UNDERSTANDING UNDERSTANDING: Semantics, Computation, and Cognition

William J. Rapaport

Department of Computer Science and Center for Cognitive Science
State University of New York at Buffalo, Buffalo, NY 14260
`rapaport@cs.buffalo.edu`, `http://www.cs.buffalo.edu`

DRAFT

17 July 1996

Contents

Abstract

What does it mean to understand language? John Searle once said: "The Chinese Room shows what we knew all along: syntax by itself is not sufficient for semantics. (Does anyone actually deny this point, I mean straight out? Is anyone actually willing to say, straight out, that they think that syntax, in the sense of formal symbols, is really the same as semantic content, in the sense of meanings, thought contents, understanding, etc.?)." Elsewhere, I have argued "that (suitable) purely syntactic symbol-manipulation of a computational natural-language-understanding system's knowledge base suffices for it to understand natural language." The fundamental thesis of the present book is that understanding is recursive: "Semantic" understanding is a correspondence between two domains; a cognitive agent understands one of those domains in terms of an antecedently understood one. But how is that other domain understood? Recursively, in terms of yet another. But, since recursion needs a base case, there must be a domain that is not understood in terms of another. So, it must be understood in terms of itself. How? Syntactically! In syntactically understood domains, some elements are understood in terms of others. In the case of language, linguistic elements are understood in terms of non-linguistic ("conceptual") yet internal elements. Put briefly, bluntly, and a bit paradoxically, semantic understanding is syntactic understanding. Thus, any cognitive agent–human or computer–capable of syntax (symbol manipulation) is capable of understanding language. The purpose of this book is to present arguments for this position, and to investigate its implications. Subsequent chapters discuss: models and semantic theories (with critical evaluations of work by Arturo Rosenblueth and Norbert Wiener, Brian Cantwell Smith, and Marx W. Wartofsky); the nature of "syntactic semantics" (including the relevance of Antonio Damasio's cognitive neuroscientific theories); conceptual-role semantics (with critical evaluations of work by Jerry Fodor and Ernest Lepore, Gilbert Harman, David Lewis, Barry Loewer, William G. Lycan, Timothy C. Potts, and Wilfrid Sellars); the role of negotiation in interpreting communicative acts (including evaluations of theories by Jerome Bruner and Patrick Henry Winston); Hilary Putnam's and Jerry Fodor's views of methodological solipsism; implementation and its relationships with such metaphysical concepts as individuation, instantiation, exemplification, reduction, and supervenience (with a study of Jaegwon Kim's theories); John Searle's Chinese-Room Argument and its relevance to understanding Helen Keller (and vice versa); and Herbert Terrace's theory of naming as a fundamental linguistic ability unique to humans. Throughout, reference is made to an implemented computational theory of cognition: a computerized cognitive agent implemented in the SNePS knowledge-representation and reasoning system. SNePS is: symbolic (or "classical"; as opposed to connectionist), propositional (as opposed to being a taxonomic or "inheritance" hierarchy), and fully intensional (as opposed to (partly) extensional), with several types of interrelated inference and belief-revision mechanisms, sensing and effecting mechanisms, and the ability to make, reason about, and execute plans. This document is Technical Report 96-26 (Buffalo: SUNY Buffalo Department of Computer Science).

The world we perceive is actually an illusion created within our mind, and there instead exists another world beyond our human perceptions, which is accessible only through the powers of the imagination.

—W. R. Hohenberger[1]

You can only look at things from where you stand.

—Kate Bush[2]

The Chinese room shows what we knew all along: syntax by itself is not sufficient for semantics. (Does anyone actually deny this point, I mean straight out? Is anyone actually willing to say, straight out, that they think that syntax, in the sense of formal symbols, is really the same as semantic content, in the sense of meanings, thought contents, understanding, etc.?)

—John Searle[3]

My thesis is that (suitable) purely syntactic symbol-manipulation of a computational natural-language-understanding system's knowledge base suffices for it to understand natural language.

—William J. Rapaport[4]

Does that make any sense? Yes: Everything makes sense. The question is: What sense does it make?

—Stuart C. Shapiro[5]

[1]Quotation on a business card, of unknown origin.

[2]From a radio interview on *All Things Considered*, heard on WBFO–FM, 21 January 1994. This was part of her answer to the question of how a woman could know if she would write differently were she a man. Bush pointed out that she was *not* a man and so *couldn't* know.

[3]Searle 1993: 68.

[4]Rapaport 1988b: 85–86.

[5]In conversation, 19 April 1994.

Chapter 7

THE NATURE OF IMPLEMENTATION

7.1 IMPLEMENTATION AS SEMANTIC INTERPRETATION: THESIS.

As I said at the very beginning (§1.2.1), mental states and processes are not merely algorithms but *processes*—in the technical sense of an algorithm in execution.[1] What is the ontological status of a computer process so understood? Note that a computer process must be implemented: It is a physical device *behaving* in a certain *way*; the *way* is described (or specified) by the algorithm. Now, the physical device running the process *implements* the algorithm. The thesis I wish to put forward and examine here is this:

An implementation is a semantic interpretation.

But what is it an interpretation of? In the case at hand, a computer process viewed as an implementation is a semantic interpretation of an algorithm; the algorithm plays the syntactic role. In other cases, that which plays the role of syntactic domain to the implementation's role as semantic domain will be different sorts of things. For reasons that will become clear below, I shall use the term *Abstraction* for the syntactic domain, so my thesis is:

An implementation is a semantic interpretation of an Abstraction.

The bulk of this chapter is devoted to explicating these notions and justifying the thesis.

7.2 GOOD OLD-FASHIONED CARTESIAN DUALISM.

Computational cognitive science, or what John Haugeland (1985: 112) has termed "good old-fashioned artificial intelligence", is, I believe, good old-fashioned Cartesian dualism. The view that mental states and processes are (or are expressible as) algorithms that are *implemented in* the physical states and processes of physical devices is (a form of) Cartesian dualism: The mental states and processes and the physical states and processes can be thought of as different "substances"

[1]How can a mental *state* be a *process*? One way is if the mentalstate is implemented as a certain sequence of neuron firings. Cf. the discussion of Damasio's theories, §§2.8.3, 3.2.2.2.1.

that "interact". How might this be?

It should be clear that an algorithm and a computer are different kinds of "substance". If one considers an algorithm as a mathematical abstraction (in the ordinary sense of the term 'abstraction'), then it is an abstract mathematical entity (like numbers, sets, etc.). Alternatively, if one considers an algorithm as a text expressed in some language, then it is, say, ink marks on paper or ASCII characters in a word-processor's file. An algorithm might even be—and indeed ultimately is—"switch settings" (or their electronic counterparts) in a computer. In any case, that is a very different sort of thing from a very physical computer.

How do mind (or algorithm) and brain/body (or computer) "interact"? By the latter being a semantic interpretation—a model—of the former. More precisely, the *processes* of the brain/body/computer are semantic interpretations of (or models of) the mind/algorithm in the sense of semantics as correspondence. But this is just what we call an implementation. So, an implementation is a kind of semantic interpretation.

Note, by the way, that the mind/algorithm is also a semantic interpretation of the brain/body/computer, since, as we saw in Chapter 4, the correspondence goes both ways. How is a mind implemented? Consider a computer program: Ultimately, the program (as text) is implemented as states of a computer (expressed in binary states of certain of its components). Isn't that purely physical? Yes, but it is *also* purely syntactic; hence, *it* can have a semantic interpretation. An abstract data type, for instance, can be thought of as the semantic interpretation of an arrangement of bits in the computer (cf. Tanenbaum & Augenstein 1981: 1, 6, 45; see §2.3). This Janus-faced aspect of the bit arrangements—thought of both as a physical model or implementation of the abstract algorithm and as a syntactic domain interpretable by the algorithm and its data structures—is simply our old friend the model muddle (§2.6).

Now, is this really good old-fashioned Cartesian dualism? Is mind–body interaction really semantic interpretation or implementation? Or might this semantic/implementational view be more like some other theory of the mind?

- It is not parallelism, since there really is a *causal* interaction: The algorithm (better: the process) *causes* the physical device to behave in certain ways.
- So it's also not epiphenomenalism, either. (And the device—or its behavior—can produce changes in the program, as in the case of self-modifying programs, or even in the case of a system competent in natural language whose knowledge base (part of the software) changes with each interaction.)
- Could it be a dual-aspect theory? Perhaps: Certainly, the physical states and processes are one "level of description", and the mental states and processes are another "level of description" *of the same (physical) system*. But talk of levels of description seems to me to be less illuminating than the theory of semantics as correspondence. More to the point, neither "level" is a complete description of the system: The algorithm is not the process, nor can one infer from the algorithm what the future behavior of the process will be (i.e., the process can behave in ways not predictable by the programmer (cf. Fetzer 1988, 1991). And even a complete physical description of the system would not tell us *what* it is doing; this is one of the lessons of functionalism.

So dualism is at least plausible. Do the physical states and processes produce mental ones?

Here is where the problem of qualia—i.e., subjective qualitative experiences, including pain and physical sensations—enters. We shall have something to say about it later (§7.6.2).

7.3 IMPLEMENTATION AS SEMANTIC INTERPRETATION: EVIDENCE.

> ... the terms of art employed in computer science—... *implementation* ...—will ultimately be definable only with reference to ... attributed semantics" (Smith 1982b: 9.)

Let's briefly review the data we first looked at in §2.3. Some of those pairs of syntactic and semantic domains were clear examples of implementations; others can be so thought of. There are three paradigmatic cases:

semantic domain		syntactic domain
17. a computer program	is an implementation of	specifications
18. a computational process	is an implementation of	a computer program or algorithm
an implementation	is an implementation of	an abstract data type

Thus, we implement a program when we compile and execute it, we implement a set of specifications when we write a computer program, and we implement an abstract data type such as a stack when we write code (in some computer programming language) that specifies *how* the various stack operations (such as *push* and *pop*) will work.

There are several other cases that, while we don't, normally, use the term 'implementation' in discussing them, are clearly of the same type as the paradigms above, e.g.:

semantic domain		syntactic domain
4, 5. a performance	is an implementation of	a musical score or play-script
10. a house	is an implementation of	a blueprint
15. a SNePS node	is an implementation of	a concept

Finally, there are a couple of examples that can clearly be thought of in the same way:

semantic domain		syntactic domain
2. a set-theoretic model	is an implementation of	a formal theory
16. a Sein-correlate (actual object)	is an implementation of	a Meinongian object

My thesis is not only that all implementations are semantic interpretations of a syntactic domain. It is also that all semantic interpretations can be seen as implementations.

7.4 WHAT EXACTLY *IS* AN IMPLEMENTATION?

> On the one hand, we have a very elegant set of mathematical results ranging from Turing's theorem to Church's thesis to recursive function theory. On the other hand,

> we have an impressive set of electronic devices which we use every day. Since we have such advanced mathematics and such good electronics, we assume that somehow somebody must have done the basic philosophical work of connecting the mathematics to the electronics. But as far as I can tell that is not the case. On the contrary, we are in a peculiar situation where there is little theoretical agreement ... on such absolutely fundamental questions as, What exactly is a digital computer? What exactly is a symbol? What exactly is a computational process? Under what physical conditions exactly are two systems implementing the same program? (Searle 1990: 24.)

7.4.1 Implementation in Computer Science.

Let's look first at the notion of implementation in its home territory: computer science. This is not to say, I hasten to add, that the term is not used elsewhere or that it does not antedate computer science—which, as we will see, it certainly does. But since the term is so ubiquitous in computer science, it is a good place to start. Once we have a clear idea of how computer scientists use the term, we'll look at how ordinary folks use it.

Given the ubiquity, it is rather surprising how few texts even *try* to define the notion. For instance, all that Michael Marcotty and Henry Ledgard say in *Programming Landscape* (1986), a standard text on programming languages, is that "the **realization** of a programming language in a computer system is called the *implementation* (Marcotty & Ledgard 1986: 8, my boldface). 'Realization', of course, is left undefined. Taken literally, it means "to make real", where 'real' is opposed to 'imaginary' or perhaps 'abstract'. Before exploring this a bit further, note, first, that this makes it seem that the physical medium is important and, second, that to "realize" could be to establish a Sein-correlate, which can be generalized to different possible worlds. Thus, intelligence, say, could be realized in several different (physical) media: This is the notion of "multiple realizability", to which we shall also return.

"Realization" itself is a rather interesting notion. According to the new *Oxford English Dictionary*, 'real' comes from the Latin for "pertaining to things", and its philosophical meaning, in part, is "having an existence in fact and not merely in appearance, thought, or language" (Simpson & Weiner 1989, Vol. **00**, **[???]** p. 272). What is it that is made real when it is "realized"? Presumably, something that exists "merely in appearance, thought, or language"—in short, something "Abstract". To *realize* is, in part, "To make real, to give reality to (something merely imagined, planned, etc.) ... In common use from *c* 1750 with a variety of objects, as ideas or ideals, schemes, theories, hopes, fears, etc. ..." (Simpson & Weiner 1989, Vol. **00**, **[???]** p. 277). Note how *psychological* or *intentional* these realizable things are. Note, too, especially in connection with the example of a performance of a play, that where English talks of a *director*, French talks of a *réalisateur* (a realizer): At least for francophones, plays and movies are *implementations* (of scripts).

7.4.1.1 Hayes's notion of implementation.

One computer-science text that says a bit more about implementation is John P. Hayes's *Computer Architecture and Organization* (1988). The term first occurs in the following passage:

> With the advent of the [IBM] System/360, the distinction between a computer's architecture and its **implementation** became apparent. As defined by the System/360 designers ..., the *architecture* of a computer is its **structure and behavior as seen by an assembly-language programmer** .... The *implementation* ... refers to the **logical and physical design techniques** used to **realize** the architecture in any specific instance. Thus all the members of the S/360–370 series share a common architecture, but they have many different implementations. For example, some S/360–370 CPUs employ fast hardwired control units, whereas others use a slower but more flexible microprogrammed approach to implementing the common instruction set. (Hayes 1988: 47, my boldface.)

Architecture is concerned with structure and behavior; these are functional, Abstract aspects. This is not to say that it is not *detailed*, however, since the architecture is "seen by an assembly-language programmer," who must know all about the details of registers, control, etc., although he or she does not have to worry about what a register looks like or how the control is actually carried out.

Implementation is concerned with "logical and physical design techniques". I am not sure what 'logical' means here, but 'physical' is clear: Implementations are the physical realizations of Abstractions. I would call a *physical* realization a special case, however. The full explication of 'implementation' requires a third term besides the implementation and the Abstraction:

$$I \text{ is an implementation of } A \text{ in medium } M$$

where A is the Abstraction and M could be physical or set-theoretical, etc. For instance—as we will see below—it is common in the study of data structures in computer science to talk about implementing a stack by means of a linked list, implementing the list in a programming language (say, Pascal), "implementing"—i.e., compiling—the Pascal program in some machine language, and then implementing the machine-language program in a real computer. As we progress along this sequence (this correspondence continuum), the implementing media begin as Abstractions themselves and gradually take on a more "physical" nature.

Perhaps you will object that program compilation should not be treated as an implementation. Hayes, however, would not object: "a sequence of ... [machine] instructions is needed to *implement* a statement in a high-level programming language such as FORTRAN or Pascal" (Hayes 1988: 209, my italics). So, to implement is to "realize" *in* some medium, which might be a physical medium or could be some *domain* or *language*. To implement is to *construct* something, out of the materials at hand, that has the properties of the Abstraction; it could also be to *find* a counterpart that has those properties. Both tasks are semantic.

Hayes, indeed, speaks of semantics in this context:

> Because of the complexity of the operations, data types, and syntax of high-level languages, few successful attempts have been made to construct computers whose machine language directly corresponds to a high-level language .... There is thus a *semantic gap* between the high-level problem specification and the machine instruction set that **implements** it, a gap that a compiler must bridge. (Hayes 1988: 209, my boldface.)

What is the gap? Presumably, that (say) the specific operations, data types, etc., of the high-level language don't correspond directly to anything in the machine language: Pascal, e.g., has the

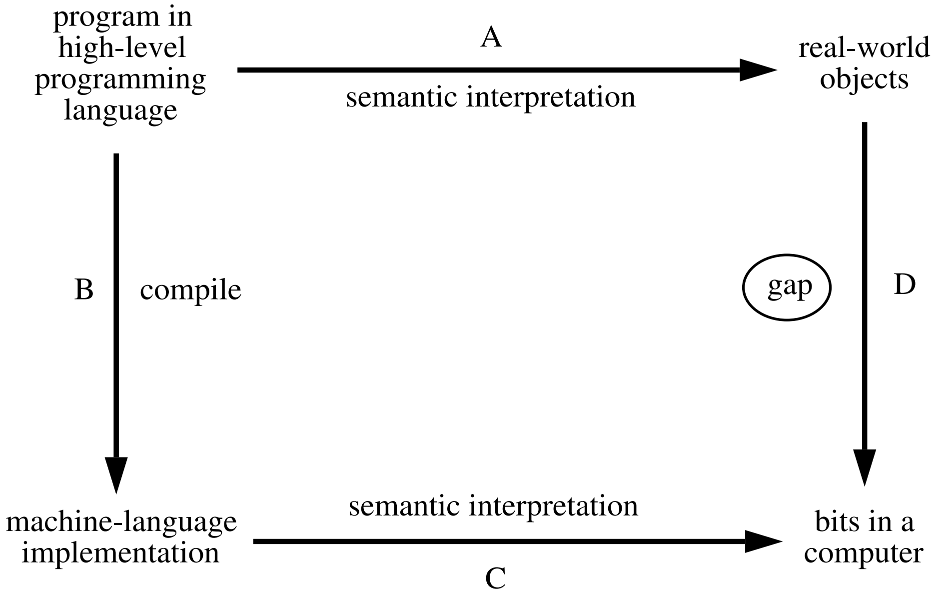

Figure 7.1: A program in a high-level programming language is semantically interpreted by real-world objects. It is also compiled into a machine-language implementation, which, in turn, is semantically interpreted by bits in a computer. What is the relation between these two semantic interpretations?

"record" data type, but my VAX's machine language probably doesn't. So, a compiler is needed to show how to construct or implement records in the machine language. (This notion of construction is, perhaps, related to the notion of the construction of the rational numbers from the integers and, hence, to the notion of theory *reduction*. We will return to this in §7.5.3.)

Why does Hayes call this a *semantic* gap? It's a bit like the fact that one natural language might not have a single word corresponding to some single word in another natural language. John Sowa (personal communication, 29 November 1993) gives the example of Russian, which has a term, 'ruka', referring to what in English has to be referred to as the hand+forearm. Of course, one can translate between the languages by defining the word in terms of others (perhaps with a cultural gloss (cf. Jennings 1985, Rapaport 1988: 102). But why is this *semantic* rather than *syntactic*?

A possible interpretation[2] is shown in Figure 7.1, which specifies four relations:

A. Presumably the semantic interpretation of a program written in a high-level programming language is the relation between, on the one hand, data structures (say) in a Pascal program (say, a record representing a student viewed as consisting of a name, an age, a class, a major, a student-number, and a grade-point average) and, on the other, an actual student in the real world.

B. The compilation relation is, or includes, the relation between that student-record data structure and a construct of data types in the machine language. Note that both A and

[2] Due to my colleague Bharadwaj Jayaraman (personal communication).

B, on my theory, are semantic relations.

C. At first sight, it seems odd to semantically interpret the machine-language program by bits. Why not map the machine-language program into the real-world objects? But, after all, if all semantic relations are merely correspondences (and vice versa), then this relation between the machine-language program and bits is just another one. After all, we could also have mapped the Pascal program into computer bits—in fact, via B and C, we have! So, given a machine-language program, one can interpret it in terms of bits in the computer. Arguably, in fact, having these two distinct interpretations of two distinct (albeit input–output—equivalent) programs is appropriate. Where the machine-language program talks of registers, the Pascal program talks of "students" (or student-records). So it is appropriate to understand the Pascal program as a "mathematical model" of such real-world objects as students and to understand the machine-language program as a "mathematical model" of such (also real-world) objects as bits in a computer.

D. So the gap concerns the relation between the real-world objects (such as students) and computer bits, since *both* are semantic interpretations of the Pascal program.

What, then, is this relation D? It could be *simulation*: The computer bits simulate the student. Or it could be *implementation*: The computer bits are a computer implementation (an implementation in the medium of the computer) of the student. Before exploring some of these options further, let's look more closely at one of the standard uses of 'implementation' in computer science: the implementation of an abstract data type.

7.4.1.2 Abstract data types.

The notion of an abstract data type and its "implementation" is one of the most common uses of 'implementation' in cognitive science. There is a relatively informal use of the notion, as it appears in programming languages such as Pascal and as it is taught in introductory computer-science courses, and there is a more formal, mathematically precise use. Let's look at the informal one first, informally, in order to provide a bit of background.

7.4.1.2.1 The informal notion of implementation. A stack is a particular kind of data structure, often thought of as consisting of a set of items structured like a stack of trays in a cafeteria: New items are added to the stack by "pushing" them on "top", and items can be removed from the stack only by "popping" them from the top. Thus, to define a stack, one needs (a) a way of referring to its top and (b) operations for pushing new items onto the top and for popping items off the top. That, more or less (mostly less, since this is intended to be informal), is a stack defined as an abstract data type.

Now, Pascal does not have the stack as one of its built-in data types (as it does arrays, records, or sets). So, if you want to write a Pascal program that manipulates data structured as a stack, you need to "implement" a stack in Pascal. This can be done in several ways; i.e., there are several ways to implement the abstract data type *stack* in Pascal. Here's one way:

1. A stack, s, is implemented as a 1-dimensional array, $A[0], \ldots, A[n]$, say, for some n;

2. $top(s)$ is defined to be a 1-argument function that takes as input the stack s and returns as output $A[0]$ (i.e., $A[0]$ is the implementation of the "top");

3. $push(s, i)$ is defined to be a 2-parameter procedure that takes as input the stack s and an item i (of the type allowed to be in the array), and yields as output the stack modified so that $A[0] := i$, and $A[j] := A[j-1]$ (i.e., each item on the stack is "pushed down"); and,

4. almost finally, $pop(s)$ is defined to be a 1-argument function that takes as input a stack s and returns as output the item on the top of s (i.e., $top(s)$) while moving all the rest of the items "up" (i.e., $A[j] := A[j+1]$).

I said "almost finally" because—as should be obvious—there is some bookkeeping that has to be taken care of:

5. We have to specify what happens if the stack "overflows" (as when we try to push an $(n+2)$nd item onto a stack implemented as an $(n+1)$-element array).

6. We have to specify what happens to the "last" item when the top is popped (does the array cell that contained that item still contain it, or does it become empty?), etc.

These (as well as the limitations due to the type allowed to be in the array) are what are called "implementation details", since the abstract data type *stack* "doesn't care" about them (i.e., doesn't—or doesn't *have to*—specify what to do in these cases).

Here's another way to implement a stack in Pascal. Do everything as before, but let $top(s) := A[n]$. This implementation of the *stack* abstract data type is "inverted" with respect to the first one. The inversion, however, is (a) a ("mere") implementation detail and (b) undetectable in the program's input–output behavior. (Start thinking about inverted spectra here; we'll come back to them in §7.6.2.)

And here's a third way: Use Pascal's *pointer* data type to implement a stack as a "linked list". I won't bore you with all the details, but here are a few. First a linked list ('list', for short) is itself an abstract data type. It is a sequence of items whose three basic operations are (1) $first(l)$, which returns the first element on the list, (2) $rest(l)$, which returns a list consisting of all the original items except the first, and (3) $make\text{-}list(i, l)$ (or $cons(i, l)$), which recursively creates a list by putting item i at the beginning of list l. Lists can be implemented in Pascal by, e.g., 2-dimensional arrays (here, the first item in each two-cell unit of the array is the list item itself, and the second item in the two-cell unit is an index to the location of the next item) or by means of "pointers" (each item on the list is implemented as a two-element "record", the first element of which is the list-item itself and the second element of which is a pointer to the next item). A stack, s, finally, can be implemented as a list, l, where $top(s) := first(l)$, $push(s, i) := make\text{-}list(l, i)$, and $pop(s)$ returns $top(s)$ and redefines the list to be $rest(l)$.

So: stacks can be implemented as arrays or as lists, and lists can be implemented as arrays or as records with pointers. Abstract data types can implement other abstract data types, or they can be implemented "directly" in the given data structures of a programming language. What's going on here?

7.4.1.2.2 The formal notion of implementation. To see what's going on, we need to look at some of the more formal approaches to the definition and implementation of abstract data types.

7.4.1.2.2.1 ***Guttag, Horowitz, and Musser.*** In "The Design of Data Type Specifications" (1978), John V. Guttag, Ellis Horowitz, and David R. Musser assert that "the process of design (of data types) consists of specifying ... operations to increasingly greater levels of detail until an executable implementation is achieved" (p. 61). So, the implementation appears to be merely a more detailed version of the original "specification". The implementation details are essential for *executability* but not, presumably, for specifiability. So the implementation details serve a purpose, but one distinct from the original abstraction.

What is the abstraction? "A *data type specification* (or abstract data type) is a representation-independent formal definition of each operation of a data type. Thus, the complete design of a single data type would proceed by first giving its specification, followed by an (efficient) implementation that agrees with the specification" (p. 61). So, implementations—the *detailed* specification of the operations—must "agree with" the *abstract*—or *undetailed*—specification; the implementation must *satisfy* the definitions. That, of course, needs to be made precise, but it is more than suggestive of semantic interpretation. If the abstraction is supposed to be "representation-independent", then perhaps the implementation *is* the (or, a) representation. (Note that there can be more than one implementation; at the very least, there can be "efficient" and inefficient ones.)

Guttag et al. give "a brief example of the implementation of one data type, Queue ..., in terms of another, CircularLists" (p. 74). So, as we noted before, abstract data types can implement each other. This is done as follows: "We first give, in a notation very similar to that for the specification, an implementation of the Queue type consisting of a *representation* declaration and a *program* for each of the Queue operations in terms of the representation" (p. 74). In the example, the representation "medium" is CircularList, and the "programming language" consists of the operations of CircularLists. So, an implementation of an abstract data type consists of a representation and programs, where the programs implement the abstract data type's operations. This is done as follows: Each operation of the abstract data type is ... defined? explicated? implemented? ... in terms of an operation of the implementing medium (the implementing abstract data type), *after* first representing each abstract-data-type entity (term) by a term of the implementing abstract data type. So, terms get interpreted by, or mapped into, elements of the interpreting domain, and predicates (operations) are mapped into predicates of the interpreting domain. So, implementation *is* semantic interpretation.

7.4.1.2.2.2 ***Goguen, Thatcher, and Wagner.*** A somewhat more detailed and philosophically sophisticated approach is to be found in J. A. Goguen, J. W. Thatcher, & E. G. Wagner's "An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types" (1978). The mathematical details they present are, I think, irrelevant to our inquiry in this chapter, but the overall picture they offer is useful, so let me attempt to summarize it here.

They begin by observing that "the term *abstraction* in computer science ... has been used in at least three ways which are distinct but related" (p. 82):

1. An abstraction is "a mathematical model or description of something" (p. 82).
2. An abstraction is "the process (or result) of generalizing" (p. 83).
3. An abstraction is "a concept" considered "independent[ly] of its representation" (p. 83).

Examples of (1) are "'abstract machines' as opposed to real hardware" (p. 82) and "abstract implementations", as "when one uses sets, sequences, or other mathematical entities to model some computational process or structure" (p. 83). So, a mathematical model is an abstraction of some real-world entity; as such, the abstraction seems to play the syntactic role. On the other hand, the implementation of a queue by a circular list (or a stack by a linked list) is an "abstract implementation", yet here it clearly plays the *semantic* role. This, I take it, is a further example of the muddle of the model in the middle.

In sense (2), abstractions ignore details. This contrasts nicely with the notion of an implementation as *providing* details. Presumably, an abstraction in sense (1) might ignore details, and hence be an abstraction in sense (2), but not necessarily. For instance, although the (admittedly informal) abstract notion of stack that I presented in §7.4.1.2 ignored the details imposed by the finiteness of the stack, we could have had an equally abstract presentation that payed attention to those details (yet could be implemented as a (finite) array, an "inverted" array, or a (finite) list). However, *merely* ignoring details does not by itself yield an abstract model in sense (1), because it might not be a *mathematical* model.

It is the third sense of 'abstraction' that Goguen et al. take to be the relevant one for abstract data types (cf. p. 81). The "representation" that such an abstraction is independent of has to do with notation, or the manner in which it is expressed:

> For example, "abstract syntax" considers syntactic structure independently of whether it is represented by derivation trees, parenthesized expressions, ... or whatever. This notion of abstract syntax is useful ... in specifying the semantics of a programming language in a manner independent of how it is implemented ....
>
> More to the point, an abstract data type is supposed to be independent of its representation, in the sense that details of how it is implemented are to be actually hidden or "shielded" from the user: He [sic] is provided with certain operations, and he only needs to know what they are supposed to do, not how they do it. (Goguen et al. 1978: 83; cf. Parnas 1972.)

That is, the programmer can deal directly with the abstract data type and ignore its implementation; one deals with it at a "high level". Consistent with our view that an implementation is a semantic interpretation, Goguen et al. observe: "Note that what is usually called an 'abstract implementation,' that is, an implementation described by sets, sequences, etc., is *not* an 'abstraction' in the above sense; rather, it is a *particular*, but rather undetailed, implementation" (p. 83). So, an abstraction in sense (1) is not necessarily an abstraction in sense (3). It is undetailed, presumably because the implementing medium (the implementing abstract data type) is *itself* abstract (in sense (2)). Still, the mathematical model is a semantic model.

Now, what is this abstraction of the third kind? Goguen et al. note that it has to do with equivalence classes, or what they call "isomorphism classes" (p. 83). They define an abstract data type as "the isomorphism class of an initial algebra in a category" of many-sorted algebras (pp. 88, 90). And they note that "An implementation is necessarily made within a specific framework, such as a particular programming language or machine" (p. 135); i.e., an implementation requires an implementing *medium*, or "framework".

Their mathematical "approach is to model an implementation framework as an algebra, with the elements of the carrier(s) being concrete data representations (machine states, primitive data types) and its operations the given basic operations (machine operations, basic instructions, programs) in these data representations" (p. 135). Note that they are *modeling* the implementing *medium* and that they do so by the *same* kind of entity as for an abstract data type, namely, an algebra! The implementation *itself*, of course, is something "physical"; it is merely being *described* algebraically.

The heart of the matter is expressed by them in their mathematical set-up as follows:

> Let B denote the implementation algebra .... [Let $T_{\Sigma,\epsilon}$ be] the specification algebra. The question now is, What relationship between $T_{\Sigma,\epsilon}$ and B constitutes an *implementation*? (p. 136.)

This is indeed the question: What is the relationship between an Abstraction (a "specification algebra") and an Implementation (an "implementation algebra")? (Note, as with Smith, that B itself is (merely) a representation or model of the actual, physical implementations.) Goguen et al.'s answer is that the relationship is a structure consisting of B, a mapping from (roughly) $T_{\Sigma,\epsilon}$ to B, and a "congruence" (a family of equivalence relations on (roughly) T's image in B) (p. 138). The core of this is, first, the mapping from the Abstraction to the Implementation, which is, on my theory and consistent with the view of Guttag et al., a *semantic interpretation*, and, second, the "congruence". The latter is a very special, intricate kind of isomorphism, one that "factors out" (or "divides out"—they use quotient spaces) the "implementation details". So, B (or that which B is a mathematical model of) implements an Abstraction T if and only if B is a domain of semantic interpretation of T, ignoring the implementation details.

An example of what I think is going on might help. Consider the abstract data type *stack*, and consider two specific implementations of it in Pascal, using an array $A[0], \ldots, A[n]$ with *top* implemented in one as $A[0]$ and in the other as $A[n]$. In both implementations, *top* is implemented as a specific element of the array. That it is $A[0]$ in one and $A[n]$ in the other is an implementation detail.

But can those details really be ignored? In this case, perhaps; in others, perhaps not—we'll return to this (§7.6). Before we do, though, we need to broaden our scope and consider to what extent the notion of an implementation is applicable *outside* of computer science and then to look at other candidate interpretations of 'implementation'.

7.4.2 Implementation Outside of Computer Science.

7.4.2.1 Music.

Some of the clearest examples outside of cognitive science of what could be called 'implementation' come from music (cf. §2.3, example 5). This ought not to be surprising: After all, a music score is very much like a computer program or algorithm, and the musician-plus-instrument (or conductor-plus-orchestra) plays a role very much like that of the computer. A musical score is *not*, of course, *mathematically* an algorithm, since much is left open to "interpretation" by the musician (e.g, tempo, dynamics, optional repeats, phrasing, etc.). Nonetheless, it *is* a set of "instructions" which, when followed or executed, produce a certain output. The "process" consisting of the musician playing that music on an instrument can plausibly be said to *implement* the score. The score can be thought of (indeed, it *is*) a piece of syntax; the playing of the score provides a "semantic interpretation" of it.

Now, as we saw above (§7.4.1.1), an implementation requires an implementing medium. And, as should be evident, there can be many different media, hence many different implementations (the common core of which can be captured by the mathematical techniques of Goguen et al.). We find the same thing in music: A given score can normally be played on a variety of instruments, modulo a few changes necessitated by the nature of the instrument. Such changes, as well as the particular features of the instrument, constitute "implementation details". Often, these change the nature of the work, for good or bad: "a [piano] transcription [of a symphony] can hold a prism up to a familiar work, showing it in a new light" (Pincus 1990: **00**). **[???]** That is, a piano transcription of a symphony is an *interpretation* of it—or, rather, *another* interpretation of "the work", i.e., of an abstract data type (the score) of which both the symphony *and* the piano transcription are (semantic) interpretations or implementations. The implementation is also, of course, an "interpretation" in the ordinary sense: In an essay on "historically accurate performances" of music, Charles Rosen (1991) speaks of "the essential gap between the composer's conception of a work of music and the *multiple possibilities of realizing it in sound*" (p. 50, my italics). The "conception" is the abstract data type; the "multiple possibilities" are different implementations. Much the same can be said, *mutatis mutandis* for scripts and productions of plays (or scripts and movies) (again, recall §2.3, examples 6, 7, 10).

Music, though, has another aspect to it reminiscent of the situation with computers. On a radio quiz show called "My Music" (radio station CJRT FM 91.1, Toronto; 28 November 1990), a tune was played on a piano for the panel to identify. One panelist "misidentified" the piece. The host said, "You were listening to a tune from Puccini's *Girl of the Golden West* but thought you were listening to Lloyd Weber's *Phantom of the Opera*," to which the panelist drily replied, "It was both." Another, perhaps more familiar, example would be "76 Trombones" and "Good Night My Someone", both from *The Music Man*, both the same tune, albeit different words, tempi, and arrangements—in short, two different implementations of a single Abstraction.

Here, though, we can see another phenomenon at work, too, for these cases also seem very much like that of the Morning Star and the Evening Star, or the conversation that is simultaneously about mathematical lattice theory and chemical lattices (§2.7.1), or the computer program that can be taken either (or: both) as playing chess or (or: and) analyzing a Civil War battle. Given the Morning Star/Evening Star way of thinking about it, it would follow that semantic interpretations and, hence, implementations, are (perhaps not surprisingly) intensional entities and that we should expect to find *de re*/*de dicto* phenomena. For instance, the panelist on *My Music* was listening to (or perceiving) *de re* an Abstraction; *de dicto*, however, he was listening to Puccini, not Lloyd Weber. *De dicto*, one describes oneself as perceiving an interpretation. One interprets what one perceives; possibly, one *can't just* perceive the Abstraction *de re*.

7.4.2.2 Language.

Another non–computer-science example of implementation was discussed by Sellars:

> [In the context of chess,] ... attention must be called to the differences between 'bishop' and 'piece of wood of such and such shape'. ... [the former] belongs to the rule language of chess. And clearly the ability to respond to an object of a certain size and shape *as a bishop* presupposes the ability to respond to it as an object of that size and shape. But it should not be inferred that 'bishop' is 'shorthand' for 'wood of such and such size and shape' ... 'Bishop' is a counter in the rule language game and participates in linguistic moves in which ... the ... longer expression does not .... (Sellars 1955/1963: 343, §56.)

"Being a bishop" is a nice example of what I am calling an Abstraction. Here, a bishop is implemented as a certain piece of wood (cf. also the example in §3.2.1). It could also, as Sellars observes, be implemented by a Pontiac if the chess game is played in Texas, where everything is supposed to be bigger:

> ... the term 'bishop' as it occurs in the language of both Texas [where it is "syntactically related ... to expressions mentioning different kinds of cars" (p. 344, §59)] and ordinary chess can be correctly said to have a common meaning—indeed to mean the bishop role, embodied in the one case by pieces of wood, and in the othe by, say, Pontiacs .... (Sellars 1955/1963: 348, §62.)

This situation is depicted in Figure 7.2. Here, we have an Abstraction (Chess) and two implementations (the ordinary Staunton pieces and the Texas pieces). We assume that the pieces that play the role of the bishop are both *called* 'bishops'; 'bishop' means the same thing in both implementations, namely, the Bishop Abstraction. That role is "embodied as"—i.e., *is implemented by*—a Pontiac in Texas and a certain ♗-shaped piece of wood in the Staunton set. The words 'bishop' as they occur in the two different languages refer to different entities (the language-entry and -departure rules in Sellars's language games differ).

Language provides non–computer-science examples of implementation in a variety of ways. For one thing, as Roy Harris has observed (Harris 1987: xi, Ch. 6), words can be considered as representations—hence, implementations—of ideas. For another, if language can be thought of as an Abstraction (as, perhaps, Chomsky's theory of universal grammar would have it), then it can be implemented in a variety of ways: first, by spoken languages (implemented in the medium of speech) as well as by signed languages (implemented in the medium of space) (cf. Coughlin 1991, cited in §2.3, example 18), and, second, in many ways in both spoken and signed languages (e.g., French, English, etc., and American Sign Language, British Sign Language, etc.).

Another example from language is also due to Sellars. At the beginning of "Some Reflections on Language Games" (1955/1963), he distinguishes between "*obeying* rules[, which] involves using the language in which the rules are formulated, [and] *conforming* to rules[, which] does not" (p. 322, §4). He goes on in the same passage to observe that "once one has learned" the metalanguage in which the rules are formulated, "one may come to *obey* the rules for" the language whose rules for use are formulated in the metalanguage. This is precisely how I view the "classical" or "symbolic" paradigm in AI in general and in natural-language competence in particular. It is a feature of the "multiple realizability" of natural-language competence: In us, *conceivably*, natural-language competence is implemented by rule-*conforming*; in a computer, it could be implemented by rule-*following*. This assumes, of course, that there are rules to which we conform. But, at least, the

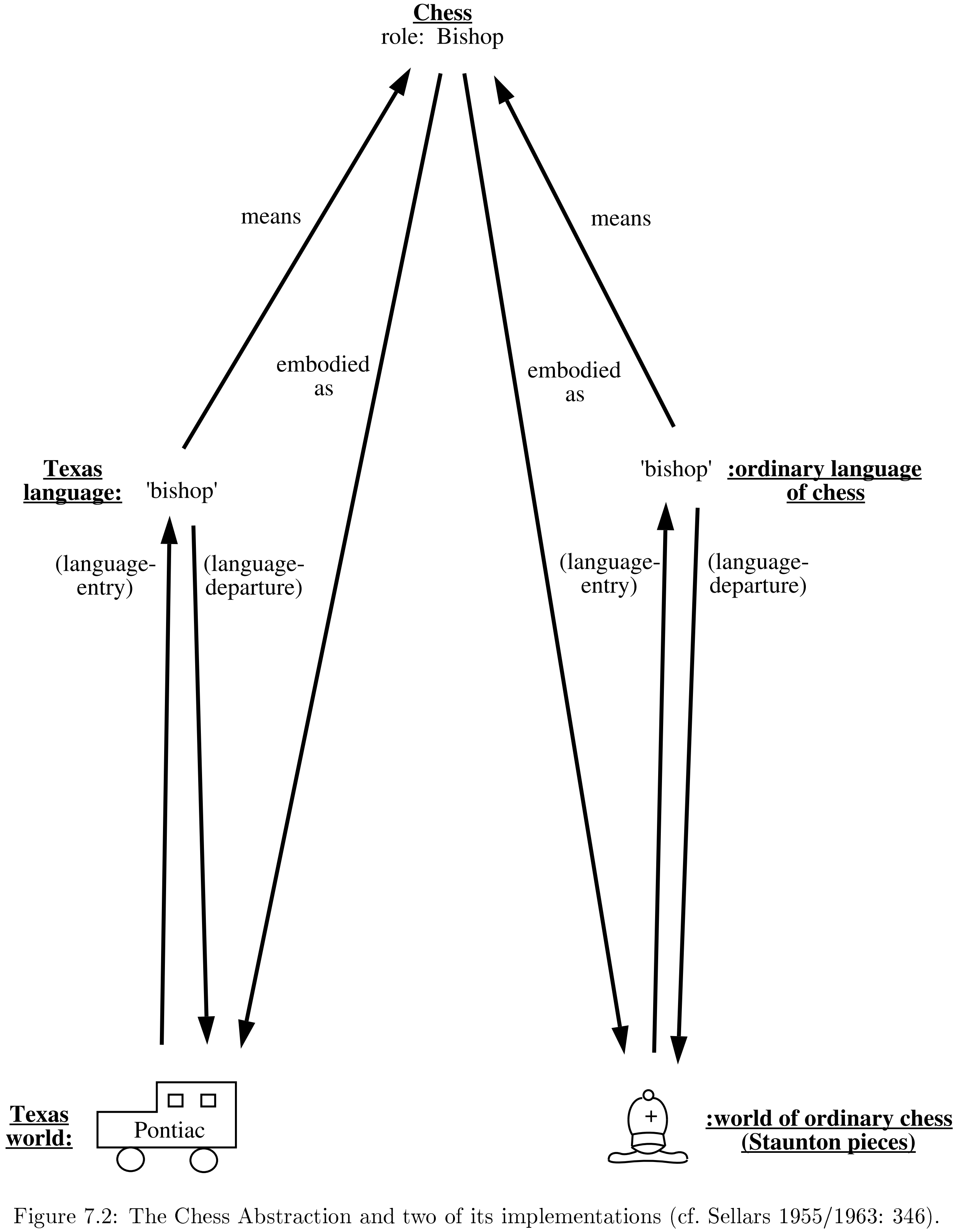

Figure 7.2: The Chess Abstraction and two of its implementations (cf. Sellars 1955/1963: 346).

search for such rules is a worthwhile scientific goal.[3]

Finally, there is the issue of the medium. I once saw a discarded library catalog card and thought, "How sad," because that's where the information "really" is, not in the electronic "card catalog", which is just a bunch of 0s and 1s (and not really even that). But that's wrong, of course. The information—the very *same, identical* information—is stored (*implemented*) electronically (but cf. N. Baker 1994). After all, the information on the card is just a bunch of ink marks on paper (cf. the "mystery of reading", §5.1). The medium *isn't* the message; rather, the message is *implemented* in the medium. There is a sense, though, in which the medium *is* at least *part* of the message. That's the sense in which qualia reside in the implementation details (again, cf. N. Baker 1994), or perhaps the sense in which a robot with a different body might have different concepts from humans. (On the former, see §7.6.3, below; on the latter, see §3.2.2.2.4, above.)

7.4.2.3 Mind.

My last example of implementation outside of computer science is the mind–body problem. It would be too simplistic to say that the mind is implemented in the brain, but that is certainly the central insight. If AI "succeeds", then in its Golden Age we will have a set of algorithms for cognition that, when implemented, yield a cognizing entity. If the algorithms are strongly equivalent to those used by the human mind (i.e., not merely input–output equivalent, but equivalent in detail), then we would say that they were implemented in the human brain:

> Symbolists emphasize that the symbolic level (for them, the mental level) is a natural functional level of its own, with ruleful regularities that are independent of their spec. **[???]** physical realizations. For symbolists, this implementation independence is the critical difference between cognitive phenomena and ordinary physical phenomena .... (Harnad 1990: 336.)

So, implementation independence might be taken as a mark of the mental. There are two ways of looking at this: (1) *Being syntactic* is a mark of the mental. (2) *Being interpretable* is a mark of the mental. The second of these subsumes Brentano's claim that intentionality is the mark of the mental.

7.4.3 Definitions.

Before summing up, let's take another look at the *Oxford English Dictionary.* The *noun* 'implement' comes from the Latin for "a filling up", as in "that which serves to fill up or stock (a house, etc.)", and from the Old French for "to fill, fill up" in the sense of "completing" (Simpson & Weiner 1989, Vol. VII, p. 721). This suggests "filling in the details", which an implementation in the sense we are concerned with certainly does.

The *verb* is of more recent origin and has three senses, all with citations beginning in the 19th century (p. 722):

[3]Unfortunately, Sellars goes on to argue that the obeying–conforming dichotomy is false (p. 325, §12), but he replaces it with a distinction between "'pattern governed' and 'rule obeying' behavior" (p. 327, §16; cf. §§18–19), which will do as well.

(a) "To complete, perform, carry into effect (a contract, agreement, etc.); to fulfil (an engagement or promise)." This is the earliest sense to be cited (1806)—implementing an obligation.

(b) "To carry out, execute (a piece of work)." Here, the citation is from 1837: implementing an invention.

(c) "To fulfil, satisfy (a condition)." This was used as early as 1857: implementing the "mechanical requisites of the barometer ... in ... an instrument".

Senses (b) and (c) seem closest to our concerns: Sense (b) relates to an Abstraction, and (c) relates to the implementation of an Abstraction—to satisfying the conditions of the Abstraction, or having the properties of the Abstraction. More recent senses (with citations from 1926 and 1944) don't clarify much; curiously, none of the citations come from computer science.

Recall that "implementation" is a relational notion, whose full context is always:

I is an "implementation" of an "abstraction" A in some "medium" M.

I have suggested that the notion of *Abstraction* be a generalization of the notion of an abstract data type, and we have seen from examples that syntactic systems and conceptual-role systems are Abstractions ripe for "implementing". Must Abstractions be abstract, that is, non-spatiotemporal? If so, then they would contrast nicely with a *physical* or *concrete* interpretation of an "implementation"—i.e., with the "medium" always being spatiotemporal. But we have seen that one Abstraction can implement another. So this characterization won't do. Instead, I suggest that we leave the notion unrefined for now except as that which can be implemented in some medium.

What about the medium? It could be abstract or concrete, giving rise to two varieties of implementation. An "abstract implementation" would be a specification, a filling-in of details, of an Abstraction. For instance, in top-down design, each level (except possibly for the last) is an "abstract implementation" of the previous one: I begin preparing my courses with a bare-bones course outline and successively refine it by adding details; or: I start solving a problem algorithmically by writing an algorithm in "pseudocode" and, by "stepwise refinement", fill in the details (e.g., pseudocode the procedures), until I finally encode it in, say, Lisp.

A "concrete implementation" would exist in a physical (or spatiotemporal) medium. It would necessarily have more details filled in, namely, those due to, i.e., contributed by, the medium. For example, my actually standing in front of the class, lecturing, is a concrete implementation of my final course outline. The actual words I say, the actual piece of chalk I use, etc., are all implementation details, filled in to "the last detail" by the very nature of the real, spatiotemporal events. Similarly, the actual execution of my Lisp program (perhaps after having been compiled into—i.e., further implemented in—machine language)—the *process*—is its concrete implementation. Note that both abstract and concrete implementations are semantic interpretations.

But this is only the tip of the iceberg. Is "implementation" a concept *sui generis*? Or is it just another name for something more familiar, such as "instance", "exemplification", "reduction", "supervenience", etc.? Let us consider some of these.

7.5 POSSIBLE INTERPRETATIONS OF "IMPLEMENTATION".

The problem in trying to determine whether the notion of "implementation" should be assimilated to some other, perhaps more familiar, notion is that there is very little agreement over the proper characterization of those other, candidate notions, or even over terminology. For instance, it seems clear that an implementation of an Abstraction is *not* an "instance" or "instantiation" of the Abstraction, because two Abstractions (e.g., two abstract data types) can implement *each other*. As we saw, the abstract data type Record can be used to implement the abstract data type List. Moreover, the abstract data type List can be used to implement the abstract data type Record. And, though there is probably no good reason to do so, one could, perversely, implement lists by records that are themselves implemented by lists. And so on. Yet "instantiation" is normally thought of as an asymmetric relation. In spite of this, we find a recognized authority on implementations, Guttag et al., saying that an implementation *is* an "instance" (Guttag et al. 1978: 62). In this chapter, I do not expect to be able to resolve the issues, only to set them out.

7.5.1 Implementation as Individuation.

Let's begin (following Castañeda 1975b) by distinguishing between "individuation" and "differentiation". Recall that early ancestor of semantic networks, Porphyry's Tree, in which a universal, such as a genus, is analyzed into sub-genera or species by means of a "specific difference" or *differentia*. Thus, for example, the *differentia* Rational applied to the genus Animal yields the species Human (= Rational Animal); all other, non-human, animals are *not* Rational. Thus, Humans are *differentiated* from non-Humans. As a category, Human is "lower" than Animal; it is more "specific"—it has an extra defining property, namely, being rational. Human is itself a universal—as it happens, an *infimum species*, i.e., a category that is not analyzed into subcategories but into concrete *individuals*, e.g., Plato, Sappho, you, me.

What is the analogue of a *differentia* that, when applied to an *infimum species* yields an individual? John Duns Scotus called it 'haecceity', or "thisness". "Instantiation" is the relation between any level of Porphyry's Tree and the level *below* it; "differentiation" is a relation between subcategories (or members) of a single category. Thus, just as Human is differentiated from non-Rational Animal, so Plato is differentiated from Sappho and you from me. And just as Human is instantiated from (or, is an instance of) Animal, so Plato, Sappho, you, and I are instantiated from (or, are instances of) Human. And Plato et al., unlike Human et al., are "individuals": "Individuation" is the relation between an *infimum species* and its individuals. As Gracia puts it, "I regard something as an individual if and only if it is a noninstantiable instance of an instantiable, while I regard universals as capable of instantiation, that is, as instantiables" (Gracia 1990: 503).

Thus, perhaps implementations are individuals, and Abstractions are universals. That does seem to hold for *concrete* implementations. But it fails to hold for *abstract* implementations, and it only works when there is a hierarchy or linear ordering of successively more detailed Abstractions. It fails to account for the relation that obtains when a list implements a stack.

On the other hand, since individuals and lower-level instantiables *can* be viewed as *implementations* of higher-level instantiables or universals, I suggest that instances and individuals are implementations, but not conversely.

7.5.2 Implementation as Instantiation.

Michael V. Anthony (1991) has explicitly argued that computer "implementations" are *not* "instantiations". The background of Anthony's argument is the "Classical" vs. "Connectionist" controversy, in particular, the question whether "a Connectionist architecture *instantiates* the Classical framework" or whether "a Connectionist architecture *implements* a Classical architecture" (p. 325, my italics), or whether there is some other (or no) relation between them.

As Anthony uses the term,

> 'Instantiation' expresses a simple relation between individuals and properties: an individual i instantiates a property P if and only if Pi. ... In the case of instantiation ... a *single* model or architecture is involved, and what is in question are its properties. (p. 325.)

Note that this is *not* necessarily the relation of instantiation we just looked at. There, instantiation was a relation between an individual or a category and its immediate superordinate category. Here, it is a relation between an individual and a *property*. Others have called the latter relation 'exemplification', though, of course, the relation between an individual and its properties can be (and has historically been) explicated in terms of category membership, and "exemplification" is the term often used for the relation between a real object and the Platonic Form that it "participates" in. Since I am not going to try to resolve several thousand years of metaphysics here, let's stick with Anthony's definition for now.

So, for a connectionist architecture to instantiate a classical framework would be for it to have classical properties. Let P_{cl} be the set of properties that "define the Classical ... framework" (p. 325). Let C_x be "a particular Connectionist architecture". Suppose that C_x has all of P_{cl}. Couldn't we then identify the Abstraction ClassicalSystem as the set of properties P_{cl} and treat C_x as an implementation of it? Anthony observes in a footnote (p. 339n7) that "individuals" could be "*abstract* objects like functional architectures" (my italics); thus, C_x is *also* an Abstraction (thus, clearly, Anthony is not speaking the language of §7.5.1).

In contrast,

> Where *implementation* is at issue ..., *two* functional architectures must be considered. A functional architecture FA1 is implemented, if at all, by the execution of a program in a distinct functional architecture FA2. (p. 325.)

So, FA2 might *itself* not have FA1's properties (so FA2 need not be an *instance* of FA1), but the *process*—the program in execution—*might* have FA1's properties (and so be an instance of FA1). In general, this seems OK. For instance, a machine-language program might have FA1's properties, but the machine language *itself* might not. As a trivial example, a machine-language program can have records, while the machine language itself doesn't have them.

"Intuitively," Anthony tells us, "the primitive operations, representational structures, etc. of FA1 get 'made up' or 'constructed' out of the resources of FA2. ... This is the relation that typically exists, for example, between assembly language functional architectures ... and higher-level architectures like LISP or Pascal ... when the latter are up and running on a computer"

(p. 325). So the idea is this: If FA2 (e.g., the machine language) has records as a primitive data type, then it's easy to implement FA1 (e.g, Pascal) in it, because they both already share the same properties—they both instantiate "having the record data type". If FA2 *lacks* records, they can nonetheless be implemented in it. But wouldn't FA2 then *have* records? Anthony seems to be trying to distinguish between essential properties and accidental ones: Records are an "essential" feature of a programming language that has them among its primitive (or built-in) data types, an "accidental" feature of a programming language that can only define them in terms of (or construct—or implement!—them out of) its primitive ones. (The latter are akin, too, to the notion of *derived* rules of inference in a natural deduction system.)

Consider, by way of analogy, the rationals and the integers. As is well known, the rationals can be ... implemented in? constructed out of? ... (equivalence classes of ordered pairs of) integers.[4] We'll come back to this implementation or construction or reduction in §7.5.3. For now, suppose we have a *language* for talking about the rationals (and sets). It will have, as one of its terms, a representation for $\frac{1}{2}$ (i.e., the rationals have, as one of its data objects, $\frac{1}{2}$). The language will also have as a primitive predicate some expression for a property P that applies only to rationals (i.e., the rationals have, as one of their properties, P)—e.g., (some) rationals have the property of being less than 1, and the set of rationals itself has the property of being dense. Now, suppose we have a language for talking about the integers (and sets). Can we talk about the *rationals* in the language of integers? Yes—by finding or constructing (analyzing?) the rationals' data objects (like $\frac{1}{2}$) in the integers and by finding or constructing (or analyzing) the rationals' properties among (or from) the integers' properties. By defining new *terms* in the language of integers, that language would now have terms for $\frac{1}{2}$ and P. That is, we could now say more in the language of integers than we thought we could; it *wasn't*, after all, limited to talking about integers. What, then, would be the difference between the language of rationals and the (extended) language of integers? The former would have certain terms and predicates (NPs and VPs) that the latter would lack; but they could be *defined* in the language of integers.

That this really is close to what Anthony has in mind can be seen from the following passage:

> ...in cases of *implementation*, lower-level architectures typically do not *instantiate* the *characteristic* properties of higher-level ones. An assembly-level architecture *implementing* LISP, for instance, does not also *instantiate* LISP: it lacks the *necessary* primitive properties (e.g., CAR, CDR), and has primitive operations LISP lacks (e.g., various operations on the contents of the accumulator). (p. 326, my italics.)

Here, 'characteristic' and 'necessary' can be taken to mean "essential". But doesn't a machine-language implementation of Lisp *have* `car`?[5] Maybe not: It can "simulate" `car`—or implement it?—but it doesn't *have* it; it can do what `car` does without *having* `car`. If you'll excuse the pun, I can do what can be done with a car without having one—by walking, taking the bus, etc.

We can draw a distinction between "weak" and "strong" implementations. For instance, a strong implementation of Lisp in machine language would be such that the machine language actually *had* identifiable data structures and procedures corresponding to lists, `car`, etc. A weak

[4] And vice versa, don't forget!—since $\mathbb{Z} \subset \mathbb{Q}$.

[5] `car` (or `first`) is the Lisp function that takes a list as input and returns its first member; `cdr` (or `rest`) is the Lisp function that takes a list as input and returns the "rest of" that list, i.e., the list consisting of all but that first member.

implementation of Lisp in machine language would be such that it would do the same things (e.g., be able to return the first element of a list) without having lists or `car` (just as I can get from my home to a store by car or by walking).

Conversely, "it is also true that an instantiation of LISP need not implement any distinct, higher-level LISP architecture" (p. 326). For example, I suppose, Franz LISP (understood as an *instantiation* (rather than an *implementation*?) of Lisp) need not *implement* SNePS. So, instances and implementations (as Anthony defines them) "are mutually independent" (p. 326).

Let's return to the rationals and the integers. Consider the integers first (and, for convenience, consider only the non-negative integers). What are they? One way to answer this is to cite Peano's axioms. That would be to present the Abstraction Integers—in fact, it is an abstract data type. Another way to answer the question is to say that integers are any things that satisfy Peano's axioms. So, e.g., the sequence consisting of $\emptyset$, $\{\emptyset\}, \{\{\emptyset\}\}$, etc., are integers. So is the sequence $\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}$, etc. So is the sequence $\emptyset, \{x \mid x \text{ is a set } \& \textit{ Cardinality}(x) = 1\}$, $\{x \mid x \text{ is a set } \& \textit{ Cardinality}(x) = 2\}$, etc. So is the sequence of symbol types 0, 1, 2, ..., 10, 11, 12, ..., 99, 100, etc. So is the sequence of symbol types 0, 1, 10, 11, 100, etc. (And if you ignore 0, so is I, II, III, ..., X, etc.). And so on. Note, for future reference, that the *differences* between each of these are what might be called "implementation details".

The vague feeling of discontent that this leaves us with is that there are *too many* integers, so to speak; isn't there an "intended interpretation"? What model-theoretic semantics teaches us is that there isn't. For any set of axioms, there are infinitely many models, including non-isomorphic ones. So, the only way to talk about "the" integers is to restrict ourselves to talk about Peano's axioms. (Recall the quotation from Russell, §3.2.1. Mathematics, on his view, is pure syntax.) The alternative is to choose, arbitrarily, some model of them and talk about *it*.

Now, since such axioms are abstract data types, such models are *implementations* in various media. For our first three examples above, the implementing media are sets "put together" in different ways; for the others, the implementing media are certain symbol types. If we restricted ourselves to some finite initial sub-sequence of the non-negative integers, we could take arbitrary physical objects as our implementing medium.

Are any of these implementations "instances" of ... integers? Peano's axioms? 'Instance' in the Porphyrian-tree sense? In Anthony's sense? I'm inclined to say 'No', but that's primarily because I find the interpretation of 'implementation' in terms of semantic models to be more illuminating. Moreover, I suspect that if one wanted to force the concept of an implementation into the mold of "instantiations", one could do so only by seeing "instantiations" as a kind of semantic modeling.

7.5.3 Implementation as Reduction.

Given all this, let's now consider the rationals (in particular, the non-negative rationals). Consider a set of axioms for the rationals; i.e., consider the abstract data type (NonNegative)Rationals. What are some of its implementations? Well, there are the fractions, i.e., the symbol types $\frac{0}{4}, \frac{1}{2}, \frac{2}{3}, \frac{1}{7}$, etc. There are the repeating decimals, i.e., the symbol types $0.\overline{0}$, $0.5\overline{0}$, $0.\overline{6}$, $0.\overline{142857}$, etc. There are also certain *constructions* from the integers, e.g., certain ordered pairs of integers: $\langle 0, 4\rangle$, $\langle 1, 2\rangle$, $\langle 2, 3\rangle$ $\langle 1, 7\rangle$, etc. Each of these can be considered to be an *implementation* of the rationals. Rational numbers are anything that satisfy the axioms.

Here, the notion of implementation details plays a larger role, since we seem to have "too many" rationals. In a Morning Star/Evening Star sense, $\frac{2}{3}$ and $\frac{4}{6}$ are the "same" rational number, as are the ordered pairs $\langle 2,\ 3\rangle$ and $\langle 4,\ 6\rangle$. We could say that that's an implementation detail, and provide rules (further axioms?) to indicate when two "intensionally distinct" rationals are "extensionally equivalent". We have such rules for, say, addition of integers: Does '$2 + 2 = 4$' state a fact about addition, or does it assert an extensional equivalence between intensionally distinct integers?. Or else we could—as in fact we normally do—implement the rationals as *equivalence classes* of ordered pairs of integers.

Now, often this "implementation" of rationals by integers (plus set theory), is called a "reduction" of the rationals to the integers. "All we really need," so the reductionist says, "are the integers (and set theory); we can define the rationals in terms of them (or, we can reduce the rationals to them)." So: Is implementation just reduction? Are all reductions implementations?

Again, we have related, but, I think, distinct, concepts. As Smith puts it, "*Reducibility* ... is a relation between *theories*; one theory is reducible to another if, very roughly, its predicates and claims can be translated into those of another" (Smith 1991: 280n39). Now, in the case of the rationals and the integers, I would really hesitate to say that the former have been "reduced" to the latter. I would be willing to say that the *theory* of rationals can be reduced to the *theory* of integers-plus-sets. But even here, when we prove some theorem about rationals, we haven't proved a theorem about integers but, at best, about certain *sets* whose "ground elements" are integers. For example, to prove a theorem about the *rational* number $\frac{1}{2}$ would be to prove a theorem about the following arcane set of sets of integers and sets of integers: $\{\{a, \{a, b\}\} \mid a, b \in \mathbb{Z}^{+} \ \& \ 2a = b\}$.[6] Suppose integers are implemented as sets, and multiplication is implemented as a set of ordered pairs of factors. Then we might have the following situation: If $\{\{\}\}$ and $\{\{\}, \{\{\}\}\}$ implement 1 and 2, respectively, then $\frac{1}{2}$ could be implemented as the monstrosity $\{\{a, \{a, b\}\} \mid a, b \in \mathbb{Z}^{+} \ \& \ \{\{\{\}, \{\{\}\}\}, \{\{\{\}, \{\{\}\}\}, a\}\} = b\}$. The mind boggles. This is supposed to be *easier* to understand than '$\frac{1}{2}$' or '$0.5\overline{0}$'?

And the only reason we're interested in those rather arcane sets "of" integers is because they implement—are models of—the abstract data type Rationals. We might feel more "comfortable" with these arcane sets insofar as we are more comfortable with good old-fashioned sets and integers rather than with rationals *per se*. But that is an epistemological consideration that is rather suspect in the long run.

Once we have implemented the rationals using integers and sets, we also have another implementation of the integers, of course (since the integers are a proper subset of the rationals—or perhaps it would be better to say that a certain proper subet of the rationals is an implementation of the Integer abstract data type). I have in mind here the sequence $\frac{0}{1}, \frac{1}{1}, \frac{2}{1}, \frac{3}{1}$, etc. As a matter of fact, there are several implementations of the integers to be found among the rationals; here are a few:

$$\frac{0}{2}, \frac{1}{2}, \frac{2}{2}, \frac{3}{2}, \ldots$$
$$\frac{0}{3}, \frac{1}{3}, \frac{2}{3}, \frac{3}{3}, \ldots$$
$$\frac{0}{1}, \frac{1}{1}, \frac{1}{2}, \frac{1}{3}, \ldots$$

[6]The ordered pair $\langle 1,\ 2\rangle$ "is" (or can be implemented as!) $\{1, \{1, 2\}\}$. The equivalence class containing $\langle 1,\ 2\rangle$ "is" $\{\langle a, b\rangle \mid a, b \in \mathbb{Z}^{+} \ \& \ 2a = b\}$. So, the rational $\frac{1}{2}$ "is" $\{\{a, \{a, b\}\} \mid a, b \in \mathbb{Z}^{+} \ \& \ 2a = b\}$.

These (especially the last one) may seem a bit odd, but recall that Peano's axioms only require that there be a successor relation, not that that relation be (implemented as) +1 (or even as +Successor(0)); the choice of a 0-element is arbitrary, too. Other arbitrarily strange ones are possible, e.g.,

$$\frac{0}{1}, \frac{1}{2}, \frac{2}{3}, \frac{3}{4}, \ldots$$

And, of course, using an order imposed by a diagonalization, the (non-negative) rationals themselves can be taken as an implementation of the integers; e.g., if we arrange them (ignoring equivalences) two-dimensionally as follows:

$$\frac{0}{1}, \frac{0}{2}, \frac{0}{3}, \frac{0}{4}, \ldots$$

$$\frac{1}{1}, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \ldots$$

$$\frac{2}{1}, \frac{2}{2}, \frac{2}{3}, \frac{2}{4}, \ldots$$

then the sequence $\frac{0}{1}, \frac{0}{2}, \frac{1}{1}, \frac{0}{3}, \frac{1}{2}, \frac{2}{1}, \frac{0}{4}, \frac{1}{3}, \frac{2}{2}, \ldots$ implements the integers, too.

Finally, we could, if we wanted to, *re-implement* the rationals in one of these implementations of the integers, by the usual ordered-pair construction.

Why bother? Well, besides whatever insights such playful model-making gives us into the logical structure of the integers, it also shows that *reduction* (or construction) for the purposes of providing stronger epistemological foundations is *not* what implementation is. All of the above are implementations; none serves any interesting or useful reductive purposes.

The upshot is that although some, or even all, reductions or constructions might be implementations, certainly not all implementations are reductions.

7.5.4 Implementation as Supervenience.

There is one major concept that has a fairly precise definition and that is a good candidate for interpreting implementation, namely, supervenience. Recall, first, where we stand with respect to implementation, semantics, and syntax: An implementation of an Abstraction in some medium is a semantic model of the Abstraction in the "medium" of some semantic domain. And a semantic model is *any* structure—including the Abstraction itself!—that can be correlated (or put into correspondence) with the Abstraction. The closer the correlation, the better the semantic interpretation, even if, in the base case of a *self*-interpretation, we must resort to syntactic understanding.

Supervenience is certainly a plausible candidate. As Smith notes, "the term *supervenience* is used to relate phenomena themselves; thus the strength of a beam would be said to supervene on the chemical bonds in the constitutive wood. ... [S]upervenience doesn't necessarily imply reducibility" (Smith 1991: 280n39). When one domain "supervenes" on another, is it *implemented* by that other domain? And when one domain is *implemented* by another, does it supervene on that other domain? What, then, is supervenience? To answer this, let's look at several of Jaegwon Kim's classic papers.

7.5.4.1 Supervenience: An introduction.

In "Supervenience and Nomological Incommensurables" (1978), Kim gives a precise formulation of the informal notion that "one *family of properties* is 'supervenient' upon another *family of properties* in the sense that two things alike with respect to the second must be alike with respect to the first" (p. 149, my italics). The ultimate goal is to see whether it makes sense to say that mental properties are supervenient on physical ones. Consider, for the moment (we'll refine this later on), the rough analogy of mental properties to software and physical properties to hardware processes. If we have two like processes, do we have two like mental properties? This depends on how much "alike" the two processes must be: (a) They might be identical (same hardware, same machine-language level, etc.), or (b) they might be two different implementations with the same algorithmic behavior.

Consider (a): Could two identical processes be the result of distinct programs? Yes, *if* it's possible to have two distinct high-level–language programs that compile into the same machine-language program. Any other possibility (e.g., two distinct machine-language programs) would have processing differences. For instance, consider two (high-level–language) programs that differ only in that one of them has a bunch of "no-ops":

Program 1:

```
begin
  x := 2;
  y := 3;
  z := x + y
end.
```

Program 2:

```
begin
  x := 2;
  for i := 1 .. 100 do begin end;
  y:= 3;
  z := x + y
end.
```

Program 2 will run more slowly. With a suitable optimizing compiler, however, both could compile into the *same* machine-language program. So, in this case, the "mental" would *not* supervene on the "physical", *except* in the sense that the two high-level programs have the same input–output behavior and the same algorithm (except for eliminable or non-essential differences).

Consider (b), different implementations with the same algorithmic behavior: For instance, consider, first, Cassie implemented on a Sun workstation running SNePS in Allegro Common Lisp using the representations of Shapiro & Rapaport 1987 vs. Cassie implemented on a TI Explorer running SNePS in TI Lisp using the (different) representations of Wyatt 1989, 1990, 1993. Assume, further, that their input–output behavior is identical. But *do* they have the same *algorithmic* behavior? Arguably not, because of the representations (this, at any rate, is an empirical question;

cf. §5.5). So consider instead two Cassies as before, but both using the Shapiro & Rapaport 1987 representations. In this case, any differences would be *implementation* side-effects, *not* "mental" differences.

The upshot of all this? So far, given several plausible assumptions and the informal presentation of supervenience, the mental does seem to be supervenient on—as well as implemented by—the physical.

The other informal aspect of supervenience is that "there is no relationship of definability or entailment between the two families" of properties (pp. 149–150). Now, implementation is neither definability nor entailment, so it could indeed be supervenience. And, as Donald Davidson points out, "Dependence or supervenience of this kind [viz., of the mental on the physical] does not entail reducibility through law or definition" (Davidson 1970: 88; cited in Kim 1978: 150). As we saw in §7.5.3, not all implementations are reductions, so implementation—like supervenience—does not entail (hence, is not) reducibility; a program isn't "reduced" to a process, nor is a program defined in terms of the process (except maybe in cases of "reverse engineering", where we seek to discover in a bottom-up fashion *what* a computer is doing by examining the details of its behavior).

Now, according to Kim, "the main point of the talk of supervenience is to have a relationship of dependence or determination between two families of properties *without* property-to-property connections [or "correlations"] between the families" (p. 150). But in the case of implementation there *are* such property-to-property correlations. So maybe implementation *isn't* supervenience? As we will see, however, Kim's explication of supervenience allows for such correlations.

Note, by the way, that it's not a question of whether mental properties "emerge" from physical properties in some mystical way. Given a program and an implementing process, *there are* correlations between them: Suppose we have a process whose behavior can be described (from the intentional stance?) in mental language. Here, we would be giving a *functional* characterization of the process—what it does and how it does it, "modulo" its physical description (cf. Goguen et al.'s quotient spaces). Suppose, then, that we are able to do this sort of reverse engineering for some physical process. We would then have the mental and the physical, and could have the correlations.

However, Kim distinguishes between supervenience and functionalism/implementationalism (p. 150, col. 2). His point is that he's more interested in the fact "that there seem to be mental states which are 'nomologically incommensurable' with respect to neurophysiological or, more generally, physical properties; there appear to be mental states which do not nomologically correlate with physical states" (p. 150). The *reason* for this nomological incommensurability is not of interest to him. So, let's move on to other issues.

The fundamental intuition is that "what happens at the psychological level is fixed in every detail once the neurophysiological events are fixed. If you recreate Mr. Jones molecule by molecule, atom by atom, then the replica will have the same mental life that Mr. Jones has" (p. 151).[7] Note also that this is a strictly internalist, methodologically solipsistic approach—there's no talk here of external relations.

Kim first defines two set-operations, $^{\#}$ and * (p. 152, col. 1):

cite defs of $\mathbf{M}^{\#}$ and $\mathbf{M}^{*}$

[7]Kim compares this to *Star Trek*. On this, see Blish 1970, esp. Ch. 1. And on molecule-by-molecule "re-creation", cf. Castañeda 1989b and § **[on $\mathbf{C}^*$], [???]** below.

Consider an example. Let P, Q **[roman font ???]** be properties, and let $M = \{\mathrm{P}, \mathrm{Q}\}$. Then $M^{\#} = \{\mathrm{P}, \mathrm{Q}, \mathrm{P} \wedge \mathrm{Q}, \mathrm{P} \vee \mathrm{Q}, \mathrm{P} \rightarrow \mathrm{Q}, \neg\mathrm{P}, \neg\mathrm{Q}, \neg(\mathrm{P} \wedge \mathrm{Q}), \neg(\mathrm{P} \vee \mathrm{Q}), \dots\}$, and $M^{*} = \{\mathrm{P} \wedge \mathrm{Q} \wedge (\mathrm{P} \vee \mathrm{Q}) \wedge \dots,$ $(\mathrm{P} \wedge\neg\mathrm{Q}) \wedge (\mathrm{P} \vee\neg\mathrm{Q}) \wedge \dots, \dots\}$, where each element of M^{*} is an M-maximal property (and—in our example—the first-listed element of M^{*} contains no occurrences of $\neg$P or $\neg$Q and the second-listed contains no occurrences of $\neg$P).

Next, let D be a domain of objects, and let M, N be sets of properties that elements of D can have. Then *M is supervenient on N with respect to D* $=_{df}$ $\Box$(objects in D that share all properties in $N^{\#}$ also share all properties in $M^{\#}$) (p. 152, col. 1). That is, Suppose M supervenes on N with respect to D, and let $d, d' \in D$. Then $\Box(d, d'$ share all properties in $N^{\#} \rightarrow d, d'$ share all properties in $M^{\#})$.[8] What's meant is not that d, d' *have* all properties in $N^{\#}$, but that *if* they have all and only the *same* properties in $N^{\#}$, then they also have the same properties in $M^{\#}$. So, where D, d, d', N, M are as before, *M supervenes on N with respect to D* $=_{df}$ $\Box((\forall\, \mathrm{P}_N \in N^{\#})[\mathrm{P}_N(d) \leftrightarrow \mathrm{P}_N(d')] \rightarrow (\forall\, \mathrm{P}_M \in M^{\#})[\mathrm{P}_M(d) \leftrightarrow \mathrm{P}_M(d')])$.

Now suppose that M is a set of mental properties and N is a set of physical properties. What might D be? What kind of thing has both mental and physical properties? Descartes would probably say that *nothing* has *both* sorts of properties—that a *res cogitans* is not a *res extensa*. But perhaps a "person" or a "self" or even an AI computer (process) has both? Consider a computer programmed to compute greatest common divisors. We can equally well say, giving a sort of "mental" description, that it computes greatest common divisors and, giving a "physical" description, that it has a register that stores certain numbers (etc.). So, what the supervenience of the mental on the physical says is that if, say, two computers or two persons have the same physical properties, then they have the same mental properties, *where there is not necessarily any relevant or interesting relation between M and N*—it's just a correlation, but not (necessarily) a semantic one, since it's not (necessarily) point-by-point—there need be no patterns to match.

What, then, of implementation? Kim presents an argument that reducibility and definability *do* (*pace* Davidson?) entail supervenience (p. 152, col. 1). Can we run a similar argument to show that if M is *implemented* by N, then M supervenes on N? The argument requires biconditionals between N and M. Surely, if N implements M, there are such biconditionals. They would be provided for by the semantic interpretation function between N and M. Suppose that two things diverge on some M-property. Then they'll diverge in $N^{\#}$. So, if there are such biconditionals, then implementation does entail supervenience. Are there really such biconditionals? Since N implements M, there could be implementation side-effects (the domain of semantic interpretation might be "bigger" than the image of M in it). Still, if things diverge on M, they'll diverge in $N^{\#}$ (though perhaps not conversely; cf. the discussion of qualia in §7.6.3).

Kim argues, *pace* Davidson, that supervenience on a *finite* N entails that "each property in M which is instantiated is biconditional-correlated with some property in $N^{\#}$" and that such generalizations are lawlike (p. 152, col. 2). This is surely true for implementation in the N-to-M direction (p. 152, col. 1). Is it true in the M-to-N direction (p. 152, col. 2)? Suppose that $Q_1, \dots, Q_n$ are the physical properties of the implementation, that P is an M-property, and that $Q_1 \vee \dots \vee Q_n \rightarrow P$. Suppose, by way of contradiction, that x (e.g., a computer process) has P (e.g., a certain input–output behavior) but that x lacks each Q_i (i.e., is implemented differently). However, x is implemented *somehow*; let K be a property that x has in virtue of its implementation. Suppose y (some other process) also has K. Now, since M supervenes on N (N implements M),

[8] That, at least, is what Kim *says*; but doesn't he mean M^{*} and N^{*}? Perhaps not; cf. p. 153, col. 1.

y has P (i.e., y has x's input–output behavior). So, K must be one of the Q_is. Kim's argument seems to carry over (although details of the relationships between M, N and P, Q are not clear).

Moreover, supervenience *is* a semantic relation:

> To summarize: (1) if M supervenes on N, there are property-to-property correlations between M and N; (2) every property in M has either a necessary or sufficient condition in N ...; (3) if N is finite, every property in M is biconditional-connected with some property in N. ... [F]inite-based supervenience ... guarantees for each property in the supervenient family a co-extension in the supervenience base; and depending on the modality that attaches to the correlations between the two sets of properties, this may yield reducibility and definability. (Kim 1978: 153–154.)

Viewing the supervenient set as the mental realm and the supervenience base as the physical realm, each mental property has a co-extensive physical property and might be reducible to it, or definable in terms of it. The co-extensiveness *almost* works for implementation, but, strictly speaking, it doesn't. For the implementing device is not a set of properties; hence, it has no extension. Rather, it *is* the extension of the mental (or Abstract) properties. It is an open question what the appropriate "modality" is for sets of mental properties and sets of physical properties.

Kim asks, "What is the basis of our belief that, say, metaphysical processes wholly determine all other processes?" (p. 154). We might ask, more generally, about the basis of our belief that supervenience-base properties determine supervenient properties and, more specifically, that physical properties (of the implementation) determine mental properties (Abstract properties). But this latter question doesn't sound right for implementations. It sounds right for some kinds of epiphenomenalism, or for an identity theory, or for a theory that mind is an emergent property of the brain, etc., but not for implementations. One reason is that we don't think of an implementation as *determining* an Abstraction. Rather, the Abstraction is epistemologically prior; this is one of the early functionalist methodological principles (cf. Chomsky 1968: 12, Fodor 1968). One can, of course, "infer" an Abstraction ("abstract" it) from an implementation; but that is (merely) reverse engineering.

Finally, however, there is a problem in assimilating implementation to supervenience: Implementation isn't a relation between sets of properties. It's a relation between "physical" things and Abstractions—a relation between two different *kinds* of things—whereas supervenience is a relation between sets of properties.

7.5.4.2 Kinds of causation.

What about causation? I have argued elsewhere that, *pace* Searle, there are a variety of causal (or quasi-causal) relationships wherever there is an Abstraction, an implementing medium, and an implementation of the former in the latter (Rapaport 1985b, 1988a). To summarize briefly: Searle (1984) distinguishes only between what I call the Abstraction and the implementing medium, saying that the latter "causes" and "realizes" the former and that there are causal relationships within the Abstraction and within the implementing medium. As I see it, however, between the Abstraction and the implementing medium is the implementation itself (just as the computer process is between the program and the computer). Recalling the analogy of plays, Hamlet is an Abstraction, Olivier (say) is the implementing medium, and Olivier-playing-Hamlet is the implementation of the former by the latter. This yields a family of causal, quasi-causal, and implementation relationships:

1. The relation between the Abstraction and the implementation. (In Rapaport 1988a, I said that an Abstraction was a "species" of an Implementation, and an Implementation was an "instance" of an Abstraction. In view of my remarks in §7.5.2, above, this perhaps should be taken with a grain of salt.)

2. The relation between the Abstraction and the implementing medium (called 'realization' in Rapaport 1988a, following Searle's terminology, though not his relata). Again, in view of my remarks thus far in this chapter, I am no longer sure that this terminology is accurate.

3. A relation of ordinary, physical causation within the implementing medium (assuming, for now, that the implementing medium is a physical one).

These are the basic relationships. Definable in terms of them, we have:

4. The relation between an implementation and the implementing medium. (The medium "realizes" a "species" of the implementation; conversely, an implementation is an "instance" of that which is "realized" in the medium.)

5. A quasi-causal relationship among events and objects *within* the implementation: For example, Olivier-playing-Hamlet "causes" Jean Simmons-playing-Ophelia to do certain things: Olivier-playing-Hamlet is an "instance" of Hamlet, which is "realized" in Olivier, who, in turn really causes real events in Simmons, who "realizes" a "species" of Ophelia.

6. A quasi-causal relationship among events and objects *within* the Abstraction: For example, Hamlet causes Ophelia to do certain things because Hamlet is "realized" in Olivier, who really causes real events in Simmons, who "realizes" Ophelia.

Whatever we choose to call them, however, these relationships are all there.

In "Causality, Identity, and Supervenience in the Mind–Body Problem" (1979), Kim considers the same sort of issue from the point of view of supervenience. Consider psychophysical causation: how mental events cause brain events, and vice versa. Compare this to what might be called "algorithmic–physical" causation: how programs cause computers to behave.[9] Well, how do they? When the program is compiled, certain "switches" are set—i.e., the contents of certain registers are set to certain strings of 0s and 1s. This is done according to machine-language specifications, which are translations from the higher-level programming-language specifications. When the program is executed—when the *process* comes into being—the switch-settings permit the flow of energy to reset some of the switches, continually, until the program halts (i.e., until the computer's switches reach a state in which none of them permit any more energy-flow to reset any of them). Similarly, mind→brain "causation" ought to consist of physical changes to brain states—the analog of switch settings. How would brain→mind "causation" work? Well, how does computer→program causation work? By switch settings that are interpreted as program, not data. An output program in such a situation would just be a symbolic record of the switch settings in

[9]And vice versa? It can happen when programs modify themselves.

the program-region of memory. So, in mind–brain interaction, perhaps the mind just *is* "switch settings" in the brain.

Kim sets the scene as follows:

> Let M be a mental event (type), and let P be its neural correlate. ... Assume further that M is a cause of a physical event P^* (this is the posited psychophysical causal relation), and ... that there is a law linking M with P^*. It follows that ... a law exists that links P and P^* .... Now we can see three related puzzles arise. (Kim 1979: 35.)

Let M be a piece of program code, let P be its corresponding switch settings, and assume that M causes some other switch settings P^* in a lawlike way. Then P and P^* are lawfully linked. But is it really M that causes P^*? It seems better to say that it is the *implementation* of M that causes P^*. P^* can be described in programming-language terms; call this description 'M^*'. What is the relation between M and M^*? I would say that it is the quasi-causal relationship 6, above.

The puzzle that is of most interest for our purposes among the three that Kim sees in this situation is what he calls

> *The problem of pre-emption.* Given that M and P are nomic equivalents and given that there is a law linking P with P^*, as well as one linking M with P^*, P appears to have at least as strong a claim as M to be the cause of P^*. (Kim 1979: 35.)

That is, P "pre-empts" M as cause of P^*. Kim suggests that psychophysical "causation" (the relation between M and P^*) is not "real" causation (*pace* Searle!) but a "simultaneous equivalence" (p. 35; cf. his earlier notion of "Cambridge dependence" (Kim 1974)). I would prefer to call it ... implementation. Note that even if it is not the case that M *causes* P, still M is crucial for understanding what P is and what its *function* is (what it does). So we needn't despair that psychophysical "causation" isn't what we might have thought it was, that—as Kim puts it, citing Norman Malcolm—"common-sense psychological explanation of bodily motions in terms of beliefs, desires, intentions", etc., would be pre-empted (p. 36). What I am taking to be implementation is what underwrites our ability to take the Dennettian intentional stance, what underwrites Chomskian/Fodorian functionalist methodology, and what makes reverse engineering difficult (knowing the switch settings without knowing the program doesn't easily tell us what a computer is doing).

Kim generalizes the problem to properties and events, so we could talk—instead of "programs" and "computers"—of *properties* of programs and *physical properties* of computers, or of *computational processes* (which are events) and *physical processes*. Do properties of programs (e.g., the property that a certain variable is declared as being of a certain type) cause physical properties of computers (e.g., that certain switches have certain settings)? Well, the program properties certainly *explain* the physical properties. Does it, for that matter, make sense to say that one *property* causes another? Surely, were the program properties different, the physical properties would be different (different programs produce different physical behaviors.) On the other hand, the computational *process* seems to be only a *different description* of the physical process, along the lines of the intentional stance.

So, where does supervenience come in?

> ... if two organisms are metaphysically indistinguishable from each other, then they will share the same psychological life; and if two physical objects are metaphysically indistinguishable from each other, then they will share the same macro-properties. (Kim 1979: 40.)

Now, Kim also says (p. 41) that the macro-properties are *supervenient* on the micro-properties *even if the macro ones could have been "realized"* [p. 40] *in different ways*. This suggests that implementation is *not* supervenience. But the quotation suggests that two implementations will be implementations of the *same* Abstraction, which *seems* inconsistent with, for instance, the cases of a single computer program that is playing chess and re-enacting a Civil War battle, or a natural-language text that is both about chemical and mathematical lattices.

One way out of this might be to distinguish between the Abstraction and an interpretation, saying that in such cases there is only one Abstraction but several interpretations. However, it is the interpretation that I have hitherto considered to be the Abstraction.

Another way is to use Smith's distinction between the actual world, a mathematical model of it, and a program: The program implements the mathematical model (the Abstraction), but the mathematical model is only partial and *could*, therefore, be a model of lots of different aspects or parts of the actual world. Also, surely, lots of different parts of the actual world *have the same structure*—that is, the same mathematical model.

Now, Kim offers a slightly different definition of supervenience:

> A family M of properties is *supervenient upon* a family N of properties with respect to a domain D *just in case* necessarily, for every property P in M and each object x in D such that x has P, there is a property Q in N such that x has Q and any object y in D which has Q also has P. (Kim 1979: 42.)

So, taking D as the domain of, say, intelligent entities (cognitive agents), which can have both mental (M) and physical (N) properties, for each mental property $P_M \in M$ that cognitive agent $x \in D$ has, there will be a physical property $Q_N \in N$ that x has, and any other cognitive agent $y \in D$ that has Q_N also has P_M. This last clause, about y, suggests a third way out: The chess-playing computer *is* a Civil War computer; only the *interpretation* of its input and output differs. So the interpretation is not a mental property.

However, the chess properties are *also* supervenient on N. So what happens when two distinct Ms supervene on N? What would be the relationship between the two Ms? (See Figure 7.3.) Could it be that M_1 supervenes on M_2? Choose $x \in D$. Choose some property in M_1, $P_{M_{1_i}}$, such that $P_{M_{1_i}}(x)$. Then, since M_1 supervenes on N, there is $Q_{i_1} \in N$ such that $Q_{i_1}(x)$ and which is such that, for any $y \in D$ such that $Q_{i_1}(y)$, $P_{M_{1_i}}(y)$. Similarly, since M_2 supervenes on N, there is $Q_{i_2} \in N$ such that $Q_{i_2}(x)$, and that is such that, for any $y \in D$ such that $Q_{i_2}(y)$, $P_{M_{2_i}}(y)$. Now, in the case at hand, we can assume that $Q_{i_1} = Q_{i_2}$. The idea is that, say, the object Soldier-1 in the Civil War program is "implemented" by the same data structure that implements the object Pawn-1 in the chess program. To see whether M_1 supervenes on M_2, we need to see whether there is $P \in M_2$ such that $P(x)$ and which is such that, for all $y \in D$ such that $P(y)$, $P_{M_{1_i}}(y)$. The obvious candidate for P is $P_{M_{2_i}}$, which, by hypothesis, is such that $P_{M_{2_i}}(x)$. Under what circumstances would an arbitrary y that is such that $P_{M_{2_i}}(y)$ also be such that $P_{M_{1_i}}(y)$? Well, we know that there is $Q \in N$ that is such that $Q(x)$, namely, Q_{i_1} (i.e., Q_{i_2}).

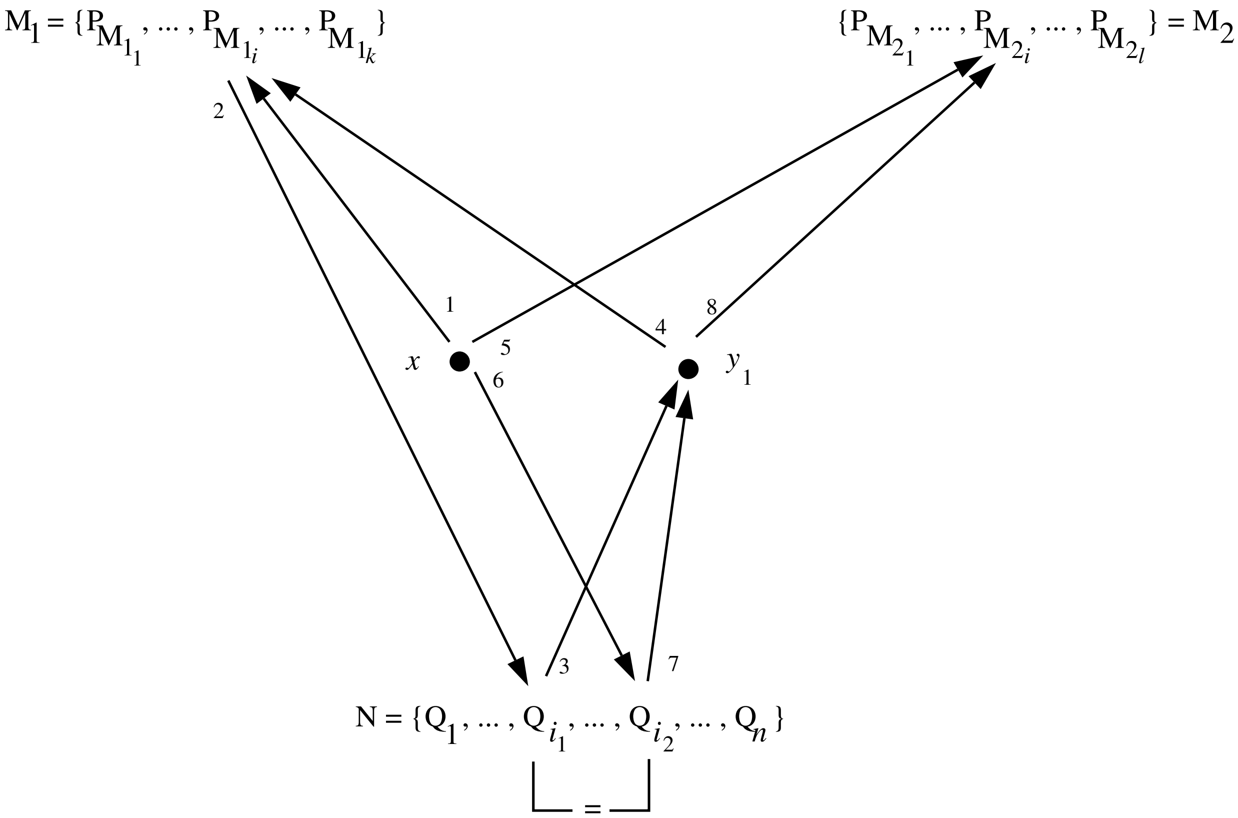

Figure 7.3: M_1, M_2 supervene on N. What is the relationship between M_1 and M_2?

And we know that an arbitrary y that is such that $Q_{i_1}(y)$ is also such that $P_{M_{2_i}}(y)$ *and* $P_{M_{1_i}}(y)$. So it appears that M_1 supervenes on M_2 (unless, perhaps, the 'necessity' clause rules that out). So, the chess properties and the Civil War properties appear to be mutually supervenient. But what does *that* mean? And there are further questions: Is supervenience symmetric? Transitive? (On these questions, see §7.5.4.3, below.)

And how does all of this relate to implementation? Let A be an Abstraction and I be its implementation in some medium S. Then it is S that has both A-properties and I-properties. So A-properties *could* supervene on I-properties. So, possibly, A is implemented by I if and only if A-properties supervene on I-properties. Indeed, Kim (pp. 43–44) sees supervenience as a very general version of the family of concepts that includes reducibility, etc. So perhaps it is the base relation in terms of which the others can be defined? I am uncomfortable with this, primarily because I see the generalized *semantic* relation as the fundamental one, and I take implementation to be a specific case of a semantic relation. So, too, for supervenience, which is, as we saw, a correlation relation.

Here is how Kim puts it:

> The macro-property of water-solubility may be exemplified in diverse molecular structures .... Certain gases may be water-soluble in virtue of their molecular structure M, and certain solids may be water-soluble in virtue of having a different molecular structure M*. Thus M will be the supervenient base of these gases' solubility in water, and M* will be that of these solids' solubility in water .... (Kim 1979: 45.)

I would say that water-solubility is *implemented in* diverse molecular structures. In any case, this sounds exactly like my Abstraction/Implementation relation. So, an implementation (Medium) has the properties that form the supervenient base for the properties of its Abstraction. Indeed, Kim says as much:

> Suppose that an occurrence of pain causes a limb withdrawal. ... This situation will be represented in the following way: This particular pain has a supervenient base, say, the excitation of certain neural fibers, although, as the functionalists argue, there may be no biconditional law correlating pain and this sort of neural base. (Thus, the notion of supervenient base corresponds roughly to the notion of "physical realization" often used by the functionalists.) (Kim 1979: 45.)

But I think it's better to say that the supervenient base is the *properties* of the implementing medium. But aren't the supervenient properties *and* the supervenience-base properties *both* properties of the medium? Is one set (the base) more "fundamental" than the other? Perhaps the properties of the supervenience-base are more "proper"—they are *properties*, whereas the supervenient "properties" are *attributes* (attributed by whom?—by external observers?).

Kim further describes the situation thus: Not only is a pain supervenient on a neural base, but the limb withdrawal is itself rather "Abstract" and supervenient on an anatomical/physiological base.

> ... there is a causal path from the supervenience base of the ... pain to this supervenience base of ... [the] limb withdrawal. It is also part of our account that

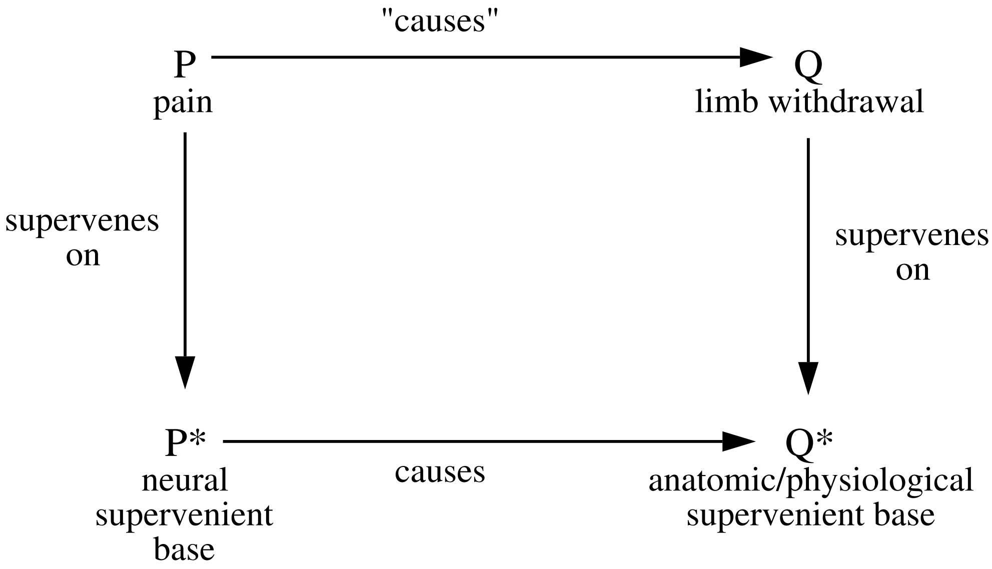

Figure 7.4: Kim's picture of the relationship among pain and limb withdrawal.

> the causal role of the pain vis-à-vis the limb withdrawal is *explained in terms of* this more basic underlying causal chain .... (Kim 1979: 45; my italics.)

Kim's picture is as shown in Figure 7.4. That is, the "causal" relationship between P and Q in Figure 7.4 is explainable (or definable) in terms of supervenience and physical causation. However, I recognize a *real* relationship at the top level, not merely a defined one; it may be *equivalent* to a combination of other relations, but it is *sui generis*. Where Kim says that "there is no independent causal path from ... P to ... Q or Q*" (p. 46), I would say that it *is* independent (though not really causal)—at the level of *functional* analysis. (Interestingly, in view of my remarks about good old-fashioned Cartesian dualism (§7.2), Kim calls this "supervenient dualism", and distinguishes it from epiphenomenalism.)

7.5.4.3 Supervenient causation.

Kim returns to causation in his 1983 essay, "Supervenience and Supervenient Causation". He begins by pointing out that one advantage of the notion of supervenience is that it allows us "to acknowledge the primacy of the physical over the psychological" (p. 45): The mental supervenes on the physical, but not vice versa (Kim 1982: 52), presumably because two entities could be in the same psychological state while in different physical states—i.e., psychological states can be multiply realized! Of course, if angels exist, then, presumably, they also think, so it's surely not the *physical* that we want to have "primacy" but—more generally—the implementing medium (cf. Fodor 1981).

But do we even want the implementing medium to have "primacy"? Do we want to say that nothing is or has a mind unless it is implemented? Would we want to say that nothing is an algebraic group unless it is implemented? In the case of minds, there does seem to be a desire for *behavioral processes*, hence for an implementation that can behave. But for the case of groups, there isn't. So, in general, both terms of the Implementation–Abstraction relation are of equal importance. One can't be *fully* understood (by a third party) without the other.

Kim defines 'weak supervenience' as before (p. 46; cf. §7.5.4.1, above). Note, again, that the object in the domain of the supervenience relation—the one that can have *both* the supervenient and the supervenience-base properties—*isn't* (merely) a computer or a physical object. It must be something (capable of) having *both* properties, such as a *person* (rather than a physical, human body) or a computer *process* (cf. Haugeland's objection, 1983: 65). A problem with supervenience is that there can be two "physically indistinguishable worlds" that are not also "psychologically indistinguishable" (p. 40). For instance, a computer c_w in world w running program p might be exhibiting some mental processes, but another computer $c_@$ in the actual world running p might not be. But is that so bad? Recall the chess/Civil War and chemical/mathematical lattice examples. In any case, Kim offers 'strong supervenience' as a remedy:

> A *strongly supervenes* on B just in case necessarily for each x and each property F in A, if x has F, then there exists a property G in B such that x has G, and *necessarily* if any y has G it has F. (Kim 1983: 49.)

and he points out that "Both relations are transitive, reflexive, but neither symmetric nor asymmetric" (p. 49). Transitivity is good; it's needed to account for levels of virtual machines, each of which can be said to supervene on, or be implemented by, a lower-level machine. Reflexivity, though, does not seem to be a property that we would want implementation to have. This means that *every* implementation *must* have implementation-dependent side-effects, since every implementation of an Abstraction will contribute something over and above what the Abstraction specifies.

If supervenience is non-symmetric, then it's possible for two properties to supervene on each other. Could each be an implementation of the other? That *seems* counterintuitive. surely, two things can implement each other—or be semantic interpretations of each other—but not at the same time. There is a directionality, a point of view of the third party that *uses* one domain as a semantic interpretation or implementation of the other—recall the discussions of the asymmetry of antecedent understanding (Chs. 2–4). So it looks as if supervenience is *not* implementation.

Is it reduction?

> If you believe that the mental strongly supervenes on the physical, you are committed to there being a physically necessary and sufficient condition for each psychological state. The physical base ... may not even be humanly discoverable; as a result it may be *unavailable* for a physicalist reduction or explanation of the psychological state. ...
>
> Thus, strong supervenience is not the same thing as the *reduction* of the supervenient family to the base family; reduction is an explanation procedure, and to carry out a reduction we must identify for each basic supervenient property its supervenient base property. (Kim 1983: 49–50.)

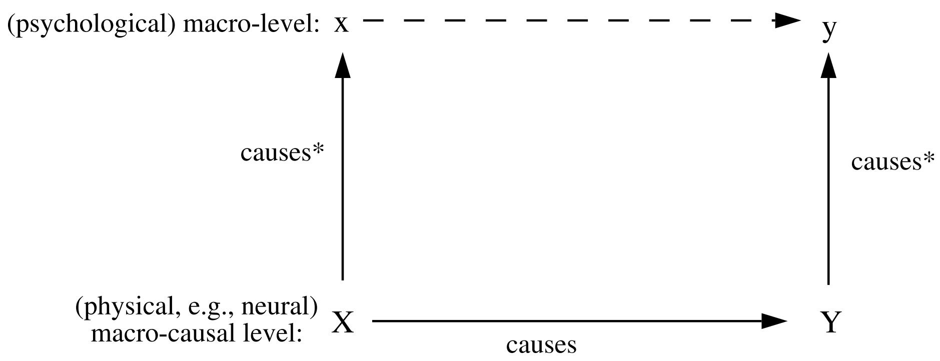

Figure 7.5: At the (physical) macro-causal level, physical state X really causes physical state Y. The relation between that level and the (psychological) macro-level—causation*—is really "mereological supervenience" (cf. p. 52).

But for an implementation, as for a reduction, we would need the identification, unless, perhaps, we have a certain kind of connectionist implementation. So, insofar as we *need* the identification for an implementation, implementation is *not* the same as supervenience. And insofar as we *don't* need the identification for an implementation, implementation is not the same as reduction. Independently, implementation is not reduction, because in reduction, one can jettison the reduced theory, but for implementation, one *needs* the Abstraction to tell you what's going on.

Kim introduces the notion of "supervenient causation" (pp. 50–51). Consider the situation shown in Figure 7.5. Here, the relation between states at the psychological level is what Kim calls 'supervenient causation', which he takes to be *reducible* to the *real* causation at the physical level. (Later (pp. 54–55), he also considers the relation between X and Y to be one of supervenient causation.) He points out that this is *not* eliminative materialism, because the psychological states are not "rejected". Nor is it Cartesian dualism, because supervenience is not causation. Nor is it epiphenomenalism, because supervenient causation is "as real as causal relations involving sundry macro-objects and their observable properties" (p. 55).

7.5.5 Summary.

So, what is implementation? We have seen that it is a very widespread phenomenon, taking many guises. It is a relation that obtains between two things—I have called them the Abstraction and the Implementation—when the Implementation is a "concrete" or "real" or "physical" thing that has all the properties of the Abstraction. But we have also seen that Implementations can be equally "abstract". So there are two sorts of implementations: abstract and concrete ones, the latter being "realizations" in some physical medium. We have seen that they typically have *more* properties than their Abstraction. So perhaps implementation is best construed (even etymologically) as a general term for *any* filling in of details; concrete implementations are fillings-in in concrete media. Thus, the notion of implementation comes along with a notion of "level": the more detailed level being "below" the "higher"—or more abstract—level, and the "concrete" or "physical" level being at the "bottom"—being the "foundation" as it were. Paul Smolensky characterizes it that way:

> For my own purposes, the crucial aspect of the implementation relation is this. Suppose we have a physical system S which at some ["lower- or 'micro'-]level of description L_μ is performing exactly the computation μ; that is, if we write down the laws governing the dynamics and interactions of those aspects of the system state that are characteristic of level L_μ , we find these processes to be exactly described by μ. If μ is an implementation of [a "higher- or 'macro'-level description] M, we are guaranteed the following: The states of this same system S have characteristics at a higher level L_M which evolve and interact exactly according to M: These characteristics define a description of S at the higher level L_M for which M is a complete, formal, and precise account of the system's computation.
>
> If μ implements M, then this constitutes the strongest possible sense in which μ and M could both be valid descriptions of the same system S. (Smolensky 1988: 59.) **[???]**

The one caveat I have here is that μ will typically have more details than M—it will say more about S, though, as we will see, what more it has to say may not be interesting from the point of view of M.

We have also seen that individuation, instantiation, reduction, and supervenience are all related to implementation, though "weaker" than it (cf., too, Smolensky 1988: 70n1). **[???]**

The single best "interpretation" of implementation seems to be that of semantic interpretation: I is an implementation of A in medium M if and only if I is a semantic interpretation or model of A, where A is some syntactic domain and M is the semantic domain.

In the remainder of this chapter, we shall look at some of the implications of this point of view: the role of the "implementation details", the question of whether an implementation is "the real thing", and the problem of whether anything can be an implementation of anything else.

7.6 IMPLEMENTATION-DEPENDENT DETAILS.

7.6.1 In the Details Lie the Differences.

Suppose we have two different implementations of an Abstraction. They may be implementations in different media, as, for example, implementations of the Stack abstract data type in Pascal using records and in Lisp using lists, or implementations of Cassie on a Vax running Franz Lisp and on a Sun running Allegro Common Lisp. Or they may be implementations in the *same* medium: for example, two implementations of a fully-equipped Ford Taurus LX—here what I have in mind is that the cars would be identical except, of course, that the metal, plastic, fabric, etc., of one of them would perforce be distinct physical objects from the metal, plastic, fabric, etc., of the other. For another example, consider two implementations of the Stack abstract data type using Pascal arrays; here, one implementation might use an n-dimensional array A with `top` $= A[n]$, while the other uses an $(n+1)$-dimensional array A with `top` $= A[0]$.

Clearly, the members of each pair of implementations will differ. The operations of the Pascal stack will be defined in terms, say, of record operations, while the operations of the Lisp stack will be defined in terms of list operations. The two Cassies' input–output behaviors ought to be the same, but the code will differ, so debugging will be a different process on each. One Taurus might dent more easily than the other, or get better gas mileage. (Even identical twins differ, as in Georges Duhamel's 1931 novel *Les Jumeaux de Vallangoujard*.) And clearly there will be implementation-dependent differences between the array-implemented stacks, some of which (e.g., "where" `top` is) are behaviorally (i.e., input–output) irrelevant and some of which (e.g., the size of the array) have behavioral consequences.

Jorge Gracia discusses the relation of an artist's "general idea of what he [sic] wants to do" and the final product, e.g., a sculpture:

> ... the sculptor's description is too general and does not identify those features of the sculpture that set it apart from others [that satisfy the description]. ... [t]he particular sculpture that the sculptor produces is not the result of his idea alone, but involves also the materials with which he works as well as the creative process itself that produces it. (Gracia 1990: 511–512.)

In general, then, when a given Abstraction is implemented in different media (or in different ways in the same medium), there will be implementation-dependent differences. Nevertheless, there will be some core, some essence, common to all of them in virtue of which they can be said to be the "same". This, I take it, is the point of Goguen et al.'s isomorphism construction.

Implementations are always more specific or detailed than their Abstractions, and they involve the implementing medium. This gives rise to *implementation-dependent side effects*. Similarly, ideas can be implemented in different languages (or differently implemented even in the same language). Clarity of exposition, literary art, and even cultural variety thrive on the implementation-dependent side effects due to the implementation-dependent differences. *Vive les différences!*

Consider the implementation of a mind. That is, suppose that (at some future time) we have a collection of algorithms that "account for" cognition—the Mind abstract data type, as it were. Suppose that we have neurological evidence that it is implemented in the human brain, and suppose that intelligence artificers (to use Dennett's happy term **SOURCE?**) have implemented it on a supercomputer. We should expect that there will be *implementation-dependent* differences between human minds and such computer minds. Does this mean that the computer mind is not a "mind"? I understand this question in the following way: Is the computer mind an implementation of the Mind abstract data type? The answer, by hypothesis, would clearly be 'Yes'. Are the differences "important"? That, of course, depends on what counts as being "important". Perhaps there will be a need to talk of *degrees* of "mindhood" (cf. Rapaport 1993b). Perhaps, for example, the Mind abstract data type will not be able to be fully implemented in dogs, or in chimps (or perhaps we will be able to distinguish between a Human Mind abstract data type and a Dog (or Chimp) Mind abstract data type).

Perhaps, in the long run, the *only* differences that will be of any significance will be the

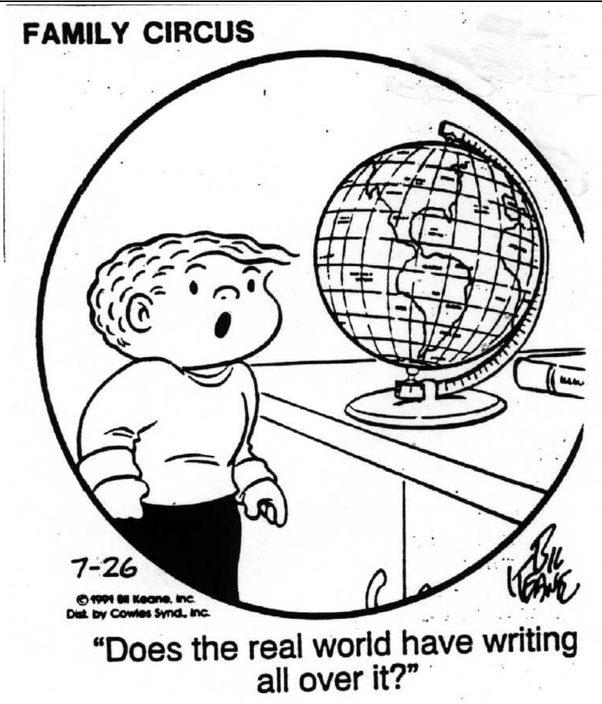

Figure 7.6:

implementation-dependent ones—the physical differences—and even these will be of no more (or perhaps no less) significance than the implementation-dependent differences that *currently* exist due to the fact that *your* mind is implemented in *your* body and mine in mine. Suppose, for example, that androids like Data of *Star Trek: The Next Generation* are commonplace. Would we—*should* we—behave differently towards them? Suppose, even, that some cognitive agents are "aware" or have "subjective experiences" (measured by, e.g., whether they have faces or are human, or by some primitive "feeling" or "intuition" that they are aware), while others are *not* thus "aware" (e.g., some computers). Suppose further that these two kinds of cognitive agents are not behaviorally distinguishable (perhaps only physically distinguishable—i.e., distinguishable on the basis of certain perceptual aspects of their implementation). Given this behavioral indistinguishability, I would say that we would *not* (better: should not; consider, after all, the ugly varieties of racism) behave differently towards them—not even towards the *non*-"aware" ones: For even they, because they were behaviorally indistinguishable, would *claim* to feel pain, say; so it would be morally wrong to inflict pain on them. What, then, would be the difference between them? Only a linguistic convention.

7.6.2 Implementation-Dependent Side Effects.

Consider an object that is a model of something. How much of it is part of its role as a model, and how much is due to its own nature—to its implementing medium? From the fact that a globe is plastic, we do not infer that the world is plastic. Nor do we infer that the world has writing on it from the presence of place names and lines of latitude and longitude on globes, even though these *are* part of its role as a model (Figure 7.6). Where do implementation-dependent side effects come from, and what, if anything, do they do?

Implementation-dependent side effects are due to implementation-dependent details. The details come from situations in which the semantic domain is "larger" than the syntactic domain. These are situations in which "completeness" holds in the sense that everything in the syntactic domain is interpreted in the semantic domain, yet in which not everything in the semantic domain is an interpretation of something in the syntactic domain. Thus, individuals can have properties that their universal lacks:

> ... written, spoken, and mental texts are all individual insofar as they are not instantiable themselves. ... As individual instances, moreover, they presuppose corresponding universals, but the universal is not the same for the three types of texts. *For the written text, it would be a written type of universal even though the universal would not be something written anywhere.* (Gracia 1990: 505–506; my italics.)

Such implementation-dependent properties, we see, *can* be essential properties of the individual; we'll come back to this in §7.6.3.

Another source of implementation-dependent details is non-isomorphic models. Recall the discussion in §2.2.2 of non-isomorphic models of the group axioms (i.e., of the Group abstract data type). For example, consider two groups of different cardinalities (e.g., the cyclic groups of orders 2 and 3) or an infinite cyclic group such as the integers under addition and the Cartesian product of that group with itself (which, unlike the former, has two disjoint subgroups except for the identity). In each case, the implementation—the model—has features that are left *unspecified* by the Abstraction (in this case, the group axioms); they are implementation-dependent details. Indeed, even *isomorphic* models give rise to implementation-dependent differences: "In any isomorphic class there are models which differ on *all* non-empty extensions. For example, in any isomorphism class there is one model at least whose domain consists of odd integers and one whose domain consists of even integers" (Jardine 1973: 231; Jardine points out that this gives rise to Quinean indeterminacy of reference).

Sometimes, the implementation-dependent details are not important and can—or even *must*—be ignored. This is because the *purpose* of an implementation or model is often to aid in understanding the Abstraction. There are two sides to this coin: If the *Abstraction*—better, the syntactic domain, the domain to be modeled, the domain to be understood in terms of the model—is itself complex, we will want the *model* to be simpler.[10] Nonetheless, it will still have features that do not represent any part of the Abstraction: "It may be *richer* in properties, but these would then not be ones relevant to its object [i.e., the Abstraction]; it [i.e., its object] wouldn't possess

[10] "It is rather paradoxical to realise that when a picture, a drawing, a diagram is called a model for a physical system, it is for the same reason that a formal set of postulates is called a model for a physical system. This reason can be indicated in one word: simplification. The mind needs in one act to have an overview of the essential characteristics of a domain; therefore the domain is represented either by a set of equations, or by a picture or by a diagram. The mind needs to see the system in opposition and distinction to all others; therefore the separation of the system from others is made more complete than it is in reality. The system is viewed from a certain scale; details that are too microscopical or too global are of no interest to us. Therefore they are left out. The system is known or controlled within certain limits of approximation. Therefore effects that do not reach this level of approximation are neglected. The system is studied with a certain purpose in mind; everything that does not affect this purpose is eliminated" (Apostel 1960: 15).

them, and so the model couldn't be taken to represent them in any way" (Wartofsky 1966: 6–7). The extra properties are implementation-dependent details, to be ignored. Again, recall the construction used by Goguen et al., to divide out such irrelevancies.

Often, however, the details *do* contribute something: This is the realm of the implementation-dependent *side effects*—phenomena contributed by the implementing medium, not by the Abstraction. Some are behaviorally relevant, others not. That a stack's top is implemented as $A[0]$ rather than $A[n]$ is not behaviorally relevant. A high-level program that cares only about stacks and not about their implementation can—and does—ignore this. In programming languages with built-in data-abstraction mechanisms, such as Modula-2, they *literally* ignore—do not know—about the implementation details (cf. Parnas 1972).

But as side effects become more and more behaviorally relevant, they become more than mere *side* effects and can be of central importance. For instance, in the chess game played with non-standard pieces (cf. §3.2.1), the implementation of the Abstract chess pieces had confusing implementation side effects. Sellars's Texas chess, played with Pontiacs implementing bishops (§7.4.2.2), will have, if not *confusing* side effects, certainly *significant* ones—the chess board will have to be pretty large, and perhaps a speed limit will have to be imposed on the bishops. More significantly, problems with an implemented computer system may be due to details of the implementation that are not part of the original specifications. That is, the system might mathematically "satisfy" the specifications, yet still fail due to hardware faults:

> . . . hardware does from time to time fail, causing the machine to come to a halt, or yielding errant behaviour (as for example when a faulty chip in another American early warning system sputtered random digits into a signal of how many Soviet missiles had been sighted, again causing a false alert . . .). (Smith 1985: 635.)

This, I take it, is at the heart of James H. Fetzer's arguments against program verification (Fetzer 1988, 1991; cf. Nelson 1992, 1994).

To some extent, the notion of an implementation-dependent detail and its attendant "side" effects is a relative one. Recall Gracia's example of the individual written text and its non-written "written type of universal". There would, however, be a further universal, of which the "written-type" and "spoken-type" of universals are instances. (For example, a high-level universal might be Lincoln's Gettysburg Address, of which the written-universal and the spoken-universal are species; one written individual falling under the former would be the one Lincoln allegedly wrote on the back of an envelope, and one spoken individual falling under the latter would be the one Lincoln uttered on 19 November 1863). Or compare Euclid's algorithm for computing greatest common divisors with that algorithm implemented in Pascal, and with that algorithm implemented in Lisp; each of these can be (further) physically implemented as processes on a variety of machines. Each level of Abstraction or implementation ignores or introduces certain details. One level's implementation detail is another's Abstraction. That is, we can (via a kind of reverse engineering) "abstractify" an implementation's details, after which they are no longer "details" *relative to* the new (more detailed) Abstraction. Consider, for example, the Stack abstract data type and the N-Element Stack abstract data type. A Pascal n-element array-implementation of a stack (*simpliciter*) will have as an implementation detail (yielding behaviorally observable side effects) that it can only store n elements. Yet the very same code will *also* be an implementation of an N-Element Stack and, as such, will *neither* have that feature as an *implementation*-dependent detail *nor* as a *side*-effect—indeed, it will be an essential feature.

Note that we have two senses of 'abstract' here: the sense in which abstract data types, specifications, and blueprints are "abstract" (relative to implementations) and the sense in which to abstract is to eliminate (or ignore) "inessential" "details": "every model deals with its subject matter *at some particular level of abstraction*, paying attention to certain details, throwing away others, grouping together similar aspects into common categories, and so forth" (Smith 1985: 637). Note, too, that the model need only be "*assumed* simpler" (Rosenblueth & Wiener 1945: 317; my italics): The implementation-dependent details are *ignored*, not *eliminated*. They are parts of the model that are *not* (intended to be) representations of the system being modeled.

7.6.3 Qualia: That Certain Feeling.

The view of implementation as semantic interpretation, with its implementation-dependent details giving rise to implementation side-effects, suggests a solution to the puzzles of qualia. Qualia, roughly, are the subjective, qualitative "feelings" or "sensations" or "experiences" that accompany various mental states and processes. Examples are the "look" of blue (as opposed to yellow, and of yellow as opposed to blue) and the "feel" of pain (or, for that matter, tactile sensation *simpliciter*). The puzzle is that these are "private" or subjective phenomena: Only I can know what my sensation of blue looks like or what my pains feel like (or that I am in pain). You cannot know what my sensation of blue is like or what my pain feels like, or know that I have any blue-sensation or that I am in pain. You can, perhaps, feel a pain that "is like" my pain—though how would you (or anyone, for that matter, including me) really know that it "is like" mine, since you can only feel your own (cf. Smith's problem about knowing whether our models match the world, §2.7.1). In any case, your pain is not *my* pain. You can, perhaps, determine that I am in pain—but only on the basis of my publically observable physical behavior, and that, of course, could be mere show or—more radically—be "real" pain behavior unaccompanied by any qualitative painful sensation. So, qualia are private, hence "mental" (according to a well-accepted tradition). Hence, they ought to be explainable functionally or as part of the Mind Abstraction. Yet functionalism seems incapable of explaining them, or so the puzzle goes.

The way out, I propose, is *to view qualia as dependent on implementation side-effects*. This does not resolve the puzzle by itself, however, for we still have to account for its privacy.

Let's begin with the problem of "absent qualia": the possibility that, for example, I feel no pain in circumstances in which others do, yet I am not oblivious to the pain stimulus—I behave appropriately. Thus, an experimenter sticks pins in my right hand and in yours. We both wince, withdraw our hands, perhaps cry out; we both say that the pin-pricking hurts, and we complain of residual soreness over the next several hours. Yet you *feel pain* and I don't. Or so we suppose for the sake of argument. The questions are: (1) Is this possible? (2) Am I any "less" of a mental agent because of my lack of feeling? The issue is sharpened when I am replaced by a computer or, better, an android: Does the android feel pain? We suppose not. But why? The central issue here is one of subjectivity, the same issue that is at the heart of the Chinese Room Argument: Does an entity that passes a Turing-like test—in this case, one for pain/pain-behavior—"really" have the phenomenon being tested for? And, if *not*, does that mean, despite its behavioral indistinguishability from a human that *does* have the phenomenon, that it is only "going through the motions" and not "really" feeling, using natural language, or thinking?

I have mixed feelings about this (if you'll excuse the pun). On the one hand, I want to say that insofar as having—or lacking—the private sensation has *no* behavioral consequences (not even to my being able to describe my pain-sensation in exquisite and poetic detail—whether I have it or not), then it is *not* part of the Mind Abstraction. If I *do* feel pain, then my sensation must be due to my body—it is an implementation side-effect. I can, of course, perceive the pain sensation. Moreover, it *is* possible that the Mind Abstraction can deal with this despite the fact (if fact it be) that, despite the privacy, it is not a mental phenomenon: For the Mind Abstraction will have, let's say, a variable or data structure of some sort whose value *would* be the sensation if I *had* a sensation and whose value is unassigned otherwise. The assignment of a value to this variable or data structure is input from my body. That is how it is implementation dependent.

On the other hand, I think it is plausible that there are never any absent qualia. Take pain, and consider the following computational implementation of it suggested by Stuart C. Shapiro (in conversation): Imagine a computer terminal with a pressure-sensitive device hooked up to the central processing unit in a certain way that I'll specify in a moment. (All of this ought, by the way, to be able to be done with current technology.) Program the computer with a *very* user-friendly operating system that allows the following sort of interaction (comments in parentheses; cf. Figure 7.7):

(User logs in, as, say "rapaport")

System: Hi there, Bill! How are you? What can I do for you today?
(Assume that this only occurs at the first login; the operating system, assume, is capable of some limited, but reasonable, natural-language conversation.)

User: I'd like to finish typing the paper I was working on yesterday—file "book.30sep92".

System: No problem; here it is!
(The file is opened. The user edits the file, closes it, and then hits the terminal sharply on the pressure-sensitive device. Assume that this device is wired to the computer in such a way that any sharp blow sends a signal to the central processing unit that causes the operating system to switch from *very*-user-friendly mode to "normal" mode.)

System: File "book.30sep92" modified and closed. Next command:

User: I'd like to read my mail, please.
(System runs mail program without comment. (User exits mail program.)

System: Next command:
(User logs off; logging off in the context of having struck the pressure-sensitive device causes the operating system to switch to yet another mode. The next day, User logs in . . .)

System: Rapaport. Oh yeah; I remember you. You hit me yesterday. That hurt!

Now, what's going on here? We have a computer with an artificial-intelligence operating system that is exhibiting pain behavior. Modulo the differences between the computer and a human, and the limitations of the natural-language interface, behaviorally (or, from the intentional stance) it is reasonable to infer (or assume) that the computer was in pain when I hit it. But did it *feel* pain? How do *humans* feel pain? We feel pain when certain neurons are stimulated and certain signals are sent to the brain. Now, in our computer, certain wires connecting the pressure-sensitive device with the central processing unit are "stimulated" and certain signals are sent to the central processing unit. Where's the difference between human and computer? Perhaps the difference is

Figure 7.7:

that, for humans, there is a "pain-sensing" neuron in the brain that is stimulated when a human is hurt. It gets its input from the pain neurons (C-fibers, or whatever), which also send their input to certain motor neurons that result in typical pain behavior (or perhaps the pain-sensing neuron sends its output to the motor neurons). Fine; build a similar such device into the central processing unit and operating system. The cases are parallel. There is a quale in both cases. Ah, but what does the computer's pain *feel like*, you ask? I don't know. Do you know what *my* pain *feels like*? We'll come back to this in a moment. My point, for now, is that pain qualia can *and will* arise whenever there is pain-behavior, and the same holds, *mutatis mutandis*, for any qualia.

Consider, next, the problem of "inverted" qualia (or, for that matter, "differential" qualia): The general problem of accounting for the particular "feel" of a qualitative experience, assuming the *presence* of qualia: Does your pain feel like mine? Does your sensation of blue look like mine? In the most perverse case—the inverted spectrum case—your sensation of blue is just like my sensation of yellow, and vice versa, all across the spectrum (possibly excepting a fixed point?); or, in the inverted-pain case, your feeling of pain is just like my feeling of pleasure, and vice versa. Can this be? How? Well, first, it seems plausible that something like this, if not quite so extreme, *can* be. There are the experiments with inverting lenses **[ref; check Cole 1990?]**, in which the subject becomes acclimated to seeing the world upside down—behavioral indistinguishability with distinct qualia. There appears to be no reason in principle not to be able to adapt this to inverted spectra (Cole 1990). And many of us can experience a similar phenomenon by closing one eye: In my own case, at least, colors appear distinctly different to each of my eyes; by crossing my eyes so as to produce a double image, I can even compare the differences in color.

Again, I suggest, this is merely an implementation-dependent side effect. Rather than speculating on how the brain might be wired, let's again consider a computer example. Consider two computer programs with the same input–output behavior, written in Pascal using stacks. Suppose that one of them implements the Stack abstract data type as an n-element array $A[0], \ldots, A[n-1]$ with `top` $= A[0]$, while the other implements it as an n-element array $A[0], \ldots, A[n-1]$ with `top` $= A[n-1]$. The internal mechanisms—the *implementations* of the stacks—are "inverted" with respect to each other, yet this is behaviorally undetectable and irrelevant. Granted, here there is no issue of "qualitative feel", perhaps. Yet the point is that the differences—and there clearly are differences, although not input–output ones—are implementation dependent. The analogue of qualia are implementation-dependent side effects. Similarly for pain: The *sensation* or *feeling* of pain, in humans, might be something that your body has (or does, or undergoes) when, for instance, you step on a tack. But that it feels the way it does is an epiphenomenon (so to speak) *of the body*. Were the same mind implemented in a different body (as in Leiber's *Beyond Rejection*), perhaps the feeling would be different (or absent).

Are qualia "mental" phenomena? They are private, yet (I hold) they are implementation dependent. Does that mean that functionalism (or strong artificial intelligence) fails to "model" some mental phenomena? That's certainly one interpretation, one move that can be made in the philosophical game. Or does it mean that what it fails to model (pain, spectra inversion) isn't mental? That is, of course, another equally plausible interpretation, another move that's open, *unless* one defines the mental in terms of what is "private" (i.e., not publicly accessible). Yet another option is that *some* of what we call 'mental' is body (or implementation) dependent, though this is not available for those who define bodily phenomena in "public" terms.

The position I find congenial is to make the "syntax" "complete". Recall my suggestion that implementation side-effects were due to situations where the semantic domain exceeded the syntactic domain. In such cases, we can *extend* the syntactic domain to make it match the semantic domain (cf. Rapaport 1981). Although any Mind Abstraction may be incomplete in *this* sense of having implementation side-effects, the *fact* of having such implementation side-effects can be made part of the Abstraction, as indicated earlier with my discussion of variables whose values are assigned externally. In this way, to paraphrase Tolstoy, every cognitive agent will "feel pain", but everyone's pain will "feel" different.

The random digits "sputtered" by a faulty chip that were interpreted as enemy missiles were also implementation side-effects—(physical) implementation details that yielded or gave rise to "mental" behavior: The computer interpreted certain physical configurations as meaning something (cf. Tenenbaum & Augenstein 1981)—it "felt" them in a certain way, so to speak. A feeling of pain is the mind's *perception* of a physical event. Thus, qualia can be thought of as the locus of "interaction" of mind and body, of Abstraction and Implementation.

It is not, therefore, unreasonable that qualia would be physical, yet "private" in virtue of being part of the Abstraction. The actual "feeling" belongs, and only belongs, to the implementation. Consider *Hamlet's* sadness (at, say, his killing of Polonius) as opposed to *Olivier's* sadness (at, say, learning of the death of a good friend) and as opposed to *Olivier-qua-Hamlet's* sadness. In the Method School of acting, Olivier-qua-Hamlet's sadness would be an implementation of Hamlet's sadness in the medium of Olivier's sadness. This is to be distinguished from Olivier merely "acting" sad (perhaps a case of absent qualia?).

The privacy of qualia just *is* its subjectivity. Compare the following three experiences:

Figure 7.8:

Seeing in a mirror something on your eyelash; seeing it (out of the corner of your eye) directly on your eyelash; feeling it on your eyelash. These are three distinct and different sensations, only one of which (the mirror case) is "objective" or external—from the third-person point of view.

Is there any *more* or *other* difference between these? I think not. Pain, etc., are just the way things are perceived in certain circumstances, some of which cannot be experienced by anyone other than the subject. "That certain feeling" *ought* to be private, because it's due to the *experiencer's* implementing medium, not anyone else's.

7.6.4 The Real Thing.

One fancy rephrasing of the question whether machines can think is this: Is a computer "simulation" of a mind "really" a mind? Compare this, for the moment, to another question (cf. Dennett 1978, Hofstadter 1981): Is a computer simulation of a hurricane "really" a hurricane? To answer questions like this, we need to spell out what a "simulation" is, what a hurricane (or a mind) is, and what it means to say that something "really is" an X.

The first observation I would like to make in this regard is that experience with an *implementation* of some Abstraction can change our understanding of the nature of the *Abstraction*. Can "straight lines" be implemented in a non-Euclidean geometry? The answer is 'Yes', but they aren't "straight" anymore; they can only be implemented as *geodesics*: shortest distances between two points. So, on a sphere, the implementation of straight line is a great circle. Similarly, consider the implementation by airplanes of flying: Airplanes fly, but not the exact way birds do (e.g., although their wings might have more or less the same shape, planes don't flap them). Planes fly only (or at least?) in the sense of moving through the air without touching the ground. No doubt that needs to be refined, so as to rule out long jumps (but mightn't a *very* long jump *be* flying?). Yet another refinement might replace the reference to air with a general term for a fluid medium: It has been suggested that the knowledge-representation community's favorite flightless bird—the penguin—does indeed fly ... in water (Ackerman 1989: 45–46). The point is, as we saw earlier

(§7.6.1), that when an Abstraction is implemented in different media, there will be implementation-dependent differences, yet there will be some common essence to both, in virtue of which they can be said to be the same. Thus, what is "really important" about straight lines is that they are geodesics; that geodesics are "straight" in Euclidean space is an implementation-dependent side effect—an "accidental" property, if you wish.

Edsgar W. Dijkstra has been quoted as saying that "the question of whether a computer can think is no more interesting than the question of whether a submarine can swim" (cf. Mike Gobbi, posting on the sci.psychology newsgroup, 20 September 1994). Assertions of equivalence such as this are notoriously ambiguous. Does Dijkstra think that it's obvious that submarines *do* swim (and therefore that computers *do* think)? Or that it's obvious that they don't? Or that it's merely a question of whether we'll extend the meaning of 'swim' to cover whatever it is that submarines do? Suppose the latter. What is it that submarines do? They move in the water. But that's what swimming is,[11] though perhaps before the advent of submarines we thought that swimming *had* to be done by animals: Do fish swim? Surely. Do people? Perhaps only by extending the term. Extending the meaning of a term occurs when we realize or decide that a property that we thought was essential isn't. This goes a long way toward explaining the unease people feel when they're told that computers can think.

So, is this extension of terms such as 'fly' to planes, 'swim' to submarines, and 'think' to computers "merely" a metaphorical extension? It may be metaphorical, but it is not "mere":

> **Eus[ebius]:** ... I do wish you would stop using terms borrowed from human behavior [to describe monkey behavior]! You're being anthropocentric!
>
> **Soc[rates]:** Well, monkeys are anthropoids. Besides, do you want me to make up a new word for a phenomenon for every species that shows it? Should geneticists stop talking about inheritance because that term was borrowed from economics? (Altmann 1989: 260.)

There are two points: First, *refraining* from such extensions, metaphorical or otherwise, would force us to miss important generalizations. Second, as Lakoff and Johnson (1980) have shown us, metaphor is an unavoidable and central feature of our language and thought.

What we do have to be careful about is mixing our metaphors. That is, an implementation must be complete unto itself; we must not import or apply features from one implementation of an Abstraction to another implementation of the same Abstraction. Thus, to take the classic case, it is of course not true that computer-*simulated* hurricanes get *real* people wet. But they *do* get *simulated* people *simulatedly* wet (Hofstadter 1981; cf. Rapaport 1988, Shapiro & Rapaport 1991). "Obviously, a computer simulation of a stomach would only digest simulated food" (Johnson 1990: 46). And a "simulated engine wouldn't generate any 'here in the world outside the computer' power—but if you put it in a suitably simulated car, and engage the suitably simulated clutch, it will just fine drive down the simulated road" (Minsky 1991). In each of these cases, we do have "the real thing": A simulation of digestion *is* digestion, a simulated hurricane *is* a hurricane. More accurately, I propose, a computer simulation of human digestion is an *implementation* of the Digestion Abstraction, as is human digestion itself. The latter may be more familiar, more prototypical (cf. Rosch 1978), but both, just as Dijkstra observed of swimming, are really digestion.

[11]Unless, of course, swimming is flying in water!

(apparent?) input → P88 program → (apparent?) output
↓
P88 machine-language program
↓
Pascal program
↓
(actual?) input → Macintosh machine-language program → (actual?) output

Figure 7.9: Hierarchy of virtual-machine levels.

The difference between a "simulation" of flying or of a hurricane and what we normally think of as "real" flying or a "real" hurricane is that the former "are one step removed from reality ... [because they use] symbolic parameter values that represent physical behavior" (Johan Lammens, personal communication, 17 August 1990). I'm not sure about the latter, but I agree with the former: Computer simulations are not part of the "real" world (except, of course, in the sense in which *everything* is part of the real world). They exist in their own simulated world, and we must be careful about "transworld" attributions. Although a simulated hurricane will not get *us* wet—because that would require a "transworld" causal relation of a kind that does not exist—the simulated hurricane must have some of the "same" (or analogous) cause–effect relationships with denizens of *its* computer universe (e.g., getting simulated people simulatedly wet) in order for it to count as a simulation—in order for both it and 1992's Hurricane Andrew to be *implementations* of the Hurricane Abstraction.

There is, however, an important family of exceptions to this principle of segregation. Computer simulations of semantic or information-processing systems are not only implementations of them, but can interact with other such implementations. In other words, if they were simulated hurricanes, they *could* get *us* wet.

A clear example of this is the computer simulation of computation itself. In one of my introductory computer science courses, I occasionally use a piece of software called the "P88 Assembly-Language Simulator". "P88" is (a fragment of) an assembly language for a hypothetical machine (Biermann 1990). (We can ignore for now whether it really *is* an (incomplete) assembly language *even if* "just" a toy one, since that is irrelevant to the point I want to make.) The P88 Assembly-Language Simulator is a Pascal program (actually, a ThinkPascal program). As such, it must be compiled into the machine language for the Macintosh computer on which it runs. My students and I can write P88 programs and "assemble" them into (a simulation of) a P88 machine-language program, which, in turn, is interpreted by Pascal as a certain Pascal program, which, in its own turn, is compiled into a Macintosh machine-language program, and executed. The levels are shown in Figure 7.9.

Suppose, now, that I write a program *in P88 assembly language* that takes two integers as input and returns their sum as output. When I cause this P88 program to be executed, a prompt appears on the screen, I input an integer, another prompt appears, another integer is input, and their sum is printed on the screen as output. Question: Was this a P88 computation? Answer (as usual): Yes and no.

Yes: It was, because the *algorithm* that computationally caused the sum of the two inputs to be output was a P88 algorithm that used data structures and instructions from the P88 language.

In other words, the two integers that were the input to the P88 program were input *to that program*, and their sum was output *by that program*.

No: It was not a P88 computation, because the P88 algorithm was executed by having a *Pascal* program perform *Pascal* computations that used data structures and instructions from the Pascal language. So, was it, then, a *Pascal* computation? Well, in some sense not really, because the Pascal algorithm was executed by having a Macintosh machine-language program perform *Macintosh machine-language* computations that used data structures and instructions from the Macintosh machine language. (Curiously, these are more like the data structures and instructions from P88 than from Pascal, but they were not executing or simulating P88 instructions or using P88 data directly.) At bottom, then, only a Macintosh machine-language program was really being executed and really computing the sum of two integers. In other words, the two integers that apparently were the input to the P88 program were actually input to the Macintosh machine-language program, and their sum was actually output by *that* program.

Yet I can, and do, use the P88 Assembly-Language Simulator to compute the sum of two integers. In other words, a computation by the P88 program was simulated by a Macintosh machine-language computation, but there *was* a computation nonetheless. The simulated computation *was* a computation. Moreover, it was a computation in two senses (once could say that there were two (simultaneous) computations (with the same input and output)): Ignoring *how* the P88 program was implemented, a P88 computation did yield the sum of the two inputs. And, ignoring the fact that it was simulating a Pascal simulation of a P88 program, the Macintosh machine-language computation also yielded the sum of the two inputs. It is important, I think, to note that the Macintosh machine-language program did so in a roundabout way: It was *not* the simplest possible such program to output the sum of two inputs, because it did that *not* simply by performing an addition, but by doing a number of other ("bookkeeping") operations that simulated a Pascal program simulating a P88 program.

As in Figure 7.8, mental "fingers" are an implementation (a semantic correlate) of actual fingers, and can serve (some of) the same purposes. Elizabeth (the child in the cartoon) is not using her actual fingers, and she *is* doing the work in her head—by using "head-fingers". Imagination and mental imagery can serve as a substitute for actual experience—one can solve problems by manipulating either the actual objects or models of them (see Figure 7.10).

Let's consider some examples, some of which we've seen in other contexts. A photograph of a map (cf. §2.7.3) can be used to find out where some city is, even though that wasn't the purpose or intent of the map in the ad. Copying information (*sic*) from a book (by Xerox or by hand) and then using that copied information (those copied sentences) rather than the original source is done all the time. We don't think twice about it or say that it's not "really" information. Here, unlike the previous example, there *is intentionality*.

Why is it information? Because of its syntax and the reader's interpretation of it. But it *carries* the information whether or not a reader interprets it. There is a possible problem: The information carried could be differently interpreted by another reader. But what's invariant is the syntactic structure. In any case, multiple interpretations are all equally good interpretations. Ah, but *is* the syntactic structure invariant? Examples such as the string 'NOWHERE', which could be analyzed as 'NOW HERE' or as 'NO WHERE', or weakly equivalent grammars suggest that the larger the context, the more aid there is in determining the syntactic structure.

What about computer simulations of minds? "The difference between a symbolic airplane

Figure 7.10:

simulator and a symbolic intelligence simulator is that the former models a physical system through the intermediary representation of parameter values, while the latter models behavior by behaving" (Johan Lammens, personal communication, 17 August 1990). Here's the insight: *We* don't interact with simulated hurricanes, so they don't get *us* wet. (As we've seen, they *do* get simulated people simulatedly wet—that's where the "internal representations" come in.) But we *do* interact with simulated minds. Now, how is that possible? *Is* it possible?

Do we thus interact, or do we only seem to? Having a conversation on some topic with a Turing-Tested simulated mind *is* having a conversation on that topic and *not* merely simulating having the conversation. For the latter is what you would do in a play. "When you get out of a TT session, something has changed: you have talked to a system about something, and most likely that has affected some of your own thoughts and beliefs" (Johan Lammens, personal communication, 17 August 1990). Of course, being in a play can do that, too, just as *reading* the play could. But in the Turing Test case, it's a dynamic, changing conversation.

So, how is the interaction possible? Because both systems deal with information, albeit implemented differently. But the implementations are "transparent" (as in the game of chess played with Staunton pieces; cf. §3.2.1).[12] One must, of course, be careful to distinguish—if possible—between information that only concerns the simulated world (or, for that matter, the real world) and information that can transcend the boundary. Smith cites an example of a training tape that was interpreted to be a real Soviet attack (1985: 00). **[???]** The very fact that this can happen shows that some "simulations" are indistinguishable from the real thing. Indistinguishable in what sense? In the Fregean *Sinn*-sense or Meinongian-object sense, not the Fregean *Bedeutung*-sense or the sense of actual objects: In the one case, there is a real-world referent (or Sein-correlate), but not in the other. At the level of *Sinn* or Meinongian object (or mental representation), all things are on a par. If we can't determine that they differ referentially, the default assumption should be that they don't. And even if we *can* determine it, it might not matter. (For more discussion of this, see Rapaport 1991a.)

So there is a difference between the digestion and hurricane cases on the one hand and the natural-language and mind–brain cases on the other: "... brains, unlike stomachs, are information-

[12]This replies to Jahren's objection in his footnote 1, p. 326. **SPELL THIS OUT**

processors. And if one information processor were made to simulate another information processor, it is hard to see how one and not the other could be said to think" (Johnson 1990: 46). That is, the difference is that it *is* the same stuff involved in the brain and computer cases: "Simulated thoughts and real thoughts are made of the same stuff: information" (Johnson 1990: 46). Well—not quite: Information is abstract; simulated and real thoughts are different *implementations* of the same *Abstraction*. (One might, however, want to say that they deal with the Abstraction *directly*, via "transparent" media.)

My claim, then, is that simulated mentality (or cognition, or intelligence) *is* mentality (or cognition, or intelligence). Recall the mental imagery debate, in which Stephen M. Kosslyn (1981; Kosslyn et al. 1981) argues that one really scans a mental (i.e., simulated) image, whereas Zenon Pylyshyn (1981) argues that one *pretends* to scan (*simulates* scanning) a real image. Or compare Searle's claim (1979) that in fictional language, one *pretends* to assert (one *simulates* asserting) rather than *really* asserting a pretended (or simulated) utterance—an utterance in a pretend-world.[13] In both of these cases, I side with the "really" people: Rather than saying that a computer *simulates* understanding something real, I would say that it *really* understands something simulated—and that in many cases, the simulated thing that it really understands is itself the real thing (internally represented).[14]

7.7 FROM MULTIPLE REALIZABILITY TO PANPSYCHISM.

Given the Principle of Multiple Realizability—the apparently obvious claim that there can be more than one implementation of an Abstraction, more than one model of a theory—an argument can be constructed for a variety of panpsychism (the view that everything is a mind). The argument, in its bare outlines, is this:

1. There is multiple realizability (of computational processes).
2. ∴ There is universal realizability (of computational processes) (by an argument of Searle's).
3. ∴ Anything can be a model of anything (else) (from (2), or by an argument of Wartofsky's).
4. ∴ Anything can be a model of a mind (from (3) by universal instantiation, or by an argument of Dipert's).
5. ∴ Anything can be a mind (by the argument of §7.6.4 that *models* of minds *are* minds).

Actually, as will be seen, there are several different arguments in this vicinity. Let's begin with Searle's argument from (1) to (2).

[13]Note the deictic shift; cf. Galbraith**?**, Segal**????**

[14]B. H. Webb (1991: 247) argues "that it is possible for a simulation to be a replication if the device used can not only represent but also instantiate the same capacities as the system." This seems congenial to my claims.

7.7.1 Multiple Realizability Implies Universal Realizability.

In "Is the Brain a Digital Computer?" (1990), Searle is concerned with the multiple realizability of computational processes. Hence, on the assumption that mental processes are computational, he is concerned with the multiple realizability of mental processes. A "disastrous" consequence of multiple realizability is that it

> would seem to imply universal realizability. If computation is defined in terms of the assignment of syntax, then everything would be a digital computer, because any object whatever could have syntactical ascriptions made to it. You could describe anything in terms of [**???**] 0's and 1's. (Searle 1990: 26.)

What evidence does Searle have for this claim that any (physical) object can be described computationally? And why is it disastrous? The latter question is easier to answer: According to Searle, universal realizability doesn't tell us what's *special* about the brain as opposed to other, less interesting, computational systems, such as the "stomach, liver, heart, solar system, and the state of Kansas" (p. 26). Perhaps—we'll come back to this (§7.7.3).

Searle claims that "For any object there is some description of that object such that under that description the object is a digital computer" (p. 26). This seems too strong. For one thing, there are certainly things that are non-computational in the strong sense of the Church–Turing Thesis. For another, merely assigning 0s and 1s to give an encoding (of, say, the atomic structure) of a physical object doesn't make it computational. To be computational, analogues are needed of Turing-machine instructions, control structures, states, the input–output tape, etc. At the very least, to be computational, an item must compute some function. So, what function does my pen compute? Well, I suppose it could be argued that it computes the constant function (or perhaps the identity function, or perhaps it loops forever—i.e., is undefined on all input). But that is trivial. On the other hand, consider the string of 0s and 1s that, according to Searle, encodes my pen. That's the Gödel number of some program (no doubt a trivial one, but who knows?). Does that make my pen an *implementation* (a model) of that program? No; it is an interesting correspondence, but not an implementation, because the interpretations of the 0s and 1s for the description of the pen are not those required for the program.

Smith (1982) has made similar observations. He describes a "computer" that "calculates oriental trajectories", which is, in fact, a car that drives west,[15] and he asks why this *isn't* a computer (p. 2). He notes that it does share two important features with computers, the second of which is close to Searle's claim: First, the oriental**???**-trajectory calculator *is* equivalent to a Turing machine, so that's not why it isn't a computer. But *why* is it equivalent to a Turing machine? Perhaps because it is input–output equivalent to a Turing machine that calculates oriental**???** trajectories? If so, then perhaps mere input–output equivalence is not sufficient: As we saw in §9.5, that a *function* is comput*able* merely means that there is a Turing machine that computes it (i.e., that has the same input–output behavior), but that does not mean that a device with that input–output behavior is *itself* a computer (that it *computes* its output from its input)—it could be a mysterious oracle. Now, the case of the car is not quite the case of such an oracle. The car, after all, has parts whose function and behavior contribute to the car's overall behavior (its output,

[15] Shouldn't that be *east*? Perhaps 'oriental' is a typographical error for 'orbital'? Or perhaps he means 'oriental' as in "pertaining to orienting oneself".

if you will). So why does Smith say it's not a computer? Because there are no *symbols* that "act as causal ingredients in producing an overall behavior" (p. 2)—symbols in the sense of markers of a formal syntactic system:

> In describing how a car works, ... the story is not computational, because the salient explanations are given in terms of mechanics—forces and torques ... and so forth. These are not *interpreted* notions; we don't posit a semantical interpretation function in order to make sense of the car's suspension. (Smith 1982: 3.)

I dispute that. First, there is a mapping between the physical parts and actions of the car and terms from physics (i.e., physical theory). Second, there is a mapping between (at least) terms from physics and my concepts. So we *do* interpret the car's parts and actions.

Here is Smith's response to the claim

> ...that we 'interpret' steering wheels as mechanisms for getting cars to go around corners—... this is a broader notion of 'interpret' than I intend. I mean to refer to something like the relationship that holds between pieces of language, and situations in the world that those pieces of language are about. (Smith 1982: 4.)

But that distinction is one that can't be drawn (or so I tried to argue in Chapter 2)—*both* are interpretations. The threat of universal (or near-universal) realizability is expressed by Smith thus:

> ... the 'received' theory of computation—the theory of effective computability that traffics in recursive functions, Turing machines, Church's Thesis, and the rest ... does not intrinsically identify the class of artefacts that computer science studies. ... [I]t is too broad, in that it includes far more devices within its scope (like chairs and Rubik cubes) than present experts would call computers. The problem stems from the fact that Turing equivalence (i.e., computing the same function) is a *weak, behavioral* metric, and we are interested in a theory that enables us to define *strong, constitutional* concepts. (Smith 1982: 5.)

I'm willing to accept Rubik's cube, however. The difficulty is that "the class of artefacts that computer science studies" is an *intended interpretation* that the theory of computation just won't let us get our hands on, any more (but, equally, no less) than Peano's axioms let us get our hands on the natural numbers. If chairs are included (as Dipert argues; see below), so be it. Even if we strengthen the theory to talk about *algorithmic* equivalence, and not mere input–output equivalence, we'll still get multiple, hence unwanted—or, better, *unexpected*—realizations.

7.7.2 Everything Models Anything.

If there is universal realizability, then anything can be a model of anything (else). This is because, from the assumptions that, given an arbitrary Abstraction, everything can be an implementation (or realization) of it, it follows by universal generalization that everything can be an implementation (or realization, or model) of anything. Alternatively, as we saw in our discussion of Wartofsky in Section 2.6.2, step (3) of the argument for panpsychism follows from the assumption that everything shares at least one property (and perhaps infinitely many) with everything (else).

7.7.3 Everything Models Mentality.

Clearly, if anything can be a model of anything (else), then anything can be a model of a mind. An argument explicitly for this consequent has recently been offered by Randall R. Dipert (1990).[16]

Dipert begins by reminding us that David Hilbert's philosophy of formalism took numbers in "purely structural, formal terms ... [C]hairs, beer mugs, or whatever could just as well represent/exemplify numbers (under the right interpretation) as do numerals or our thoughts of numbers" (p. 6).[17] Similarly, "programs, together with their hardware implementation ... may not look much like more usual [i.e., biological] embodiments of minds" (p. 6), but they could be, in the same way that chairs can be embodiments of numbers. However—or so Dipert observes—not even adherents of so-called "strong AI" would take *chairs* as embodying minds, because "brains are much more complex than chairs, and so chairs and tables lack some of the structural features of mental properties" (p. 6). However, I am an adherent of "strong AI" and am as willing to accept the claim about chairs as I am about the standard water-pipes-and-valves model, which I am quite willing to do. Dipert (along with Searle and Smith, evidently) thinks this is problematic. Here's the argument that shows why:

(P1) Ordinary, middle-sized objects at room temperature (let's call them OMSORTs, for short)—e.g., chairs, coffee mugs, baseballss (and, presumably, brains)—are highly complex, dynamic entities (pp. 7–8).

(P2) Suppose there is a good cognitive science theory T of the sufficient conditions for "cognition and other mental processes" (p. 8).

(P3) Suppose there is an AI system C that implements T (p. 9).

(C1) $\therefore$ By formalism, C has cognition (p. 9).

(P4) For every OMSORT O, there is an interpretation such that O exemplifies T (although we might not be able to exhibit the interpretation that does the job) (pp. 9, 11).

(C2) $\therefore$ By formalism, O has cognition (pp. 9–10).

> These reflections should also make us resist our initial temptation to say that exemplifying some humanly-graspable ... set of properties [is] *sufficient* for having mental properties—unless we are willing to say, with Leibniz, that everything is a mind. (Dipert 1990: 11.)

Thus,

(P5) Conclusion (C2) is absurd or uninteresting.

(C3) $\therefore$ Conclusion (C1) is absurd or uninteresting.

---

[16]Dipert himself is sympathetic to the conclusion, even though he is playing devil's advocate in criticizing it; cf. Dipert 1990: 20n16.

[17]Cf. Hilbert's *Gesammelte Abhandlungen*, vol. 3, p. 403, as cited in Coffa 1991: 135.

Now, one difference between the arbitrary OMSORT O and the AI system C is that for C we *do* know what the interpretation function is: We can *understand* how and why C behaves as it does; we can interpret C's behavior as a mind. We accept it as such (this is what we do in our everyday solution of the problem of other minds).

Note, too, that some OMSORTs that we *might* very well be willing to accept as implementations of minds—namely, connectionist implementations that haven't been "properly treated" (Smolensky 1988)—are such that we might very well *not* understand them (what, e.g., do the connection weights "mean"?).

What's wrong with Leibniz's position? Mainly that if everything is a mind, then we can't explain the difference between a *human* mind and a coffee mug. In this regard, we might be no more worse off than a topologist who, as the joke has it, can't distinguish a doughnut from a coffee mug, since both are toruses. To say that there is a way to view doughnuts and coffee mugs such that they are alike is *not* to say that there is no way in which they differ. Similarly, if computational cognitive science tells us that, from a certain perspective, brains and mugs are alike, that's not to say that, from some other perspective, they're not. We *can* explain the difference between a mind and a mug: The mug mind can't communicate with us and therefore is irrelevant. That is, the mug *qua* mind can't communicate; the mug *qua* coffee-holder is a perfectly functional device. Brains *can* communicate, but they can't hold coffee—so we don't use them for that purpose. Not to be too macabre, we *could* use a brain as a paperweight, I suppose; a Martian (or a Black Cloud) might, and might never realize that its paperweight implemented a mind, any more than we realize that a coffee mug might. As a further analogy, compare a high-level program (e.g., to compute greatest common divisors, or even an AI program) with a machine-language version in the same way we have been comparing brains with mugs. The latter might not *look* like the former, but under the right circumstances it might very well behave like the former.

What, then, is the import of Dipert's argument? Is it merely that, for any theory, there are infinitely many models, many of which are non-isomorphic (cf. Mac Lane 1981: 467) and many of which are not the intended model(s) and are such that we did not antecedently take them to be models? If so, so what? Sure, there *have* to be unintended models, and there is no way to pick out or mark or identify the intended ones; that's one of the main lessons of the theory–model relationship. What we learn from the existence of unintended models (assuming that we are completely satisfied with the *theory* of which they are models) is that some things have properties and features that we didn't expect them to have.

Of course, Dipert's transcendental argument merely shows that it's highly likely that OMSORTs can be taken as (models of) minds, not that they *are*. Two highly complex things, just because of their high complexity, need not be models of each other. (Two highly complex patterns need not be matchable.)

On the other hand, Dipert's claim might be the weaker one that (it is highly likely that) there are *some* OMSORTs that model the cognitive theory T but that we would not antecedently have taken as an intended model. I think we can only bite the bullet on this. But perhaps it can be made palatable: Suppose the OMSORT is a (particular) baseball. Imagine complex dynamic *processes* "within" the baseball, presumably at the subatomic level, that model the mental processes. What, for instance, might correspond to perception? (Does the baseball "see"?) Perhaps nothing so corresponds in the sense of external causes of internal processes, but there might be internal processes that, in a methodologically solipsistic fashion, correspond to (or model) perception. Or perhaps there *are* such external causes, but they need not be actual events as *we* characterize (or see) them. The world that the baseball-mind "perceives" might (indeed, probably would) have different categories than the human mind or the AI mind (as Kant told us long ago; cf., too, Winston's Problem (§3.2.2.2.4) and Kirsh 1991: 12).

7.8 SUMMARY.

In this chapter, we have investigated the mind-brain relationship as a case of implementation. I have proposed that implementation is best understood as semantic interpretation (rather than as individuation, instantiation, reduction, or supervenience). It is a relationship between an Abstraction (a generalization of the notion of an abstract data type) and an implementing medium. This relationship can be found in music and language, as well as in the theory of abstract data types. In general, something is an implementation of an Abstraction in an implementing medium. Mind is an Abstraction that can be implemented in brains as well as in computers. Implementations, however, have implementation-dependent details that give rise to qualia—implementation side-effects.

If Mind can be implemented in a computer, could a computer that implemented a natural-language–understanding program really understand language? I, of course, would say 'Yes'. Searle, of course, says 'No'. So let's return to Searle's Chinese room, in the light of our present conclusions.