# Syntax, Semantics, and Computer Programs: Comments on Turner's *Computational Artifacts*

**William J. Rapaport**

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

rapaport@buffalo.edu
http://www.cse.buffalo.edu/~rapaport/

March 29, 2019

**Abstract**

Turner argues that computer programs must have purposes, that implementation is not a kind of semantics, and that computers might need to understand what they do. I respectfully disagree: Computer programs need not have purposes, implementation is a kind of semantic interpretation, and neither human computers nor computing machines need to understand what they do.

## 1 Introduction

In 2004, I created a university course on the philosophy of computer science. At that time, there were very few resources to draw on.[1] In the intervening decade-plus, much has changed. In addition to Matti Tedre's textbook *The Science of Computing* (2015) and Gualtiero Piccinini's monograph *Physical Computation* (2015), we now have Ray Turner's very useful *Computational Artifacts: Towards a Philosophy of Computer Science* (2018).

[1] For more information on my course, see Rapaport 2005b and the course syllabi at https://cse.buffalo.edu/~rapaport/510.html. The current draft (comments appreciated!) of a textbook based on my lectures is available as Rapaport 2019.

In this commentary on Turner's book, I present my own variations on three of his themes: the purposes of computer programs, the nature of implementation, and whether programs need to be understood in order to be executed.

## 2 Must Computer Programs Have Purposes?

> [C]an one provide an ontological analysis of programs without some mention of their intended function? (Turner, 2018, p. 44)

*Must* a computer program include as a part of it (even if only as an unexecutable comment) an expression of its intended function? Or *can* a program be considered separately from any purpose that it might have? Certainly, some (and probably almost all) programs have an intended function or goal. If we let $G$ be an expression of a goal, and $A$ be an algorithm encoded in some computer programming language, then we can ask whether the basic form of a program is:

> In order to accomplish goal $G$, do $A$

or whether it is just:

> Do $A$

with no mention of any $G$.[2]

Turner distinguishes between a program's *functional* description (what I have labeled '$G$') and its *structure* (my '$A$'). The functional description tells us "*what* the device is supposed to do" (p. 44, my italics). One would thus expect that the structure tells us *how* it does that. Although Turner comes close to saying that on p. 45, when he notes that a definition of sorting "tells us *what* sorting is, not *how* to sort" (my italics), his official characterization of structure is "what it *actually* does" (p. 44, my italics). But, of course, what a program *actually* does might *not* be how it is *supposed* to accomplish its function; after all, programs can have bugs. Nevertheless, bugs notwithstanding, the standard way of understanding the function-structure distinction is in terms of "what" versus "how", with an assumption that the "how" correctly implements the "what".

There is no question that commercially produced computer programs are designed for a purpose, expressed in the program's specification. In its most abstract form, a specification is a description of the program's expected input-output

[2] There is a large literature on this, including Marr 1982; Suber 1988; Cleland 1993; Egan 1995, 2010, 2014; Peacocke 1995, 1999; Piccinini 2004, 2006, 2008; Rescorla 2007, 2012, 2014, 2015; Sprevak 2010; Buechner 2011, 2018; Anderson 2015; Shagrir and Bechtel 2015; Hill 2016; Dewhurst 2018; Shagrir 2018. For an overview, see Rapaport 2017a, 2019.

behavior. (Thus, it is a "functional description" in both the teleological and the set-theoretical senses of 'function'!)

But this does not mean that programs cannot be studied independently of those purposes. In fact, a program $P$ that is *intended* to compute a function $f$ (characterized as a certain set of input-output pairs) might actually ("incorrectly") compute some other function $g$; yet $P$ is still a perfectly good expression of an algorithm. And it may be difficult, perhaps even logically impossible, to determine which function it is supposed to compute (Buechner, 2011, 2018).

Nor does it mean that programs cannot be created without an intended purpose: A program that, with no rhyme or reason, performs arbitrary, unrelated arithmetic operations on an input is nonetheless a program. As Daniel C. Dennett has observed:

> Algorithms, **in the popular imagination**, are algorithms *for* producing a particular result. ... [E]volution can be an algorithm, and evolution can have produced us by an algorithmic process, without its being true that evolution is an algorithm *for* producing us. (Dennett, 1995, p. 308, my boldface, original italics)

We see the same thing in the case of language. Sentences are "designed" by speakers to express their thoughts. A thought is a specification of *what* a speaker wants to say; a sentence is *how* the speaker expresses the thought (successfully or not; for discussion on that point, see Rapaport 2003). But sentences can be studied independently of those thoughts, and they can be created independently of any thoughts to be expressed (consider Lewis Carroll's *Jabberwocky*).

Thus, my answer to Turner's question is: Yes, a purely syntactic ontological analysis of programs *can* be provided; no (external) semantics is *necessary*. In the next two sections, we will look more closely at the nature of both syntactic and semantic understanding.

# 3 Is Implementation a Kind of Semantics?

> Moreover, some philosophers appear to endorse the view that implementation provides *the* semantic *definition* of the language. For instance, Rap[a]port argues that an implementation is always semantic interpretation [192, 193]. This would seem to *imply* that we can use an implementation as a semantic definition. (Turner 2018, pp. 98–99, my italics; Turner's "[192, 193]" are Rapaport 1999, 2005a, respectively.)

Were this something that I advocated, I would not be alone: Frederick B. Brooks (1975, p. 64) observed that "Not only is a formal definition an implementation, an

implementation can serve as a formal definition" (before going on to list, on the next page, the advantages and "formidable" disadvantages). But this is not quite my view. Let me explain.

Although the term 'semantics' is typically used in connection with the meaning or truth of *linguistic* expressions, in its most abstract and general form, it is the study of the relations between *any* two domains, one of which is intended to be understood in terms of the other. The one *needing* understanding is typically considered to be a "syntactic" domain, and the one *providing* the understanding (the interpretation) is typically considered to be a "semantic" domain.

Similarly, the term 'syntax' is typically used in connection with the *grammar* of a language, but in its most abstract and general form, the "syntactic" domain is simply any domain (not necessarily a language) understood solely in terms of its components, their properties, and the relations among them, i.e., understood solely in terms of what Charles Morris (1938, p. 6) described as "the formal relation of signs to one another."[3] When the domain is a *language*, this understanding is its *grammar*; when the domain is a *logic*, this understanding is its *proof theory*.

A semantic interpretation of a syntactic domain requires a "semantic" domain: a distinct domain that is also understood solely in terms of *its* components, their properties, and the relations among them, i.e., understood solely in terms of *its* syntax. (When this domain is the world, or a model of the syntactic domain, its "syntax" is its *ontology*.)[4] A semantic interpretation of the syntactic domain in terms of the semantic domain is a way to understand the former in terms of the latter by laying out the relations *between* the two domains; those relations, it should be noted, are not part of either domain (Rapaport, 1988, 1995, 2012, 2017b, 2018).

When I argued (in the two essays that Turner cites) that implementation is semantic interpretation, all I meant was that implementation—typically, but not necessarily, the *physical* implementation of an *abstraction*—is a *kind* of semantic relation in this sense, and (more specifically) that it is *not* some *other* kind of relation—that it is not individuation, instantiation, reduction, or supervenience. In particular, I did not offer it as a (and certainly not as "the") "semantic *definition* of a language", nor do I see how being a "definition" would be *implied* from its being *a* way of understanding a syntactic domain. A physical implementation of a computer program might be the static physical "switch settings" (or electronic analogues thereof) that implement the text of a program in a physical computer, or

[3]This way of characterizing syntax suggests that it might be an abstract analogue of a "mechanism", e.g., "a structure performing a function in virtue of its component parts, component operations, and their organization" (Bechtel and Abrahamsen, 2005, p. 423).

[4]Characterizations of ontology that suggest this connection to syntax include Smith 2003 and Hofweber 2018, §3.1: "the study of the most general features of what there is, and how the things there are relate to each other."

it might be the actual dynamic process that comes into being when the program is executed. I do not see how either of these implementations would be a *definition* of that program, though I do think that they would be ways of *understanding* that program.

Turner suggests (p. 99) that there are "normative requirements on semantic theories", giving as an example "provid[ing] criteria of correctness for the user or the implementer", and he argues that a mere translation of a computer program into a compiled implementation of it would "just pass the burden of meaning onto another language." But this "burden" is always and unavoidably the case: There are two ways to understand a syntactic domain $S$. The first way is the one sketched out above: One can understand $S$ *semantically*, in terms of a semantic domain $T$. But for $T$ to provide an understanding of $S$, $T$ itself must already be understood. How do you understand $T$? Again, in one of two ways, the first being to treat $T$ as a syntactic domain and to understand it in terms of yet another antecedently understood semantic domain $T'$. But for this recursion to stop, there must be a domain that is not understood in terms of *another* domain, but one that is understood in terms of *itself*. To understand a domain in terms of itself is to understand its *syntax*: its component parts, their properties, and their relations to each other (including, if the domain is a logic, its proof theory). Thus, the second way to understand any domain is to understand it *syntactically*. Because syntactic understanding is the base case of this recursive characterization of understanding, it follows that, in the final analysis, all understanding (of this kind)[5] is syntactic understanding (Rapaport, 1986, 1995, 2017b).

As for the issue of "correctness", Turner argues (pp. 102–103) that, on my view of implementation as semantic interpretation, an implementation of finite sets as finite lists does not "provide the semantic interpretation of finite sets" because, in that case,

> the axioms for lists [would] fix the correctness of the axioms for finite sets. But this is the wrong way around. On the contrary, it is the axioms for the abstract type that must be preserved by the implementation.

But why would the list axioms "fix the correctness" of the set axioms? All the list axioms have to do, as Turner notes, is "preserve" the set axioms. If we "understand" lists better than sets (because, say, our programming language only recog-

[5]The parenthetical hedge is simply to allow for the possibility of there being other kinds of understanding that might not involve either syntactic or semantic understanding, though I would be hard-pressed to think of one. "Intuitive understanding", perhaps? But I would consider an intuitive understanding of some domain to be a kind of understanding that one has by having become familiar with that domain, and that would just be what I am calling syntactic understanding. See Rapaport 1995 for further discussion.

nizes lists, not sets), then our programming language allows us to understand sets in terms of lists.

The important point, however, and one that Turner and I seem to agree on, is that each domain can be understood in terms of the other. Which one is counted as the *syntactic* domain that begs to be understood and which one is counted as the *semantic* domain that provides the understanding depends on which one is understood "better" or "antecedently". As Turner puts it,

> If I take the semantics as having priority, then I must know that I am evaluating the formal rules. On the other hand, if I take the proof theory to have priority, then I must know that I am evaluating the semantic account. (p. 125)

That is, if I antecedently understand the semantic domain, then I can use it to understand the syntactic domain; but if I antecedently understand the "syntactic" domain, then I can use *it* to understand the "semantic" domain. He calls this a difference in "intentional stance" (though its relation to Dennett's (1987) intentional stance is not clear) and notes that "It determines what is *function* and what is *structure*" (p. 125), suggesting that one person's function might be another's structure. Perhaps that means that the functional $G$ and the structural $A$ that were discussed in §2 are not so different after all.

# 4 Do Computers Need to Understand What They Do?

> An important psychological part of the appeal of Turing's machines concerns the nature of their basic instructions; these are said to be *atomic* in the sense that they can be performed *without thought*. (Turner, 2018, p. 192)

After saying this, Turner goes on to quote Gregory Chaitin, who says that a computer does not have to ...

> ... comprehend the result of any part of the operations it performs. ... [A] program ... can demand any finite number of mechanical manipulations of numbers, but it cannot ask for judgments about their meaning. (Chaitin, 1975)

Although Turner mistakenly attributes this passage to Turing, surely Turing would have agreed, as does Dennett:

> Turing's idea was a ... strange inversion of reasoning. The Pre-Turing world was one in which computers were people, who had to understand mathematics in order to do their jobs. Turing realised that this was just not necessary: you could take the tasks they performed and squeeze out the last

> tiny smidgens of understanding, leaving nothing but brute, mechanical actions. IN ORDER TO BE A PERFECT AND BEAUTIFUL COMPUTING MACHINE IT IS NOT REQUISITE TO KNOW WHAT ARITHMETIC IS. (Dennett, 2013, p. 570, capitalization in original)[6]

In the present section's epigraph, above, Turner talks of "Turing's machines", and Chaitin is talking about "digital computers". Yet Turner then observes:

> The instructions must be complete and explicit, and the **human** computer is not required to *comprehend* any part of the basic operations; **she** cannot ask for judgments about their meaning. (p. 193, my boldface, Turner's italics)

Why does Turner introduce a *human* "computer"? Of course, Turing (1936) famously analyzed how a human computes, but—as Dennett notes—he then "squeeze[d] out the last tiny smidgens of understanding", so the human who might execute a program (just as well as a machine that might execute it) need not "understand" it. After all, the machine *cannot* understand it even if the human can.[7]

But Turner goes on to ask:

> Is it that these atomic instructions have a meaning but may be performed without understanding? For the sake of argument, assume that they are not meaningless; they have semantic content. This does not mean that eventually the human computer will be able to perform atomic instructions without thought. The human computer may be able to. But, to begin with at least, **the meaning must have been deployed.** How else could one carry them out? Turing cannot be arguing that the human computer can be trained to do this without thought. This would make little sense, since the atomic operations are the most primitive. If the computer is to be trained to learn these, then there must be more primitive ones that form the basis of the learning process. And the atomic instructions are the most primitive ones. So it is hard to make out how, if they have meaning, how [sic] the human computer can initially proceed without thought. So, if the atomic instructions are taken to have meaning, it is hard to see what can be meant by the claim that the human computer does not comprehend the result of any part of the operations **it** performs.
>
> Seemingly, we must assume that Turing's atomic instructions are *meaningless*. (p. 193, my boldface, Turner's italics)

This passage appears to be a *reductio* argument for the conclusion "that Turing's atomic instructions are *meaningless*". Turner follows it with another *reductio* argument, this time for the *opposite* conclusion, thus setting up an antinomy:

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

[7] I have argued elsewhere that a suitably programmed AI computer *could* (syntactically) understand what it is doing (see, e.g., Rapaport 1988, 2012). The fact remains that computers don't *have* to understand, and certainly current computers *don't* understand.

> On the assumption that all basic or atomic instructions are meaningless, have no semantic content, it is hard to maintain any compositional theory of meaning. Via compositionality, a Turing machine program is a collection of conditional expressions that inherits its semantic content from its atomic instructions. And so, if all the atomic instructions are meaningless, the compositionality thesis would lead us to conclude that all programs in the language of Turing machines are meaningless, i.e., the collection of instructions that constitute any Turing machine, is meaningless. But this is absurd.
>
> So it would seem that the operations are not meaningless, yet they can be immediately grasped without thought. It is hard to see how both can be true. (pp. 193–194)

I have quoted these two arguments at length, because I find them rather confusing: Is he arguing about *machines* that compute, or about *humans* that compute?

First, why *must* a human computer be able to understand the atomic instructions simply because they have meaning? I once executed some instructions (admittedly, not atomic ones) that—unknown to me—had meaning, yet I executed them without understanding what I was doing:

> I vividly remember the first semester that I taught a "Great Ideas in Computer Science" course aimed at computer-phobic students. We were going to teach the students how to use a spreadsheet program, something that I had never used! So, with respect to this, I was as naive as any of my students. My TA, who had used spreadsheets before, gave me something like the following instructions:
>
> enter a number in cell_1;
> enter a number in cell_2;
> enter '=⟨click on cell_1⟩⟨click on cell_2⟩' in cell_3
>
> I had no idea what I was doing.[8] I was blindly following her instructions *and had no idea that I was adding two integers*. Once she told me that that was what I was doing, my initial reaction was "Why didn't you tell me that before we began?". . . .
>
> Now, (I like to think that) I am a cognitive agent who can come to understand that entering data into a spreadsheet can be a way of adding. A Turing machine that adds or a Mac running Excel is not such a cognitive agent. It does not understand what addition is or that that is what it is doing. And it does not have to. (Rapaport, 2017a, pp. 34, 43–44)

Second, certainly a human *can* understand (atomic) instructions. And understanding them can make it easier for a *human* to execute them:

[8] Some current implementations of Excel require a plus-sign between the two clicks in the third instruction. But the version I was using at the time (1992) did not, making the operation that much more mysterious!

> My wife recently opened a restaurant and asked me to handle the paperwork and banking that needs to be done in the morning before opening (based on the previous day's activities). She wrote out a detailed set of instructions, and one morning I went in with her to see if I could follow them, with her looking over my shoulder. As might be expected, there were gaps in her instructions, so even though they were detailed, they needed even more detail. Part of the reason for this was that she knew what had to be done, how to do it, and why it had to be done, but I didn't. This actually disturbed me, because I tend to think that algorithms should really be just "Do A," not 'To G, do A." Yet I felt that I needed to understand G in order to figure out how to do A. But I think the reason for that was simply that she hadn't given me an algorithm, but a sketch of one, and, in order for me to fill in the gaps, knowing why I was doing A would help me fill in those gaps. But I firmly believe that if it made practical sense to fill in all those gaps (as it would if we were writing a computer program), then I wouldn't have to ask why I was doing it. No "intelligence" should be needed for this task if the instructions were a full-fledged algorithm. If a procedure (a sequence of instructions, including vague ones like recipes) is not an algorithm (a procedure that is fully specified down to the last detail), then it can require "intelligence" to carry it out (to be able to fill in the gaps, based, perhaps on knowing why things are being done). If intelligence is not available (i.e., if the executor lacks relevant knowledge about the goal of the procedure), then the procedure had better be a full-fledged algorithm. There is a difference between a human trying to follow instructions and a machine that is designed to execute an algorithm. The machine cannot ask why, so its algorithm has to be completely detailed. But a computer (or a robot, because one of the tasks is going to the bank and talking to a teller!) that could really do the job would almost certainly be considered to be "intelligent." (Rapaport, quoted in Hill and Rapaport 2018, p. 35)

But Turner says, in the first argument quoted above, that if a human computer carries out an instruction, then "the meaning must have been deployed" and the *human* must "comprehend the result of any part of the operation *it* performs" (my italics—is "it" a human or a machine?).[9] Nevertheless, insofar as the human *uses* that understanding, she is *not* merely executing the instructions by themselves—she is *not* being (or behaving as) a (human) *computer*. Computers do *not*—indeed, they *should* not—*have to* understand what they are doing.

What is meant by 'meaning'? Consider my Excel experience: I carried out the operations without "deploying" the "meaning" of *addition*. That is, I did

[9] On p. 193, Turner refers to the human computer as 'she', as will I.

not deploy the *external* meaning of addition—a semantic interpretation. I *did* "deploy" an *internal* meaning involving clicking on certain cells, but that kind of meaning—*syntactic* "meaning"—*can* be had by a computer; it only involves the computer's ability to execute the instructions, not its ability to understand—*semantically* understand—what those instructions mean in an external sense. Following Piccinini (2004, pp. 401–402, 404), let's call this kind of meaning "internal". (Michael Rescorla (2012, §3, pp. 707–708) calls it—or something very similar—"indigenous meaning"; and I have referred to it as "syntactic semantics" (Rapaport, 1988, 2017a).)

Here, perhaps is where the distinction between a *human* computer and a computing *machine* comes in. In order for a human to execute an algorithm by "following" its instructions, presumably she needs to understand them; hence, they would seem to have to have a meaning for her to understand. Does this imply that a computing *machine* would also have to "understand" them? No, because neither a Turing machine nor a (finite) physical implementation of one *follows* instructions. (I am talking about a single-purpose Turing machine, *not* something like a Mac or PC, which, when loaded with a program, arguably *can* be said to "follow" that program; I discuss these below). A Turing-machine program (e.g., a set of certain quintuples) is a *description* of it (if you like, it is a description of its behavior). In the case of a *physical* Turing machine, its "program" is simply the way it is "hardwired", or the way that its "gears" work (to put it metaphorically); no understanding of instructions is needed, because no instructions are "followed".

The case of a human following instructions is different. But even here, the only thing that the human has to understand are such things as the atomic operations of 'print', 'erase', and 'move', all of which seem quite devoid of "meaning" or "purpose". A better human analogy for the way that a Turing machine behaves is not that of a human explicitly following a program, but the way our brains produce our behavior: We don't "follow" what our brains "tell" us to do, nor do we understand what our brains cause us to do; we just do things because our brains cause us to do them—better: because of the way that our brains are "hardwired".

What about Macs or PCs—(finite) implementations of a *universal* Turing machine? A universal Turing machine is, first and foremost, a single-purpose Turing machine whose program *qua* single-purpose Turing machine is a fetch-execute cycle. It does not "follow" this any more than any Turing machine "follows" its program. It is merely "hardwired" to behave that way. But what about the program that is inscribed on its tape and that—arguably—it *does* "follow"? Does it need to "understand" the (coded) instructions on its tape in order to execute them? I would argue that it does not. Suppose that one of those coded instructions is a sequence of '0's and '1's that codes for "print '1' ". The universal Turing machine, after scanning that coded instruction, is simply hardwired to print a '1'. (It doesn't have

to think to itself "Ah! This means that I have to print a '1'. I'll do that now.")

As for Turner's second argument, why is it "absurd" that programs are meaningless? Presumably for the reasons I discussed in §2: Turner believes that programs have purposes; they are of the form "To *G*, do *A*". But, as I urged in that section, although (external, semantic) meaning can be *given* to a program, a program doesn't *need* meaning, or purpose. Programs are of the form "Do *A*".

Compositionality is a red herring: Any "meaning" that a program inherits from the atomic instructions of its programming language is a matter, not of *external* semantic interpretation, but of its "internal" (or "indigenous", or "syntactic") semantics. "Internal" semantics can be provided via procedural abstraction: named subroutines composed by sequence, selection, and repetition of atomic operations, packaged up into a "molecular" operation, and given a name so that it can be repeatedly used without having to explicitly repeat the atomic parts in a program. For example, Karel the Robot (Pattis et al., 1995) has an atomic instruction "turnleft" that causes it to rotate 90° counterclockwise. A sequence of three of these can be given the name "turnright":

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

(Note that this does not enable Karel to rotate 90° clockwise, but it gets the same effect by having Karel rotate 270° counterclockwise.)

*We* understand what the two-word, *English* expression 'turn right' means, but a Karel program, of course, does not. As Drew McDermott (1980) famously pointed out, merely naming a subroutine with an English expression does not give it the *external* meaning of that expression. But the program's ability to associate the nine-character string 'turnright' (*not* the two-word, English expression 'turn right') with that new procedure gives 'turnright' an *internal* or syntactic meaning. That is how you can get a compositional theory of (internal, syntactic) meaning from (externally, semantically) meaningless atomic operations. No (semantic) understanding by a human or a computer is required.

# 5 Conclusion

Although I find much to agree with in Turner's book, we differ on three points that I have explored here: Turner holds that computer programs must have purposes; I claim that they need not have any. Turner does not see implementation as a kind of semantic relation, whereas I do. Finally, Turner seems to think that the atomic

operations of a computer program must have a meaning that can be semantically understood. I have tried to show how the only kind of meaning that they need can be understood syntactically.

# References


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

Bechtel, W. and A. Abrahamsen (2005). Explanation: A mechanistic alternative. *Studies in History and Philosophy of the Biological and Biomedical Sciences 36*, 421–441. http://mechanism.ucsd.edu/research/explanation.mechanisticalternative.pdf.

Brooks, Frederick P., J. (1975). *The Mythical Man-Month*. Reading, MA: Addison-Wesley.

Buechner, J. (2011). Not even computing machines can follow rules: Kripke's critique of functionalism. In A. Berger (Ed.), *Saul Kripke*, pp. 343–367. New York: Cambridge University Press.

Buechner, J. (2018, Spring). Does Kripke's argument against functionalism undermine the standard view of what computers are? *Minds and Machines 28*(3), 491–513.

Chaitin, G. J. (1975, May). Randomness and mathematical proof. *Scientific American 232*(5), 47–52. http://www.owlnet.rice.edu/~km9/Randomness%20and%20Mathematical.pdf.

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

Dennett, D. C. (1987). *The Intentional Stance*. Cambridge, MA: MIT Press.

Dennett, D. C. (1995). *Darwin's Dangerous Idea*. New York: Simon & Schuster.

Dennett, D. C. (2009, 16 June). Darwin's 'strange inversion of reasoning'. *Proceedings of the National Academy of Science 106, suppl. 1*, 10061–10065. http://www.pnas.org/cgi/doi/10.1073/pnas.0904433106. See also Dennett 2013.

Dennett, D. C. (2013). Turing's 'strange inversion of reasoning'. In S. B. Cooper and J. van Leeuwen (Eds.), *Alan Turing: His Work and Impact*, pp. 569–573. Amsterdam: Elsevier. See also Dennett 2009.

Dewhurst, J. (2018). Individuation without representation. *British Journal for the Philosophy of Science 69*, 103–116.

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

Egan, F. (2010, September). Computational models: A modest role for content. *Studies in History and Philosophy of Science 41*(3), 253–259.

Egan, F. (2014, August). How to think about mental content. *Philosophical Studies 170*(1), 115–135. Preprint at https://www.academia.edu/4160744/How_to_think_about_Mental_Content; video at https://vimeo.com/60800468.

Hill, R. K. (2016). What an algorithm is. *Philosophy and Technology 29*, 35–59.

Hill, R. K. and W. J. Rapaport (2018, Fall). Exploring the territory: The logicist way and other paths into the philosophy of computer science. *American Philosophical Association Newsletter on Philosophy and Computers 18*(1), 34–37. https://cdn.ymaws.com/www.apaonline.org/resource/collection/EADE8D52-8D02-4136-9A2A-729368501E43/ComputersV18n1.pdf.

Hofweber, T. (2018). Logic and ontology. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy* (Summer 2018 ed.). Metaphysics Research Lab, Stanford University. https://plato.stanford.edu/archives/sum2018/entries/logic-ontology/.

Marr, D. (1982). *Vision: A Computational Investigation into the Human Representation and Processing of Visual Information*. New York: W.H. Freeman.

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

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

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

Peacocke, C. (1995). Content, computation and externalism. *Philosophical Issues 6*, 227–264.

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

Piccinini, G. (2004, September). Functionalism, computationalism, and mental contents. *Canadian Journal of Philosophy 34*(3), 375–410. http://www.umsl.edu/~piccininig/Functionalism_Computationalism_and_Mental_Contents.pdf.

Piccinini, G. (2006). Computation without representation. *Philosophical Studies 137*(2), 204–241. http://www.umsl.edu/~piccininig/Computation_without_Representation.pdf.

Piccinini, G. (2008). Computers. *Pacific Philosophical Quarterly 89*, 32–73. http://www.umsl.edu/~piccininig/Computers.pdf.

Piccinini, G. (2015). *Physical Computation: A Mechanistic Account*. Oxford: Oxford University Press.

Rapaport, W. J. (1986). Searle's experiments with thought. *Philosophy of Science 53*, 271–279. http://www.cse.buffalo.edu/~rapaport/Papers/philsci.pdf.

Rapaport, W. J. (1988). Syntactic semantics: Foundations of computational natural-language understanding. In J. H. Fetzer (Ed.), *Aspects of Artificial Intelligence*, pp. 81–131. Dordrecht, The Netherlands: Kluwer Academic Publishers. http://www.cse.buffalo.edu/~rapaport/Papers/synsem.pdf; reprinted with numerous errors in Eric Dietrich (ed.) (1994), *Thinking Machines and Virtual Persons: Essays on the Intentionality of Machines* (San Diego: Academic Press): 225–273.

Rapaport, W. J. (1995). Understanding understanding: Syntactic semantics and computational cognition. In J. E. Tomberlin (Ed.), *AI, Connectionism, and Philosophical Psychology (Philosophical Perspectives, Vol. 9)*, pp. 49–88. Atascadero, CA: Ridgeview. http://www.cse.buffalo.edu/~rapaport/Papers/rapaport95-uu.pdf. Reprinted in Toribio, Josefa, & Clark, Andy (eds.) (1998), *Language and Meaning in Cognitive Science: Cognitive Issues and Semantic Theory (Artificial Intelligence and Cognitive Science: Conceptual Issues*, Vol. 4) (New York: Garland).

Rapaport, W. J. (1999). Implementation is semantic interpretation. *The Monist 82*, 109–130. http://www.cse.buffalo.edu/~rapaport/Papers/monist.pdf.

Rapaport, W. J. (2003). What did you mean by that? Misunderstanding, negotiation, and syntactic semantics. *Minds and Machines 13*(3), 397–427. http://www.cse.buffalo.edu/~rapaport/Papers/negotiation-mandm.pdf.

Rapaport, W. J. (2005a, December). Implemention is semantic interpretation: Further thoughts. *Journal of Experimental and Theoretical Artificial Intelligence 17*(4), 385–417. https://www.cse.buffalo.edu//~rapaport/Papers/jetai05.pdf.

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

Rapaport, W. J. (2012, January-June). Semiotic systems, computers, and the mind: How cognition could be computing. *International Journal of Signs and Semiotic Systems 2*(1), 32–71. http://www.cse.buffalo.edu/~rapaport/Papers/Semiotic_Systems,_Computers,_and_the_Mind.pdf. Revised version published as Rapaport 2018.

Rapaport, W. J. (2017a). On the relation of computing to the world. In T. M. Powers (Ed.), *Philosophy and Computing: Essays in Epistemology, Philosophy of Mind, Logic, and Ethics*, pp. 29–64. Cham, Switzerland: Springer. Paper based on 2015 IACAP Covey Award talk; preprint at http://www.cse.buffalo.edu/~rapaport/Papers/rapaport4IACAP.pdf.

Rapaport, W. J. (2017b, Fall). Semantics as syntax. *American Philosophical Association Newsletter on Philosophy and Computers 17*(1), 2–11. http://c.ymcdn.com/sites/www.apaonline.org/resource/collection/EADE8D52-8D02-4136-9A2A-729368501E43/ComputersV17n1.pdf.

Rapaport, W. J. (2018). Syntactic semantics and the proper treatment of computationalism. In M. Danesi (Ed.), *Empirical Research on Semiotics and Visual Rhetoric*, pp. 128–176. Hershey, PA: IGI Global. References on pp. 273–307; http://www.cse.buffalo.edu/~rapaport/Papers/SynSemProperTrtmtCompnlism.pdf. Revised version of Rapaport 2012.

Rapaport, W. J. (2019). Philosophy of computer science. Current draft in progress at http://www.cse.buffalo.edu/~rapaport/Papers/phics.pdf.

Rescorla, M. (2007). Church's thesis and the conceptual analysis of computability. *Notre Dame Journal of Formal Logic 48*(2), 253–280. http://www.philosophy.ucsb.edu/people/profiles/faculty/cvs/papers/church2.pdf.

Rescorla, M. (2012, December). Are computational transitions sensitive to semantics? *Australian Journal of Philosophy 90*(4), 703–721. http://www.philosophy.ucsb.edu/docs/faculty/papers/formal.pdf.

Rescorla, M. (2014, January). The causal relevance of content to computation. *Philosophy and Phenomenological Research 88*(1), 173–208. http://www.philosophy.ucsb.edu/people/profiles/faculty/cvs/papers/causalfinal.pdf.

Rescorla, M. (2015). The representational foundations of computation. *Philosophia Mathematica 23*(3), 338–366. http://www.philosophy.ucsb.edu/docs/faculty/michael-rescorla/representational-foundations.pdf.

Shagrir, O. (2018). In defense of the semantic view of computation. *Synthese*. https://doi.org/10.1007/s11229-018-01921-z.

Shagrir, O. and W. Bechtel (2015). Marr's computational-level theories and delineating phenomena. In D. Kaplan (Ed.), *Integrating Psychology and Neuroscience: Prospects and Problems*. Oxford: Oxford University Press. http://philsci-archive.pitt.edu/11224/1/shagrir_and_bechtel.Marr's_Computational_Level_and_Delineating_Phenomena.pdf.

Smith, B. (2003). Ontology. In L. Floridi (Ed.), *Blackwell Guide to the Philosophy of Computing and Information*, pp. 155–166. Oxford: Blackwell. https://philpapers.org/archive/SMIO-11.pdf.

Sprevak, M. (2010, September). Computation, individuation, and the received view on representation. *Studies in History and Philosophy of Science 41*(3), 260–270.

Suber, P. (1988). What is software? *Journal of Speculative Philosophy 2*(2), 89–119. Revised version at http://www.earlham.edu/~peters/writing/software.htm.

Tedre, M. (2015). *The Science of Computing: Shaping a Discipline*. Boca Raton, FL: CRC Press/Taylor & Francis.

Turing, A. M. (1936). On computable numbers, with an application to the *Entscheidungsproblem*. *Proceedings of the London Mathematical Society, Ser. 2, Vol. 42*, 230–265. https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf.

Turner, R. (2018). *Computational Artifacts: Towards a Philosophy of Computer Science*. Berlin: Springer.