a lot of what happens in the literature is that people are taken at their word when they state where their view belongs. So, when they say that they are giving a mechanistic account, then get slotted into the mechanistic account category. If they say they are giving a theory of computation, they get slotted into that category. And so on. But when we pull all these views apart and look at their pieces, we can see that it's not so simple and that just because someone says they are doing one thing doesn't mean they actually are. *So* much more is going on and so many people are just lumping and assuming. This can also make it difficult to engage in the debate in a way that challenges this framework and that attempts to clarify questions about physical computation and which theories address what.

implementation or “bridging the gap” between formal abstract computational descriptions and physical systems. As we know, the mechanistic account does *not* do this. This is only apparent when the conceptual shifts are made explicit. This means that when people engage with the mechanistic account, they generally do so *without* realizing the entire framework has shifted. I think a lot of this comes from not understanding the full context of how the concept of medium-independence functions in the theory. It cannot replace multiple realizability because it is not a new term for a familiar concept. It is an imported concept used as a guise for a familiar concept. If we take them to be similar concepts, then we end up with the illusion that the mechanistic account is *relational*. But, as I have argued, it is not. So, when people engage the view, most of the time, they are engaging a view that I will call “the mechanistic account*.” This view keeps the functionalist framework intact. Medium independence is understood in a multiple realization way, Marr still has abstract levels, and formal computations are still playing a role.

Adopting the mechanistic account* makes it tractable when comparing it to other theories that address physical computation. A consequence is that the mechanistic account, in its original form, is impenetrable to the arguments against it. It is not susceptible to the criticisms brought against it because the criticisms target a different view. If the goal is to argue against the mechanistic account, then you have to target the framework it’s built on. In the next section, I will give examples of how people engage with the view. The final example will be one where the author targets the mechanistic account directly, working within the conceptual shifts as Piccinini formulates them.

5.1 Targeting explanation

Haimovici (2013) argues that the mechanistic account faces a dilemma involving a tension between mechanistic and functional explanations. She says a good mechanistic explanation must describe a system's actual causal structure, which requires the identification of its actual functional and structural properties (p. 153).[65] Alternatively, for a computational explanation to be a good mechanistic explanation, we need a way to individuate computations based on structural and functional properties—meaning that multiple realizability would no longer be possible for computational systems (p. 159). The first horn is a problem for mechanistic explanation, while the second is a problem for computational explanation.

Haimovici says that "the goal of the mechanistic account is to build a bridge between formal abstract descriptions of computation and the functions and activities of concrete physical systems in such a way that it provides the means to distinguish whether a given system is computational or not empirically" (p. 154).[66] Here she argues that the mechanistic account is given mainly in functional terms, challenging whether it's a mechanistic explanation at all. She points out that, according to the mechanistic account, to implement a concrete computation, a physical system needs to exhibit certain degrees of freedom" (p. 157).[67] Possessing degrees of freedom means that the system needs to possess a certain number of physical variables that can vary independently from one another. She goes on "given the fact that the differences in computational vehicles that are functionally relevant to defining computations can be abstractly characterized, without

[65] Note that this reason also disqualifies Fresco's view as being mechanistic.
[66] The mechanistic account does not build a bridge because it does not answer the implementation question. Moreover, it is concerned with a non-platonist understanding of physical computation, which does not rely on abstract descriptions to identify computational systems.
[67] "Implement" here is being used loosely. It would be more precise to say something like "in order to ***perform*** a concrete computation…" – using the term "implementation" loosely in this ways gives the illusion that the mechanistic account addresses the implementation relation.

detailing all the specific physical properties of the vehicles, Piccinini holds that computational vehicles are medium-independent" (p. 157).[68] she concludes that, according to the mechanistic account, computational vehicles are medium-independent and are characterized mainly in terms of their functional properties rather than their structural ones. That "computational vehicles are abstract and multiply realizable in different physical media, e.g., mechanical, electromechanical, electronic, or magnetic" (p. 159). She thinks that this is a *functional* explanation and not mechanistic.[69]

What Haimovici targets is the use of mechanism sketches to support medium independence. Recall that according to the mechanistic account, functional analysis is to provide a mechanism sketch and that this mechanism sketch allows Piccinini to incorporate medium independence into his view. She thinks this generates a tension between mechanistic explanation and functional analysis for individuating computing systems because mechanism sketches are weak in explanatory power because they omit details and are thus an insufficient explanation. Put differently; she thinks that adopting a mechanism sketch as the appropriate level of explanation goes against the mechanistic orthodoxy that "good mechanistic explanatory texts describe all of the relevant components and their interactions, and they include none of the irrelevant components and interactions" (Craver 2007, p. 140).[70]

[68] This trades "abstraction" for "abstract" characterization. Medium independence is born from the process of abstraction while multiple realizability is born from the abstract versus physical details distinction.
[69] This takes medium independence and multiple realizability to be the same when they are not. When we distinguish abstraction from abstract entities, we can see how the two come apart. Also, consider how Piccinini understands functional explanation—he thinks he is doing that (under his reconceptualized framework)—what Haimovici is describing here is what Piccinini calls "task analysis."
[70] But consider how Marr is revised in Piccinini's account. A mechanism sketch, although medium-independent, is *still* part of the nested hierarchy. It's mechanisms from top to bottom. It seems that Haimovici has the intuition that Piccinini is giving an explanation that ends up falling at the algorithmic level. But this intuition is only generated when we understand Marr's levels as he presented them. If we use Piccinini's new framework, the algorithmic level is just another mechanistic level—it's just the implementation level all the way up.

The second horn is that there is a problem for computational explanation. Haimovici focuses on the concept of multiple realization in this section and follows Weiskopf's (2011) view that realization is a relation between properties at different levels of organization (Haimovici, 2013, p. 169). She adopts this view because it pairs nicely with the mechanistic account for several reasons. First, Weiskopf uses a mechanistic criteria to distinguish different realizers of functional kinds by appealing to differences in the structural properties of the components, such as their size and location. Second, this account focuses on properties or kinds described in the same way that Piccinini employs to characterize computational systems, that is, in terms of their functional properties, by concentrating on the relation between functionally characterized kinds and the different mechanisms that instantiate those kinds. Finally, Weiskopf proposes identifying kinds in terms of their roles in explanatory schemas, grounding the notion of realization and the criteria for evaluating cases of multiple realizability in scientific practices (p. 169). Adopting this kind of realization means that cases of multiple realization need to meet two conditions: systems with different structural properties must satisfy the same functional description, and functional descriptions must individuate a kind in virtue of being of explanatory use in the pertaining sciences. Once the structural details are filled in, the result will be a description of a particular mechanism that implements a computational system, not a description of a computational system that is multiply realizable (p. 171).[71] [72] She thinks that result of this is that it makes multiple realizability incompatible with computations.[73]

5.2 Targeting the mappings

Coelho Mollo (2018) tries to improve on the mechanistic account. He thinks a tension between the "New mechanists" and the mechanistic account of computation needs to be resolved. The New mechanists take mechanistic explanation to be the following:

> Mechanistic explanation proceeds by individuating the underlying components and activities that form the mechanism, as well as their organisation, unveiling how they bring about the phenomenon to be explained. This often involves nested mechanisms—components of mechanisms are themselves mechanisms that can be decomposed into components, which might in turn also be decomposable mechanisms, until a level is reached in which mechanistic decomposition is no longer possible. This leads to the multilevel nature of mechanisms and mechanistic explanation (p. 3480).

He then goes on to give the mechanistic account of computation:

[71] Again, I caution the reading of "implementation" here.

[72] Haimovici focus is on multiple realization. This follows from the previous claim that what Piccinini is doing is actually giving a functional analysis. If we are thinking of functional analysis in the way that we typically do, then it makes sense to rely on multiple realization. But, as we know, medium-independence and multiple realization are not interchangeable concepts. Thus, her focus should be on medium independence. If it were, it's not clear that this criticism would go through.

[73] Piccinini (2015) replies to Haimovici in the following way. "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 level(s) of mechanistic organization (Piccinini 2015, 124-125).

Computational mechanisms are a type of teleolofunctional mechanism. Teleofunctional mechanisms are mechanisms that have teleological functions, that is to say, they have purposes or ends. Computational mechanisms, according to Piccinini are a subset of teleofunctional mechanisms (p. 3481). …Concrete computation, in its turn, is defined as the manipulation of vehicles according to a rule sensitive only to (some of) their physical properties. A rule, finally, is a mapping from inputs (and possibly internal states) to (internal states and) outputs. Vehicles and their activities are arrived at by means of mechanistic decomposition. …One important property of vehicles in concrete computation is their being medium-independent. The rules that govern the changes undergone by the vehicles are sensitive only to some dimensions of variation of their physical properties, i.e. their degrees of freedom. Degrees of freedom abstract away from the physical properties themselves—consisting only of their dimensions of variation—and are thus characterized in medium-independent fashion (p. 3482).

He, like Haimovici, thinks that the difference between these two presents a dilemma for the mechanistic account:

1) Good mechanistic explanation tends toward full structural detail at all levels of the mechanism.
2) Computational explanation is necessarily abstract, insofar as it ignores most structural detail, caring only about degrees of freedom (p. 3483).

As we know (footnote 20), Piccinini rejects one. Coelho Mollo agrees with Piccinini and also rejects one. But he says that doing so comes at a cost: it avoids the dilemma but gives up the mechanistic view (p. 3484).

Coelho Mollo argues that a good mechanistic explanation requires including all structural details, or it collapses into a functional explanation. He thinks that almost all structural detail is left out of the mechanistic account, leaving only constraints on the degrees of freedom of the structural components and that the weak structural constraint remaining are on par with the structural constraints posed by functional explanation. He says, "the abstraction from details that is an essential characteristic of computational explanation is comparable to the abstraction from details found in functional explanations"… and that providing any further structural details… "would immediately foil any attempt at multiple realizability or medium-independence"… "computational explanation and functional explanation look therefore much alike" (p. 3485).[74]

Coelho Mollo argus that he can solve this problem by viewing the mechanistic account as a *hypothesis* about the nature of physical systems that are able to implement computations in a robust, non-trivial way. We would say that the role of mechanisms in the mechanistic view is to provide part of the connection between abstract computation and the world that a theory of computation must deliver (3487). What Coelho Mollo is advocating for is that we incorporate a *relational* view into the mechanistic account. His improved version is the following view:

[74] I think that this is a fair criticism, but I think that it targets medium-independence and not functional explanation. This relies on a different conception of functional analysis than Piccinini accepts. So, it seems that what Coelho Mollo has to say is that Piccinini is doing something like task analysis.

> The appeal to mechanism in the mechanistic view plays an analogous role to mapping relations in the simple mapping account…it provides the bridge between functional computational individuation, and implementation in physical systems. But it provides further constraints than its competitors: it includes mapping as well as causal considerations, into a richer, more constrained, picture. The mechanistic view requires that physical computational systems not only have physical states mappable onto abstract computational states, as per the simple mapping view, or that they be causal systems, as per the causal view. They must be more than that, they must be mechanisms—organised systems with relatively clear boundaries, decomposable into physical parts that play a role in bringing about the overall behaviour of the system; and they must be teleofunctional mechanisms with the teleological function of performing computations (p. 3490).

But notice something important: in order to do this, Coelho Mollo *must* adopt the mechanistic account* because the mechanistic account would never be compatible with a mapping view in this way—they rely on two fundamentally different frameworks! Essentially, this takes a Milkowskian approach to a mechanistic account of physical computation. It includes a mapping relation and a further specification of mechanisms—it incorporates mechanisms into a theory of implementation. Thus, this view now counts as a non-semantic theory of implementation, but critically, the mechanistic portion of the view is *not* the mechanistic account, it's necessarily the mechanistic account*.

5.3 Targeting the mechanistic account directly

Maley (2022) argues that the mechanistic account does not work as an account of computation. But he takes a different approach from the previous two arguments. He targets the view *within* the framework offered and thus targets the view and the elements that make it up. To target the mechanistic account, he chains together issues that arise between the various concepts required to build the view. He argues that there are problems regarding how the concepts are meant to fit together.

First, Maley addresses functions. One subset of mechanisms with functions includes those that perform computations. To demonstrate how the mechanistic account works, Maley gives us an example of a calculator:

> A calculator is a mechanism, and it has teleological functions. One of those teleological functions is to perform calculations. Are calculations instances of generic computation? Yes, because there are mappings from inputs to outputs via internal states that follow a rule, sensitive only to differences between voltage levels in circuit elements. So how does it perform these computations? In a typical electronic calculator, this is done8 by appealing to the Boolean operations performed on circuits with binary values. At this level, the functional mechanism is only sensitive to differences between two voltage levels of the circuits, and follows rules that map inputs (and internal states) to outputs (p. 7).

Maley argues that a problem with this account is that it ends up overgeneralizing, and too many things end up being counted as computational. To show this, he evaluates the details of the mechanistic account, starting with the concept of '*rules*.' Maley argues that Piccinini's construal of 'rule' is too broad and leads to triviality. To demonstrate, he considers a common lawn sprinkler:

> This device follows rules in Piccinini's sense. In particular, it maps inputs and internal states to outputs. The input is liquid at a certain pressure, the internal state is the current position of the nozzle, and the output is liquid at a certain pressure and velocity. The output of this particular sprinkler is approximately) a periodic step function (in angular degrees) for a given input pressure. Thus, the sprinkler unambiguously satisfies the MAC construal of a rule (p. 9).

According to the mechanistic account, the sprinkler counts as a computer. But sprinklers aren't computing when they spray water, so this notion of 'rule' makes the account too liberal. Moreover, Maley argues that how these rules apply to *vehicles* is also a problem.

According to the mechanistic account, computation involves the processing of vehicles where a vehicle is understood as a variable or a value of a variable. Connecting this notion with the idea of *generic computation*, vehicles are the kinds of things that have spatiotemporal parts, implying that they are themselves spatiotemporal. Moreover, these vehicles are *medium-independent*. We know whether the vehicles have medium-independence when the rule is sensitive to only some spatiotemporal parts but not others. Maley argues this falls apart when applied to natural systems like neural firings. Consider Piccinini & Bahar's (2013) claim about why neuronal systems compute:

> The functionally relevant aspects of spike trains, such as spike rates and spike timing, are similar throughout the nervous system regardless of the physical properties of the stimuli (i.e., auditory, visual, and somatosensory) and may be implemented either by neural tissue or by some other physical medium, such as a silicon-based circuit. Thus, spike trains—sequences of spikes such as those produced by neurons in real time—appear to be medium-independent vehicles, thereby qualifying as proper vehicles for generic computation. Analogous considerations apply to other vehicles manipulated by neurons, such as voltage changes in dendrites, neurotransmitters, and hormones (Piccinini & Bahar, 2013: 462).[75]

Maley maintains that nothing about medium independence follows from this case. This is because medium independence has to do with the rules and their sensitivity to certain differences in the properties of the vehicles in question. But there is no difference between neural firings caused by visual stimuli and those caused by auditory stimuli (Maley 2022, p. 12). Additionally, medium independence doesn't follow from the fact that spike trains can be implemented in neural tissue or some other physical medium. He thinks that this entails *multiple realizability* rather than medium independence; acknowledging the difference between the two.

Maley argues that introducing semantics into the account is a solution to these problems. On this view, physical computing mechanisms are those systems with mechanisms that process physical representations, where the mechanism is sensitive only to the properties of the representations that are responsible for doing the representing (p. 22). Note Maley's final statement before closing this paper:

[75] Cited by Maley on p. 11.

> By coupling this part of the mechanistic account with a version of a semantic account, we can develop a new account of physical computation that prioritizes physical representations. But that story is a longer one to tell, and must wait for another day (p. 25-26).

Maley gives us a quick sketch of what he has in mind:

> At a very abstract level, compilers simply take a set of strings as input and produce as output another set of strings (this is what all software does, considered abstractly enough). More concretely, however, compilers take code written in a high-level language and produce assembly code: crucially, however, those strings are representations. An input string, such as
>
> ```
> printf("Hello, world!\n")
> ```
>
> represents an instruction in C++; the corresponding lines of assembly code produced as output represent instructions for the relevant instruction set architecture. If the input and output did not represent instructions, then this would simply not be a compiler (p. 21-22).

I'm going to speculate on how this view might be developed. If Maley remains consistent with this example, he will give a semantic view in terms of *internal* representations or internal semantics. This means that his semantic-mechanistic account is not a "semantic account" in the way that theories of interpretation are semantic. Thus, it would still be a non-semantic mechanistic account, just a weaker version than Piccinini's. Additionally, it's worth considering whether Maley will adopt all the baggage that comes with the mechanistic account. I have some reason to suspect that he won't do this, so I will assign Maley's non-semantic mechanistic account the asterisk. In fact, for reasons that will unfold in the next chapter, I think this is the *best* way for Maley to develop his view.

6 From mechanisms back to mappings

At the 2022 Philosophy of Science Association Biennial Meeting, Piccinini and Anderson presented what they referred to as a "robust mapping account."[76] In this presentation, they presented the following account:

> Physical system *PS* implements computing system *CS* if and only if:
>
> 1. *PS* satisfies a physical description (a PDPS) that
>
> 2. Maps (at least) robustly onto the computational definition (CDCS) that defines *CS*.

They think that this view will rule out pancomputationalism, accommodate conventional and unconventional models of computation in both natural systems and artifacts, and can be incorporated in accounts of implementation that include additional considerations (e.g. mechanistic and semantic accounts). What they then went on to present was a bonafide non-

[76] This presentation was part of a symposium that included papers from me (presenting the distinction between implementation and interpretation and the role of triviality!), Gualtiero Piccinini & Neal Anderson, David Barack, Paula Quinon, J. Brendan Ritchie, and Pawel Stacewicz.

semantic mapping account.[77] This is a tacit admission that the mechanistic account, on its own, cannot address questions about implementation, nor can it address questions about interpretation. I also imagine that Piccinini means to reference *his own* account in the parenthetical listed above. If this is the case, his view will need some conceptual revisions. This is, of course, a welcome change for me.

In the final chapter of this dissertation, I will draw the connections between theories of implementation, theories of interpretation, and mechanistic accounts to show how they pair quite nicely with each other. This is something that, given the above admission, even those that once seemed to be the most hard-pressed to reject a compatibilism between the views might be open to adopting.

[77] I have discussed my dissertation project with Piccinini on several occasions. He has also seen my poster presentation that lays out how his view shifts the direction of the literature. I hope that some of my work influenced the emergence of this new view. The manuscript is under review, so only time will tell how this unfolds in the paper.

8 Aspects of a theory of computation

1 Introduction

In chapter three, I argued that when we evaluate computational theories, we can distinguish three questions: individuation, implementation, and interpretation. I argued that non-semantic accounts answer the implementation question while semantic accounts answer the interpretation question. The locus of the debate is often framed in terms of non-semantic versus semantic views, but I argued that this was a mistake when we make it clear that they are answering different questions. Sometimes this debate is framed in terms of this question. I derive the individuation question from Piccinini (2015) and Shagrir (2022) because they both frame the debate in terms of the individuation question, including, sometimes, applying it retroactively to older views. I argued that this way of framing the debate supports the idea that semantic and non-semantic views (and even a mechanistic account) are alternatives to a theory of computation and should be rejected. However, we can still accept the individuation question as part of a theory of physical computation. One can maintain their metaphysical commitments regarding that question while still adopting my framework that distinguishes between implementation and interpretation.

In chapter four, I argued that two types of triviality are central problems when it comes to theories that address physical computation. I identified these two types as a trivialization of the implementation conditions and a trivialization of the interpretation conditions. Both types of triviality can emerge independently from distinct arguments or arise from the same argument. I have also shown that some writers are concerned with one type while giving reasons to dismiss or accommodate the other. This is a reasonable stance, but it does not eliminate the fact that the two types of triviality each warrant an independent response. Toward this end, I have argued that each type of triviality should be articulated as a separate problem that requires its own independent answer.

In chapter five, I argued that theories often just classified as "non-semantic" theories of computation always address the implementation relation. This is *not* the claim that you must give a non-semantic theory to address the implementation relation. Instead, it is *observation*—it is how the literature has unfolded. But you might agree that this trend demonstrates that it's possible to characterize the implementation relation non-semantically without bringing in additional explanatory posits, such as a reference to external semantic content. I have also argued that in virtue of addressing the implementation relation, these views also respond to the problem of trivial implementations. Implementation views come in different forms and place more or fewer restrictions on the physical system. Whether any of them is a successful account of implementation is up to the reader's preference, but we can all agree that they are addressing the relation between a computational architecture and an implementing physical system. I also argued in chapter seven that it is possible to give a theory of implementation within the mechanist framework. So long as the view maintains the relational nature of the implementation question, it still counts as a theory of implementation.

In chapter six, I argued that "semantic accounts" address a second aspect of physical computation, namely, interpretation. I noted how the individuation question is most obviously derived from semantic accounts because some involved in the debate answer it explicitly. Semantic theorists offer a semantic account because they take on the metaphysical commitment that computation necessarily (or essentially) involves the manipulation of representations to some degree. But I argued that once they develop their account, they end up providing a theory of interpretation that addresses the problem of trivial interpretations and that we should distinguish their answer to the individuation question from their theory of interpretation.

In chapter seven**,** I argued that mechanistic accounts take very different forms. I showed how it's possible to give what I call a "weak" mechanistic account that addresses the implementation question where the additional constraints on the physical system are cashed out in terms of mechanistic explanation. This type of view qualifies as a non-semantic theory of implementation. The main part of chapter six was dedicated to showing how the mechanistic account given by Piccinini (2015) reconceptualizes the very nature of what a theory of computation is meant to address. I argued that if we accept his view as given, it is incompatible with any implementation and interpretation view because the need for such a theory is dissolved within the new framework. But, as I will show later in this chapter, it is possible to adopt a mechanistic account and *reject* the conceptual framework that it rests upon. But, when we do this, we accept a view that I call "the mechanistic account*." I have speculated that Maley might consider something like this when he proposes the semantic-mechanistic account.[78] The good news is that the mechanistic account* is compatible with theories of implementation and interpretation. In this chapter, I will propose two ways to integrate the mechanistic account* into a theory of physical computation that addresses implementation, interpretation, and a specification of the mechanisms.

[78] But rejecting the framework doesn't mean accepting the traditional one. For example, Maley has argued that Marr's representational (algorithmic) level and implementation level should be collapsed to account for analog computation. So, again, I caution that I am merely speculating here about how Corey's account will turn out.

By restructuring the literature and specifying distinct questions that target different aspects of physical computation, I don't mean to claim that I've identified *everything* we need to address when it comes to physical computation. I also am not saying that trivial implementations and interpretations are the only kinds of triviality problems that arise when addressing physical computation. In recent literature, the issue of triviality has become more nuanced and is now sometimes framed as a problem of indeterminacy. Indeterminacy problems can take different forms, and it might be possible to address them within my framework by identifying whether they should be addressed by a theory that targets implementation, interpretation, or mechanisms.[79] But some forms of indeterminacy might be better handled by additional theoretical posits that draw connections between a theory of computation and its applications to the brain along with considerations about methodologies in cognitive science. My future work will target computation in this domain.

In section two, I will show how my framework has two possible forms. But first, I will argue that if we adopt the mechanistic account, we sacrifice implementation and interpretation for a view that addresses the individuation question with a full-blown theory. In section three, I will build a theory of computation using my framework. People may have different preferences about what theories they pair up based on their commitments, but I will show how it can be done using my preferences as an example.

2 A framework for a theory of computation

[79] For example, see Papayannopoulos, Fresco, and Shagrir (2022).

In this section, I will argue that there are three ways that we can conceive of the relationship between the individuation question, a theory of implementation, interpretation, and mechanisms. I will provide three schemata for formulating a theory of computation. The first is the most extreme because it relies solely on the mechanistic account. I will reject this schematic as an appropriate formulation on the basis that a theory of computation is insufficient if it does not address implementation and interpretation. The following schemata will adopt the mechanistic account* into the framework. To review, my framework involves the following definitions:

> **The Individuation Question:** How are computing systems distinguished from non-computing systems?
>
> **The Implementation Question**: What are the conditions under which a physical system implements a computational structure?
> **Trivial implementation**: Too many or all physical systems can be seen as implementing some computational structure.
>
> **The Interpretation Question**: Which computational process(es) does the physical system perform?
> **Trivial interpretation**: A physical system can be interpreted as performing every computational process.

2.1 Mechanisms-only

If we accept the mechanistic account, then our theory of computation would include a mechanism-only theory of computation because the framework dissolves the implementation relation and eliminates the need to interpret the system:

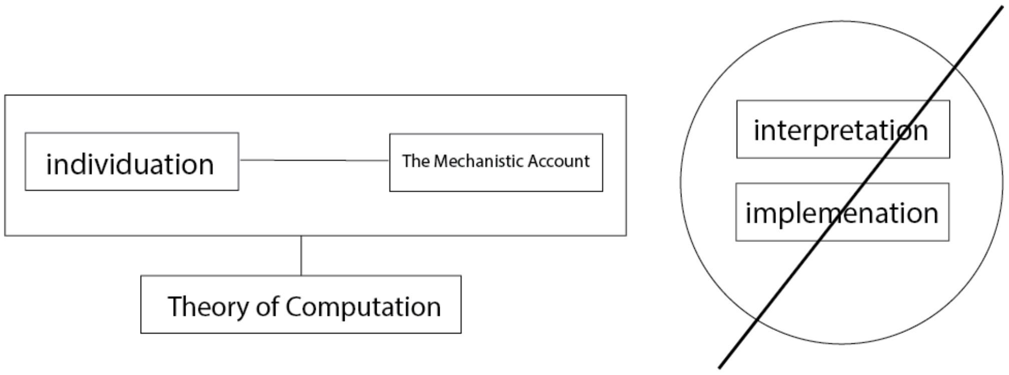

Figure 5. Mechanism-Only Schematic

In its original form, we can understand the mechanistic account as answering the individuation question by providing a full-blown theory. I reject this way of giving a theory of computation, so I will set this aside. We also have reason to believe that Piccinini himself sees this picture as an unviable theory of computation given his newest project with Anderson (2023, in progress) that brings the implementation relation back into the picture.

2.2 Three questions + three theories: version one

Another way that we can conceive of the literature is to take the mechanistic account* as addressing the individuation question. This gives us a full-blown theoretical answer to the individuation question—something that is missing from accounts that fork into a theory of implementation or individuation. This takes a metaphysical claim and backs it up with an actual metaphysical theory. This schematic requires answers to the implementation question and the interpretation question as well. But this formulation gives up the opportunity to provide independent answers to the individuation question. So, for example, if you think that an answer to the individuation question necessarily requires some reference to semantic content, then you may reject this framework. The following graphic demonstrates version one of the schematic:

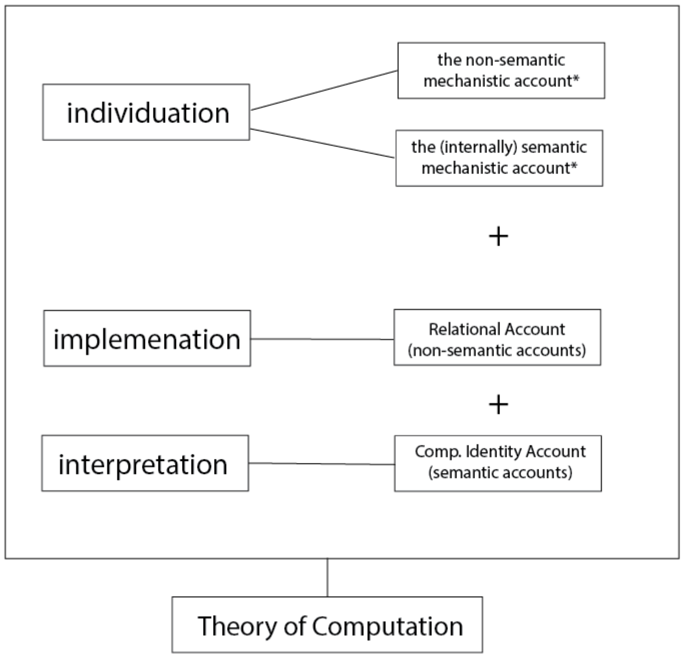
individuation
the non-semantic mechanistic account*
the (internally) semantic mechanistic account*
+
implemenation
Relational Account (non-semantic accounts)
+
interpretation
Comp. Identity Account (semantic accounts)
Theory of Computation

Figure 6. Three Questions + Three Theories Schematic

This graphic shows that there are independent answers to each question. There are two options available to answer the individuation question: the mechanistic account* and Maley's semantic mechanistic account*. You could adopt one or the other depending on whether you think it's possible to answer the individuation question without at least an internal semantics. Then, an answer to the implementation question and an answer to the interpretation question are added.

Together, all three forms a theory of computation that addresses different aspects of physical computation.

If you are inflexibly committed to the idea that an answer to the individuation question necessarily requires reference to (external) semantic content, then you will reject this schematic. But accepting this version does not mean that one has to give up their commitment to the idea that computation is necessarily a semantic process. Many who give an interpretation view think that semantic ascription is a matter of context or user dependence. This means that the representational content is not inherent to the physical system. If this is the case, it's perfectly reasonable to accept something like Maley's mechanistic account* as an answer to the individuation question because semantics is taken care of when the theory of interpretation is in place. Put differently; the individuation question does not have to do the heavy lifting when it comes to supporting the idea that computation involves the manipulation of representations. If we accept that there are three questions with three answers, then an external semantics can enter via any theory because what matters is that *it's accounted for* in a theory of computation. It gets pinned to the individuation question mainly because the literature does not distinguish between these theories, so those who provide a semantic account associate semantic content with individuation. Under this new framework, the interpretation view can be the vehicle instead.

2.3 Three questions + two theories and an option

The third schematic leaves the individuation question alone and allows for whatever metaphysical claim one wants to give. The mechanistic account* comes in to possibly beef up an implementation view. You can read this version in two ways. The first is that the mechanistic account* should be part of a theory of computation; without it, the theory would be incomplete. The other version lets one leave the mechanistic account* out of the theory altogether. If you think that implementation and interpretation is all there is to a theory of computation, then you can choose to sideline the mechanistic account*. One reason to do this is that you might want to adopt Milkowski's mechanistic implementation view. If you slot his view into the implementation spot in the schematic, then the mechanistic account* becomes redundant. The following graphic shows version two of the schematic:

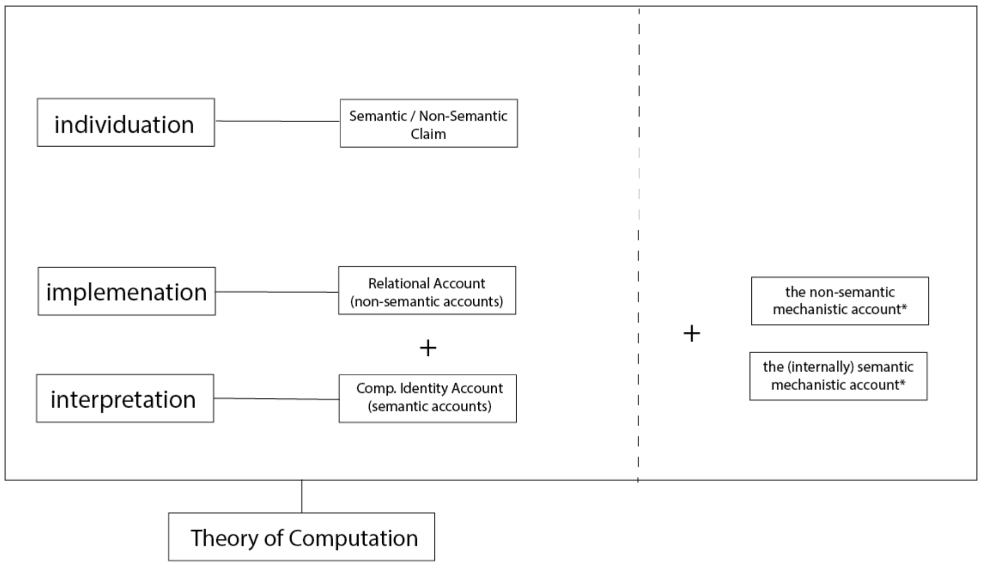
individuation
Semantic / Non-Semantic Claim
implemenation
Relational Account (non-semantic accounts)
+
interpretation
Comp. Identity Account (semantic accounts)
+
the non-semantic mechanistic account*
the (internally) semantic mechanistic account*
Theory of Computation

Figure 6. Three Questions + Two Theories and an Option Schematic

The dotted lines represent the option to take or leave either of the mechanistic accounts* depending on how you want to piece the theories together. The schematic allows one to answer the individuation question however they would like, but it's not required (hence no '+' sign). A benefit of this is that if you reject even the mechanistic account*, you can keep it out completely.

Moving forward, I will adopt this version of the framework because I think it's possible to answer the individuation question however you would like (including not at all) and then adopt an implementation and interpretation view. I think it's possible to eliminate the individuation question, but I take it that many people will want to retain the right to their answer, so I will leave it in. Finally, I think it is important to leave an option to include or reject the mechanistic account*. This is because I believe that you can adopt a mechanistic theory of implementation, making the additional mechanistic account redundant. You might also want to reject the view for other reasons.

3 Building a theory of computation

In this section, I will give an example of how we can pair different theories to show how they can be brought together to form a more complete theory of physical computation. Because I distinguish the individuation question from a theory of implementation and interpretation, you can adopt a theory without taking on the associated answer to the individuation question. This also means that if you have proposed either an implementation or interpretation view, and you are pairing your view with the other, you do not need to change your answer to the individuation question.

I find mapping views difficult to understand when it comes to biological systems, so my preference leads me toward a computational complexity account rather than a mapping account that relies on some morphism relation between states or structures. Given this, I'll take on Scheutz's bisimulation view. He answers the individuation question by saying that for something to qualify as a computer, it has to be at least a usable, physical system that allows for data input and a measurable output, which works within reasonable time constraints and is sufficiently reliable. This is a loose definition that can be paired with further commitments like ‘executing an algorithm.’ I will adopt Scheutz’s claims here as an answer to the individuation question because it gives us some advantage in addressing trivial implementations. For example, it immediately eliminates systems like walls and pails of water.

Scheutz’s theory of implementation relates formal computational structures to a physical system by starting with the physical system and progressing through various levels of abstraction until an abstract behavioral explanation can be given. This relates a formal computation to a physical system by moving “up” from the physical world rather than moving “down” from abstract descriptions. This approach shows how a physical theory can be used to distill a mathematical mapping between the inputs and outputs of a physical system by abstracting over the physical features. There is still some work to be done if we want to extend this theory to biological computing systems like the brain. Still, I think that relaxing the mapping relations between the formal description the physical states is important. I also think that focusing on the system's behavior—i.e., the inputs and outputs—is an important part of the process. I take it that some of what will come out of cognitive science and neuroscience as we learn more about the physical features of the brain, their nature, and how they relate to cognition will help us to complete this theory. If we want to develop it into an account that supports computationalism.

This means that I don’t think that a theory of implementation necessarily needs to be able to identify all computing systems *right now*. If you think it should, you might opt for a different view. I think that it probably captures the obvious cases—the artifacts. But the difficult cases like biological computation and neural computation will not be as straightforward. What I do think, and that is beyond this project's scope, is that any view that requires a mapping from formal states to physical states will not work when it comes to the brain. Issues pertaining to neural reuse, which cells should bear the mappings, what role physical features play in computational processes, etc., all matter for drawing any relation. I am skeptical of the success of mapping views beyond our artifacts, so I will likely reject any mapping view when it comes to implementation. But we can see immediately that Scheutz's view works toward addressing trivial implementations. So, even though I think there's more to be done with this view, I think it addresses the problem of trivial implementations sufficiently.

I will adopt Shagrir's theory of interpretation because his view naturally lends itself to a pairing with an implementation view. But I reject his answer to the individuation question. I think that computation does not require an external semantics, but it does require an internal semantics. For Shagrir, whether a system performs one computation or another depends on the environment in which the system is embedded. I agree that context matters, but I think some additional considerations should be added to Shagrir's view. For example, Shagrir (2006) considers the Zipser-Anderson model, which explains how cells combine information from retinotopic coordinates with information about the eye orientation in area 7a of the brain. Zipser-Anderson identified the following equation to explain the process of integration:

$$p_i = (k_i^T e + b_i) \exp(-(r - r_i)^T (r - r_i)/2\sigma^2)$$

Figure 8. Zipser – Anderson Equation

In the equation, *e* stands for eye orientation with respect to two angular components, k_i and b_i that represent the cell's gain and bias parameters, r_i is the center of the activity field in retinotopic coordinates and σ describes the width of the Gaussian (p. 405). Zipser and Anderson relied on data from anatomical and physiological experiments on macaque monkeys. They used previously acquired data from recording neuronal activity extra-cellularly during various visio-spatial tasks (Zipser & Anderson 1988). The neuronal recordings from twelve retinal receptive fields were arranged according to peak eccentricity and complexity and plotted on a series of topological graphs.[80] From there, they used a three-layer network trained to map visual targets to head-centered coordinates, given any arbitrary pair of eye and retinol positions (p. 681). They concluded that a striking similarity between model and experimental data supports the conjecture that the cortex and the network generated by back propagation compute similarly (p. 684). Based on the context and the expertise of Zipser and Anderson, Shagrir accepts that the brain computes the above function when performing integration.

But there is an additional model offered to explain the very same phenomenon. Mazzoni et al. (1991) offer a neural network model, but instead of relying on the backpropagation algorithm, they used an $A_{R\text{-}P}$ learning algorithm (associative reward-penalty algorithm) to model area 7a. After training the network, it learned to perform the transformation task to the level of desired accuracy. This model showed that the cells behave in accordance with the equation:

$$\Delta w_{ij} = \rho r(x_i - p_i)x_j + \lambda\rho(1 - r)(1 - x_i - p_i)x_j,$$

Figure 9. Mazzoni et al. Equation

[80] For a more detailed description and visual representation of the graphs see Zipser & Anderson 1988, pgs. 680-681.

In this equation, the first term in the sum computes the reward portion of the learning rule, and the second term is the penalty portion. Mazzoni et al were motivated to develop a model based on the $A_{R\text{-}P}$ algorithm because it is more *biologically plausible* than the backpropagation algorithm used by Zipser and Anderson.

What about this additional function? The context difference has to do with the goals that influence model development. Zipser and Anderson relied on the backpropagation algorithm to develop their model while Mazzoni et al. relied on a more *biologically plausible* algorithm.[81] If biological plausibility is a modeling goal, then you might end up with a completely different description of the same phenomenon—this means that different interpretations can result from the same context. Given this, I think Shagrir's view needs further specifications to determine the context. I prefer biologically plausible models, so I would choose the Mazzoni et al. model.

I also think that it can be helpful to adopt the mechanistic account* to help strengthen the implementation view. I reject that computation is possible without something like an internal semantics, so I will adopt Maley's mechanistic account*. The semantic mechanistic account* can help to identify the parts of physical systems that count as computational. I mentioned how it's not clear which structures in the brain we should be mapping to, and it's not clear which physical parts are necessary features for performing computations. I reject a mapping view, but these questions

[81] It has been argued that while back propagation is very successful when it comes to training neural networks, that it's not biologically plausible that the brain does something like back propagation (Song et al., 2020).

are still important. So, when claiming that particular physical features of the brain realize a computational structure, a mechanistic account* can help us identify which mechanisms we are referring to. Advancing Scheutz's view beyond where he left it above.

4 Conclusion

In this project, I have argued that we should hold three questions distinct when it comes to a theory of computation: individuation, implementation, and interpretation. How one answers the individuation is distinct from a theory of implementation and a theory of interpretation, although it will very likely impact which type of theory you propose when it comes to physical computation. I have shown how typically the implementation relation is given by a non-semantic view. I leave open whether there could be a semantic view of implementation, but I am skeptical of a project that attempts to do this. It seems to me that as soon as an external semantics enters the theory, the view ends up addressing interpretation and *not* implementation. I also leave open whether it's possible to give a non-semantic theory of interpretation, but I will say that it doesn't seem to make any sense that a theory of interpretation would not rely on semantic interpretation in some form. So, while it might be possible to give these theories, I imagine the literature has gone the way it has because it doesn't make much sense to do otherwise. This is only apparent when we pull implementation and interpretation apart, though. When we don't do this, it *appears* that the debate is one about semantic versus non-semantic views. But, as I have shown, if the disagreement is located anywhere, it's found in how one answers the individuation question. Further debates may also be found between the different implementation views and between the different interpretation views, but in the end, a theory that addresses each question is *at least* needed to address physical computation. My framework shows how we can accomplish this task.

WORKS CITED

Adriaans, Pieter. 2020. *Information*. https://plato.stanford.edu/archives/fall2020/entries/information/.

Angius, Nicola, Primiero Guiseppe, and Raymond Turner. 2021. *The Philosophy of Computer Science*. Spring. https://plato.stanford.edu/archives/spr2021/entries/computer-science.

Armstrong, D. M. 1968. *A Materialist Theory of Mind*.

Blanchowicz, James. 1997. "Analog Representation Beyond Mental Imagery." *The Journal of Philosophy* 94 (2): 55-84.

Block, Ned. 1978. "Troubles with Functionalism." In *Perception and Cognition: Minnesota Studies in the Philosophy of Science*, by Wade C. Savage. Minneapolis: University of Minnesota Press.

Block, Ned, and Jerry Fodor. 1972. "What Psychological States Are Not." *Philosophical Review* 81 (2): 159-181.

Bradbury, J. W., and S. L. Vehrencamp. 2000. "Economic models of animal communication." *Animal Behavior* 59 (2): 259-268.

Bromley, Allan G. 1987. "The Evolution of Babbage's Calculating Engines." *Annal of the History of Computing* 2: 113-136.

Brown, Curtis. 2012. "Combinatorial-State Automata and Models of Computation." *Journal of Cognitive Science* 13 (1): 51-73.

Care, Charles. 2006. "A chronology of analogue computing." Coventry: Department of Computer Science, University of Warwick.

Carnap, Rudolf, and Bar-Hillel Yehoshua. 1952. "An Outline of a Theory of Semantic Information." Technical Report 247, Research Laboratory of Electronics, Massachusetts Institute of Technology, Cambridge.

Carnap, Rudolf, and Y. Bar-Hillel. 2952. "An Outline of a Theory of Semantic Information." Research Laboratory of Electronics, MIT.

Centre for computing history. n.d. *The Manchester Baby, the world's first stored program computer, ran its first program*. http://www.computinghistory.org.uk/det/6013/The-Manchester-Baby-the-world-s-first-stored-program-computer-ran-its-first-program.

Chalmers, David. 1996. "Does a Rock Implement Every Finite-State Automaton?" *Synthese* (108): 310-333.

Chalmers, David J. 1995. "On Implementing a Computation." *Minds and Machines* 4: 391-402.

Chalmers, David. 2006. "Phenomenal Concepts and the Explanatory Gap." In *Phenomenal Concepts and Phenomenal Knowledge: New Essays on Consciousness and Physicalism*, by S. Walter and S. Alter. Oxford University Press.

Chalmers, David. 1996. *The Conscious Mind*. New York: Oxford University Press.

Chomsky, Noam. 1965. *Aspects of the Theory of Syntax*. Cambridge, Massachusetts: The MIT Press.

Chomsky, Noam. 1957. *Syntactic Structures*. New York: Die Deutsche Bibliothek.

Chrisley, R. L. 1995. "Why Everything Doesn't Realize Every Computation." *Minds and Machines* 4: 403-430.

Chrisley, Ronald. 2000. "Transparent computationalism." In *New Computationalism: Conceptus-Studien 14*, by Matthias Scheutz, 105-121. Saint Augustin.

Church, Alonzo. 1936. "An Unsolvable Problem of Elementary Number Theory." American Journal of Mathematics. 345-363.

Coelho Mollo, Dimitri. 2018. "Functional individuation, mechanistic implementation: the proper way of seeing the mechanistic view of concrete computation." *Synthese* 3477-3497.

Colburn, T. 1999. "Software, Abstraction, and Ontology." *The Monist* 82 (1): 3-19.

Copeland, Jack B. 2020. *The Modern History of Computing*. https://plato.stanford.edu/archives/win2020/entries/computing-history/.

Copeland, Jack. 1996. "What Is Computation." *Synthese* 108 (3): 335-359.

Craik, K.J. W. 1943. *The Nature of Explanation*. Cambridge: Cambridge University Press.

Craver, Carl. 2007. *Explaining the Brain*. Oxford: Clarendon Press.

Craver, Carl. 2001. "Role Functions, Mechanisms, and Hierarchy." *Philosophy of Science* 68 (1): 53-74.

Craver, Carl, and James Tabery. 2019. *Mechanisms in Science*. https://plato.stanford.edu/archives/sum2019/entries/science-mechanisms/.

Cummins, Robert. 1983. *The nature of psychological explanation*. Cambridge: MIT Press.

Dennett, Daniel C. 1978. *Brainstorms*. Vol. Sixth Printing. Montgomery: Bradford Books.

Deutsch, J. A. 1960. "Learning-Behavior-Information." *Contemporary Psychology: A Journal of Reviews* 5 (5): 147-150.

Dietrich, Eric. 1989. "Semantics and the Computational Paradigm in Cogntive Psychology." *Synthese* 79: 119-141.

Dretske, Fred. 1981. *Knowledge and the Flow of Information*. Basil Blackwell.

Evans, Gareth. 1982. "The Varieties of Reference." By John Henry McDowell. Clarendon Press.

Feigl, Herbert. 1967. *The "Mental" and the "Physical": The Essay and a Postscript*. Minneapolis: University of Minnesota Press.

Feigl, Herbert. 1958. *The 'Mental' and the 'Physical*'. Vol. 2, in *Concepts, Theories and the Mind-Body Problem. Minnesota Studies in the Philosophy of Science*, 370-479. Minneapolis: University of Minnisota Press.

Fisher, R. 1935. *The Design of Experiments*. Oliver and Boyd.

Floridi, Luciano. 2010. *Information - A Very Short Introduction*. Oxford: Oxford University Press.

Floridi, Luciano. 2004. "Open Problems in the Philosophy of Information." *Metaphilosophy* (35).

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

Floridi, Luciano. 2005. "Is Semantic Information Meaningful Data?" *Philosophy and Phenomenological Research* 70 (2): 351-370.

Floridi, Luciano. 2004. "Outline of a Theory of Strongly Semantic Information." *Minds & Machines* 14 (2): 197-221.

Fodor A., Jerry, and Zenon W. Pylyshyn. 1988. "Connectionism and cognitive architecture: A critical analysis." *Cognition* 28 (1-2): 3-71.

Fodor, Jerry. 1990. *A Theory of Content and Other Essays*. Cambridge, MA: MIT Press.

Fodor, Jerry. 1980. "Methodological solipsism considered as a research strategy in cognitive psychology." *Behavioral and Brain Sciences* 3 (1): 63-109.

Fodor, Jerr. 1968. *Psychological Explanation*. New York: Random House.

Fodor, Jerry. 1974. "Special Sciences (or: The Disunity of Science as a Working Hypothesis)." *Synthese* 28 (2): 97-115.

Fodor, Jerry. 1968. "The Appeal to Tacit Knowledge in Psychological Explanation." *The Journal of Philosophy* 65 (20).

Fodor, Jerry. 1975. *The Language of Thought*. Cambridge: Harvard University Press.

Fodor, Jerry. 1981. "The Mind-Body Problem." *Scientific American* 244 (1): 114-123.

Fodor, Jerry, and Zenon Pylyshyn. 1988. "Connectionism and cognitive architecture: A critical analysis." *Cognition* 28 (1-2): 3-71.

Francescotti, Robert. 2002. "Understanding Physical Realization (and what it does not entail)." *The Journal of Mind and Behavior* 23 (3): 279-291.

Freeth, Tony. 2022. *An Ancient Greek Astronomical Calculation Machine Reveals New Secrets*. January 1. Accessed 2023. https://www.scientificamerican.com/article/an-ancient-greek-astronomical-calculation-machine-reveals-new-secrets/.

Frege, Gottlob. 1948. "Sense and Reference." *The Philosophical Review* 57 (3): 209-230.

Fresco, Nir. 2019. "Explaining computation without semantics: Keeping it simple." *Minds and Machines* 20 (2): 165-181.

Fresco, Nir. 2014. *Physical Computation and Cognitive Science*. Springer.

Frigerio, Aldo, Alessandro Giordani, and Luca Mari. 2013. "On Representing Information: A Characterization of the Analog/Digital Distinction." *Dialectica* 67 (4): 455-483.

Gandy, Robin. 1980. "Church's Thesis and Principles for Mechanism." Edited by Jon Barwise, Keisler H. Jerome and Kenneth Kunen. *The Kleene Symposium: Proceedings of the Symposium Held June 18–24*. Madison. 123-148.

Gillett, Carl. 2002. "The Dimensions of Realization: A Critique of the Standard View." *Analysis* 63: 316-323.

Giovannelli, Alessandro. 2017. *Goodman's Aesthetics*. https://plato.stanford.edu/archives/fall2017/entries/goodman-aesthetics/.

Godfrey-Smith, Peter. 2000. "On the Theoretical Role of 'Genetic Coding'." *Philosophy of Science* 67: 26-44.

Godfrey-Smith, Peter. 2009. "Triviality Arguments Against Functionalism." *Philosophical Studies* 145 (2): 273-295.

Goodman, Nelson. 1968. *Languages of Art*. Indianapolis.

Graham, George. 2023. *Behaviorism*. https://plato.stanford.edu/archives/spr2023/entries/behaviorism/.

Gregersen, Erik. 2022. *Analog Computer*. https://www.britannica.com/technology/analog-computer#ref255063.

Grice, H. P. 1957. "Meaning." *The Philosophical Review* 66 (3).

Haimovici, Sabrina. 2013. "A Problem for the Mechanistic Account of Computation." *Journal of Cognitive Science* 14: 151-181.

Harman, Gilbert H. 1967. "Psychological Aspects of the Theory of Syntax." *The Journal of Philosophy* 64 (2): 75-87.

Haugeland, John. 1981. "Analog and analog." *Philosophical Topics* 12: 213-222.

Haugeland, John. 1985. *Artificial Intelligence: The Very Idea*. Cambridge: MIT Press.

Haugeland, John. 1978. "The nature and plausibility of cognitivism." *Behavioral and Brain Sciences* 1 (2): 215-226.

Hilbert, D., and W Ackermann. 1928. "Grundzüge der Theoretischen Logik." Berlin: Springer.

Horgan, Terence, and John Tienson. 1989. "Representations without Rules." *Philosophical Topics* 65 (20): 147-174.

Jackson, F., R. Pargetter, and E. Priro. 1982. "Functionalism and Type-Type Identity Theories." *Philosophical Studies* 42: 209-225.

Jackson, Frank. 2010. *Language, Names, and Information*. Oxford: Wiley-Blackwell.

Jackson, Frank. 1986. "What Mary Didn't Know." *Journal of Philosophy* 83: 291-295.

Johnson-Laird, P N. 1988. *The Computer and the Mind: An Introduction to Cognitive Science*. Cambridge: Harvard University Press.

Johnson-Laird, Philip. 1983. *Mental Models*. Lawrence Erlbaum Associates, Inc.

Katz, M. 2008. "Analog and digital representation." *Minds and Machines* 18: 403-408.

Kim, Jaegwon. 1999. "Making sense of emergence." *Philosophical Studies* 95: 3-36.

Kim, Jaegwon. 1998. *Mind in a Physical World*. Cambridge: MIT Press.

Kim, Jaegwon. 1992. "Multiple Realization and the Metaphysics of Reduction." *Philosophy and Phenomenological Research* 52: 1-26.

Klein, Colin. 2008. "Dispositional Implementation Solves the Superfluous Structure Problem." *Synthese* 165 (1): 141-153.

Kosslyn, Stephen. 1996. *Image and Brain: The Resolution of the Imagery Debate*. The MIT Press.

Kosslyn, Stephen M. 1981. "The medium and the message in mental imagery: A theory." *Psychological Review* 88 (1): 46-66.

Levin, Janet. 2021. *Functionalism*. https://plato.stanford.edu/archives/win2021/entries/functionalism/.

Lewis, David. 1971. "Analog and Digital." *Nous* 321-327.

Lewis, David. 1972. "Psychophysical and Theoretical Identifications." *Australasian Journal of Philosophy* 50: 249-258.

Li, Ming, and Paul Vitanyi. 2008. *An Introduction to Kolmogorov Complexity and Its Applications*. New York: Springer.

Lovelace, Ada. 1843. *Translator's Notes to M. Menabrea's Memoir*. Scientific Memoirs 3 .

Ly, A., M. Marsman, J. Verhagen, R.P. Grasman, and E.J. Wagenmakers. 2017. "A Tutorial on Fisher information." *Journam of Mathematical Psychology* 80: 40-55.

Ly, Alexander, Maarten Marsman, Josine Verhagen, Raoul Grasman, and Eric-Jan Wagenmakers. 2017. "A Tutorial on Fisher Information." *Journal of Mathematical Psychology* 80: 40-55.

Lycan, W. G. 1987. *Consciousness*. Cambridge: MIT Press.

Lycan, W. G. 1981. "Form, Function, and Feel." *The Journal of Philosophy* 78 (1): 24-50.

Machamer, Peter, and et al. 2000. "Thinking about Mechanisms." *Philosophy of Science* 67 (1): 141-154.

MacKay, D. M. 1983. "The Wider Scope of Information Theory." In *The Study of Information: Interdisciplinary messages*, by F. Machlup and U. Mansfield, 485-492. New York: Wiley.

MacLennan, Bruce J. 2009. "Analog Computation." In *Encyclopedia of Complexity and Systems Science*, by Robert A. Meyers, 271-294. Springer Reference.

Maley, Corey J. 2009. "Analog and digital, continuous and discrete." unpublished.

Maley, Corey J. 2021. "The physicality of representation." *Synthese* 14725-14750.

Maley, Corey J. 2023. "Analog computation and representation." *The British Journal for the Philosophy of Science*.

Maley, Corey. 2022. "Medium Independence and the Failure of the Mechanistic Account of Computation." *Ergo* 1-27.

Marr, David. 2010. *Vision a Computational Investigation into the Human Representation and Processing of Visual Information*. MIT Press.

Mazzoni, P., and et al. 1991. "A More Biologically Plausible Learning Rule for Neural Networks." *The National Academy of Sciences*. 4433-4437.

McCulloch, W. S., and W. H. Pitts. 1943. "A Logical Calculus of the Ideas Immanent in Nervous Activity." *Bulletin of Mathematical Biophysics* 115-133.

McCulloch, Walter. 1974. "Recollections of the Many Sources of Cybernetics." *ASC Forum*. 5-16.

McGrath, Mike. 2019. *Coding for Beginners: programming made easy for all ages*. In Easy Steps Limited.

Medlin, Brian. 1967. "Ryle and the Mechanical Hypothesis." In *The Identity Theory of Mind*, by C. F. Presley.

Melnyk, Andrew. 1996. "Searle's Abstract Argument against Strong AI." *Synthese* 108 (3): 391-419.

Milikan, Ruth. 2004. *Varieties of Meaning*. Cambridge, MA: MIT Pressq.

Milkowski, Marcin. 2013. *Explaining the Computational Mind*. Cambridge: The MIT Press.

Milkowski, Marcin. 2012. "Is computation based on interpretation." *Semiotica* 188 (1/4): 219-228.

Milkowski, Marcin. 2007. "Is computationalism trivial?" In *Computation, Information, Cognition - The Nexus and the Liminal, ed.*, by G. Dodig-Crnkovic and S. Stuart, 236-246. Newcastle: Cambridge Scholars Press.

Miller, G. A. 1951. *Language and Communication*. New York: McGraw Hill.

Miller, George. 1956. "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information"." *Psychological Review* 63 (2): 81-97.

Millhouse, Tyler. 2019. "A Simplicity Criterion for Physical Computation." *British Journal for Philosophy of Science* 70: 153-178.

Minksy, Marvin. 1968. *Computation: Finite and Infinite Machines*. Cambridge, MA: MIT Press.

Nagel, Thomas. 1961. *The Structure of Science*. New York: Harcourt.

Neisser, Ulric. 1967. *Cognitive Psychology*.

Neumann, John von. 1945. "First Draft of a Report on the EDVAC." Pennsylvania, June 30.

Newell, A, and H.A. Simon. 1961. "Computer simulation of human thinking." *Science* 2011-2017.

Owens, Larry. 1986. "Vannevar Bush and the Differential Analyzer: The Text and Context of an Early Computer." *Technology and Culture* 27 (1): 63-95.

Papayannopoulos, Philippos, Nir Fresco, and Oron Shagrir. 2022. "Computational indeterminacy and explanations in cognitive science." *Biology & Philosophy* 37 (47): 1-30.

Papinaeu, David. 1995. "Arguments for supervenience and physical realization." In *Supervenience: New Essays*, by E. Savellos and U. Yalcin, 226-243. Cambridge: Cambridge University Press.

Piccinini, Gualtiero. 2006. "Computation without representation." *Philosophical Studies* 137 (2): 205-241.

Piccinini, Gualtiero. 2007. "Computing Mechanisms." *Philosophy of Science* 74: 501-526.

Piccinini, Gualtiero. 2004. "Functionalism, Computationalims, and Mental States." *Studies in History and Philosophy of Science Part A* 35 (4): 811-833.

Piccinini, Gualtiero. 2015. *Physical Computation: A Mechanistic Account*. Oxford: Oxford University Press.

Piccinini, Gualtiero. 2004. "The First Computational Theory of Mind and Brain: A Close Look at McCullock and Pitts's "Logical Calculus of Ideas Immanent in Nervous Activity"." *Synthese* 175-215.

Piccinini, Gualtiero, and Andrea Scarantino. 2010. "Information Processing, Computation, and Cognition." *SSRN Electronic Journal*.

Piccinini, Gualtiero, and Carl Craver. 2011. "Integrating Psychology and Neuroscience: Functional Analyses as Mechanism Sketches." *Synthese* 183 (4): 283-311.

Piccinini, Gualtiero, and Corey J. Maley. 2021. *Stanford Encyclopedia of Philosophy*. https://plato.stanford.edu/archives/sum2021/entries/computation-physicalsystems/.

Piccinini, Gualtiero, and S. Bahar. 2013. "Neural Computation and the Computational Theory of Cognition." *Cognitive Science* (34): 452-488.

Place, U. T. 2004. *Identifying the Mind, Selected Papers of U.T. Place*. New York, New York: Oxford University Press.

Place, U. T. 1956. "Is Consciousness a Brain Process?" *British Journal of Psychology* 44-50.

Place, U. T. 1960. "Materialism as a Scientific Hypothesis." *The Philosophical Review* 69 (1): 101-104.

Polger, Lawrence, and Thomas Shapiro. 2016. *The Multiple Realization Book*. Oxford University Press.

Presley, C. F. 1967. *The Identity Theory of Mind*. Univeristy of Queensland Press.

Putnam, Hilary. 1963. "Brains and Behavior." In *Analytical Philosophy*, by R.J. Butler, 1-20. New York: Barnes and Noble.

Putnam, Hilary. 1960. "Minds and Machines." In *Dimensions of Minds*, by Sidney Hook, 138-164. New York: New York University Press.

Putnam, Hilary. 1967a. "Psychological Predicates." In *Art, Mind, and Religion*, by W. H. Capitan and D. D. Merrill, 37-48. Pittsburgh: University of Pittsburgh Press.

Putnam, Hilary. 1988. *Representations and Reality*. Cambridge, MA: MIT Press.

Putnam, Hilary. 1975. "The Meaning of Meaning. In K. Gunderson." In *Minnesota Studies in the Philosophy of Science* , by K. Gunderson, 131-193. Minneapolis: University of Minnesota Press.

Putnam, Hilary. 1967b. "The Mental Life of Some Machines." In *Intentionality, Minds, and Perception*, by H. Castaneda, 177-200. Detroit: Wayne State University Press.

Pylyshyn, Z W. 1984. *Computation and Cognition: Towards a Foundation for Cognitive Science*. Cambridge: MIT Press.

Pylyshyn, Zenon. 1984. *Computation and Cognition*. Cambridge: MIT Press.

Pylyshyn, Zenon W. 1986. *Computation and Cognition: Toward a Foundation for Cognitve Science*. The MIT Press.

Pylyshyn, Zenon W. 1981. "The imagery debate: Analogue media versus tacit knowledge." *Psychological Review* 88 (1): 16-45.

Recanati, Francois. 2012. *Mental Files*. Oxford University Press.

Recanati, Francois. 2016. *Mental Files in Flux*. Oxford: Oxford University Press.

Rescorla, Michael. 2014. "A theory of computational implementation." *Synthese* 191: 1277-1307.

Rescorla, Michael. 2013. "Against Structuralist Theories of Computational Implementation." *The British Journal for the Philosophy of Science* 64: 681-707.

Rescorla, Michael. 2020. *The Computational Theory of Mind*. https://plato.stanford.edu/entries/computational-mind/.

Rescorla, Michael. 2020. *The Language of Thought Hypothesis*. https://plato.stanford.edu/entries/language-thought/index.html.

Riehl, A. 1879, 1881, 1887. *Der philosophische Kritizismus und seine Bedeutung für die positive Wissenschaft. Geschichte und System*. Vol. 3 volumes.

Ryle, Gilbert. 1949. *The Concept of Mind*. The University of Chicago Press.

Schembri, Theirry, and Olivier Boisseau. n.d. *Heathkit EC-1*. https://www.old-computers.com/museum/computer.asp?st=1&c=787.

Scheutz, Matthias. 2001. "Computational versus Causal Complexity." *Minds and Machines* 11: 543-566.

Scheutz, Matthias. 1999. "When Physical Systems Realize Functions..." *Minds and Machines* 9: 161-196.

Schliehauf , Bill. 2004. *Battenberg Course Indicator*. http://www.gwpda.org/naval/ou5274.htm.

Schneider, Susan. 2009. "The Nature of the Symbols in the Language of Thought." *Mind and Language* 523-553.

Schonbein, Whit. 2014. "Varieties of Analog and Digital Representation." *Minds & Machines* 24: 415-438.

Schroeder, Marcin J. 2004. "An Alternative to Entropy in the Measurement of Information." *Entropy* 6: 388-412.

Schweizer, Paul. 2016. "Computation in Physical Systems: a Normative Mapping Account." *International Association for Computing and Philosophy*.

Searle, John. 1990. "Is the Brain a Digital Computer?" *Proceedings and Addresses of the American Philosophical Association*. American Philosophical Association. 21-37.

Searle, John. 1992. *The Rediscovery of the Mind*. Cambridge, MA: MIT Press.

Sequoiah-Grayson, Sebastian, and Luciano Floridi. 2022. *Semantic Conceptions of Information*. https://plato.stanford.edu/entries/information-semantic/.

Shagrir, Oron. 2001. "Content, Computation and Externalism." *Mind* 110 (438): 369-400.

Shagrir, Oron. 2020. "In defense of the semantic view of computation." *Synthese*.

Shagrir, Oron. 2022. *The Nature of Physical Computation*. New York: Oxford University Press.

Shagrir, Oron. 2006. "Why We View the Brain as a Computer." *Synthese* 153 (3): 393-416.

Shannon, C. E. 1948. "A Mathematical Theory of Communication." *The Bell System Technical Journal* 27: 379-423.

Shannon, Claude E., and Warren Weaver. 1949. *The Mathematical Theory of Communication*. Urbana, IL: University of Illinois Press.

Shannon, Claude, and Warren Weaver. 1963. *The Mathematical Theory of Communication*. Illinois: University of Illinois Press.

Shoemaker, Sydney. 2001. "Realization and Mental Causation." In *Physicalism and its Discontents*, by Carl Gillett and Barry Loewer. New York: Cambridge University Press.

Sieg, Wilfred. 2009. "Church without Dogma: Axioms for Computability." In *New Computational Paradigms: Changing Conceptions of What is Computable*, by Barry Cooper, Lower Benedikt and Sorbi Andrea, 139-152. New York: Springer Verlag.

Sipser, Michael. 2006. *Introduction to the Thoery of Computation*. Vol. 2nd Edition. Boston, Massachusettes: Thomson Course Technology.

Skinner, B. F. . 1984. "The operational analysis of psychological terms." *The Behavioral and Brain Sciences* 7: 547-581.

Small, JS. 2001. *The Analogue Alternative: The electronic analogue computer in Britain and the USA*. London & New York: Routledge.

Smart, J.J.C. 1959. "Sensations and Brain Processes." *The Philosophical Review* (Duke University) 68 (2): 141-156.

Smith, Brian Cantwell. 2010. *Age of significance: Introduction.*

Smith, Brian Cantwell. 1982b. "Reflection and Semantics in a Procedural Language." *Ph.D. thesis. M.I.T., Computer Science Lab.* Cambridge, Massachusettes: Massachusetts Institute of Technology, January 25.

Smith, Brian Cantwell. 1982a. "Semantic Attribution and the Formality Constraint." *unpublished report, Xerox PARC.*

Smith, Brian Cantwell. 2002. "The Foundations of Computing." In *Computationalism New Directions*, by Matthias Scheutz, 24-58. Cambridge: The MIT Press.

Smith, Brian Cantwell. 1996. *The origins of objects.* Cambridge: MIT Press.

Song, Yuhang, Thomas Lukasiewicz, Zhenghua Xu, and Rafal Bogacz. 2020. "Can the Brain Do Backpropagation? —Exact Implementation of Backpropagation in Predictive Coding Networks." *Adv Neural Inf Process Syst* 1-23.

Sprevak, Mark. 2010. "Computation, individuation, and the received view on representation." *Studies in History and Philosophy of Science* 41: 260-270.

Sprevak, Mark. 2018. "Triviality arguments about computational implementation." In *Routledge Handbook of the Computational Mind*, by Mark Sprevak and M. Colombo, 175-191. London: Routledge.

Stacewicz, Pawel. 2020. "Analogicity in Computer Science: Methodological Analysis." *Studies in Logic, Grammar, and Rhetoric* 63 (76): 69-86.

Stich, Stephen P. 1983. *From Folk Psychology to Cognitive Science: The Case Against Belief.* Cambridge, Massachusettes: MIT Press.

Stoljar, Daniel. 2022. *Physicalism.* https://plato.stanford.edu/archives/sum2022/entries/physicalism/.

Thomson, James. 1876. "On an Integrating Machine Having a New Kinematic Principle." *Proceedings of the Royal Society of London*. Royal Society. 262-265.

Turing, Alan. 1936. "On Computable Numbers, With An Application ti tge Entscheidungsproblem." *Proceedings of the London Mathematical Society, Series 2, 42.* 230-265.

von Neumann, John. 1993. "First Draft of a Report on the EDVAC." *IEEE Annals of the History of Computing* (University of Pennsylvania) 15 (4).

Von Neumann, John. 1961. "The general and logical theory of automata." In *Collected Works, vol. V*, by A. H. Taub, 288-328. Oxford: Pergamon Press.

Watson, John B. 1913. "Psychology as the Behaviorist Views it." *Psychological Review* 20: 158-177.

Weiskopf, Daniel. 2011. "Models and mechanisms in psychological explanation." *Synthese* 183 (3): 313-338.

Weiskopf, Daniel. 2011. "The functional unity of special science kinds." *British Journal for the Philosophy of Science* 62: 233-258.

Wimsatt, W. C. 1972. "Teleology and the Logical Structure of Function Statements." *Studies in History and Philosophy of Science.*

Wittgenstein, Ludwig. 1953. *Philosophical Investigations.* Oxford: Basil Blackwell Ltd.

Zenon, Pylyshyn. 1980. "Computation and Cognition: Issues in the Foundations of Cognitive Science." *Behavioral and Brain Sciences* 3 (1): 111-132.

Zipser, David, and Richard A Andersen. 1988. "A Back-Propogation Programmed Network that Simulates Response Properties of a Subset of Posterior Parietal Neurons." *Nature* 331 (6158): 679-684.