Computational Reflections

Brian Cantwell Smith

Contents

Acknowledgments

First and foremost I would like to honor the memory of Jim ("Jeem") des Rivieres, to whom this book is dedicated. Jeem devoted several years to implementing and supporting 2/3Lisp. Among other things he wrote an impressive run-time compiler for 3Lisp, collaborated on the writing of des Rivières and Smith 1984,[1] and uncovered and repaired a bug in the code for the 3Lisp implementation published in the original doctoral dissertation.[2] Jeem was a Founding Wizard of the Knights of the Lambda Calculus, an occasional organization headquartered at the top of the 3Lisp reflective tower. He unfortunately passed away in 2019, before this manuscript could see the light of day, but my thanks and debt to him endure.

I would also like to thank Daniel Friedman of the Indiana University Computer Science Department for his early and steadfast support of reflection. I believe it was at the Principles of Programming Languages (POPL) conference in 1983 that Dan first voiced public support for the earliest 3Lisp work. Thanks are also owed to Mario Tokoro and Akinora Yonezawa, mainstays of the early reflection community, who hosted a number of 3Lisp and reflection workshops in Japan in the 1980s.

Also collaborating in the development of 3Lisp and reflection, Jun Luo wrote an implementation of 3Lisp in Ruby[3],

---

[1]des Rivières and Smith (1984).
[2]Smith (1982)
[3]Available at: https://github.com/groupofn/3Lisp.Ruby

which remains today a working version of the dialect. I am pleased to call Jun a friend, and have benefitted immeasurably over the years from his musings about the nature of computing, the nature of reflection, and the nature of language, consciousness, and mind more broadly. Jun has a singular understanding of the ontological dimensions of 3Lisp and reflection.

Jun is also a member of "The Group of N," a small group of colleagues and former students who have come together over the years to discuss computing and, recently, to aid in working out details necessary for completion and publication of this volume. Current members include: Jim Brink, who as well as exploring 3Lisp's technical details has masterminded the innumerable organizational and copyediting details involved in assembling the complete manuscript; Güven Güzeldere, for advice on making the argument accessible to scientists and philosophers of mind; Stephen Hockema, for coordinating external references and internal cross-references; Jun Luo, for guidance on contemporary AI architectures; Henry Thompson, for the Foreword; and Matt Zukowski, for finalizing the figures. All six are close friends, as well as being colleagues and as supportive a group of intellectual fellow travelers as one could hope for. Not only do I owe them all immense thanks; it is unlikely that this volume would have seen the light of publication without their assistance and encouragement in its final stages. Hats off to every one of you, and warmest thanks.

Immeasurable thanks, finally, to my wife, Gillian Einstein, for assembling this team, and for her constant nourishment and stewardship of the entire process. Jill's partnership over many years has been absolutely sans pareil, in so many ways. I cannot thank her deeply enough.

Foreword

It is a distinct pleasure to have been asked to write a forward to Brian Cantwell Smith's remarkable book, *Computational Reflections*. I count myself one of Brian's oldest friends and colleagues, having known him since the late 1970s when we were both in our twenties, computer zealots working at Xerox Palo Alto Research Center (PARC).

This volume is ambitious but Brian's lead up to the work laid out in these pages supports his ambitions. He started undergraduate study at Oberlin College in Ohio in 1967, where his interests included both physics and religion. Leaving after only two years, he became employed as a programmer at the Division of Physics Laboratory of the National Research Council of Canada (Ottawa); his project involved both the Fermilab (Chicago) and the Lawrence Research Laboratory (Berkeley). At all three sites he "programmed like crazy" in machine language on PDP 9 and PDP 15 microcomputers, building systems for experimental control and data gathering.

After a few years, Brian moved to Cambridge, MA to take classes at the Massachusetts Institute of Technology (MIT), studying what was then known as Social Inquiry; in particular the politics of high technology. But Brian soon realized that the understanding of computing the social scientists were critiquing was not the understanding of programmers, the latter of which he described as, as "computation in the wild".

"What drove me out of Social Inquiry and back to [Computer Science] was needing to be back in the practice. That skill was not something that people on the outside understood. I had to go back into the heart of the beast, as it were."

To return to his "computation in the wild", he began taking classes geared toward applying to the Ph.D. program in Electrical Engineering and Computer Science.

There was one small hitch, however, to his continuing education; when the MIT administration discovered Brian had not completed an undergraduate degree, they determined he could not be registered for graduate study. However, he so impressed the newly-appointed head of the Artificial Intelligence Laboratory, Patrick Winston, that Winston gave Brian an informal oral exam in topics from the MIT undergraduate computer science curriculum and based on Brian's performance, awarded him the credits necessary for an MIT undergraduate degree. This cleared the way for Brian's admission to the MIT Computer Science graduate program. Brian's unusual depth in both the technical and synthetic were unsurprisingly recognized early; this remarkable intellectual combination has continued to this day.

At MIT, Brian met Terry Winograd, who in 1976 left MIT for Stanford. Terry invited Brian to spend the summer in the Understander Group at PARC, where Brian joined in the development of a Knowledge Representation Language (KRL), which came to embody some of the ideas that Brian later developed in his Masters and Ph.D. dissertations [refs].

These biographical details bring us to the time and place where he and I first met: PARC. It was really, from this moment that Brian began to build the expertise necessary for the profound but radical thesis of this book:

- Member of the Scientific Staff, Xerox PARC
- Director, Xerox PARC System Sciences Lab
- Adjunct Professor of Philosophy, Stanford University
- Founding member of Stanford University's Center for the Study of Language and Information
- Founding member and first president, Computer Professionals for Social Responsibility
- President of the Society for Philosophy and Psychology
- Professor of Cognitive Science, Computer Science, and Philosophy, Indiana University
- Kimberly J. Jenkins University Distinguished

Professor of Philosophy and New Technologies, Duke University
- Dean of the Faculty of Information, University of Toronto
- Professor of Information, Philosophy, Cognitive Science, and the History and Philosophy of Science and Technology, University of Toronto
- Senior Fellow, Massey College, University of Toronto
- Reid Hoffman Professor of Artificial Intelligence and the Human, University of Toronto

Working at PARC, "As an exercise in using KRL representational structures, Brian Smith tried to describe the KRL data structures themselves in KRL-0. A brief sketch was completed, and in doing it *we were made much more aware of the ways in which the language was inconsistent and irregular*. This initial sketch was the basis for much of the development in KRL-1."[1] (emphasis added)

In addition, Brian's input into the (never completed) KRL-1 meant that not only could some parts of a system's data be *about* other parts, but that these data would be more than just commentary. They would actually play a role in the system's operation. For Brian, working on KRL-1, this use of data was motivated by a desire to formulate aspects of knowledge representation – negation and disjunction – intensionally as knowledge about knowledge, rather than extensionally as primitives built into the vocabulary of the representation language itself.

Brian called this set of ideas idea *reflection*

There was an assumption at the heart of Brian's reflective architectures started in KRL-1 but which occupied just one section of one chapter in his proposed Ph.D.; its resolution has

---

[1]Bobrow and Winograd, 1978, "Experience with KRL-O: One Cycle of a Knowledge Representation Language", in *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, Morgan Kaufmann Publishers, Burlington, MA. Available online at https://www.ijcai.org/Proceedings/77-1/Papers/032.pdf.

been a lifetime of intellectual thought and work for Brian.

Looking back, it seems that any difficulties in the resolution acted rather like the grit in the oyster, stimulating Brian's wholesale reconsideration of the nature of computation, and Computer Science as currently practiced, which is, at its heart, what this book is about.

I daresay that from my vantage point as a career computer scientist, Brian has accurately diagnosed a problem that the whole field of computing has missed. Brian convincingly argues that this has led the field to be altogether stuck, but unable to see that. He believes that seeing the problem will unstick the field.

As Brian himself revealed to me in a recent conversation, "That this is important needs to be said." As computer scientists as well as philosophers of computing who care about the field, this claim deserves our attention.

A word to the wise: because the discussion is not only interdisciplinary but also relentlessly foundational, it cannot assume a particular starting place. So it is not generally an option to say that this or that term will be used *as in* one or another field. And this will especially affect the use of standard terms in computer science. The best interpretive strategy is probably to hold definitions in abeyance, and let the ensuing discussion and argument do the work of gradually fixing the meaning of the book's terminology.

The ideas put forward in this book are dense and at times, highly technical, but so very important. Brian has described it to me as written rather like a detective story, in which the same set of problems is explored repeatedly, getting closer each time to a complete and self-consistent picture. As he said, "…what I've written should be read more like novel than like a manual. What things mean will gradually take shape. Be patient".

As Brian says, be patient, dear reader. If you care about computer science, as a practitioner, theorist, or concerned citizen, this book matters for you. Its conclusions matter, even if parts of it are obtuse to you. So even if, as a programmer, you find it difficult to see why you should care if the theorists have

got it wrong, be patient. If you're a theorist, and you find Brian's critique at times irrelevant or founded in misunderstanding, be patient. If you're a citizen, and the technical details are off-putting, be patient.

If you *are* patient, and stay the course, when you get to the end you will be well rewarded by arguments elegantly made, important fissures in our understanding of computing spotlighted, and the way forward, clearer than before. You will surely have learned what a subtle and committed mind finds important. There is a vision presented in these pages; I know that I as a practitioner am asking myself what I can do to realize this vision. As citizens, we need to cheer from the sidelines, and keep asking these fundamental questions about computing, especially in our contemporary era of machine learning and AI. Our future depends on it.

<div align="right">

Henry S. Thompson
Toronto and Edinburgh
November 2024

</div>

1 Overview

This is a book about the foundations of computing—about what computing is, and how we understand it. It starts by identifying several problems in our current understanding that came to my attention in the early 1980s, in the course of developing what I called a reflective programming language. Though I made some effort to repair the difficulties then, the proposed reconstructions did not go deep enough. Cracks remained: confusions about what is sign and what is signified, what is concrete and what is abstract, what is model and what is modeled. The issues have permeated computer science since the outset, going back at least to the 1936 Turing paper that launched computer science.

The story told here builds on the 1980s investigation, but deepens the analysis and broadens the scope, ultimately growing into a systematic critique of the field as a whole. In the end I argue for reconstituting the theoretical framework on which present-day computer science rests. Conceptual clarity, intellectual coverage, and doing justice to programmers' intuitions require a foundation that is semantically clearer, ontologically less ambiguous, and more conceptually rigorous. Among other things, building such a framework means not taking 'computing' or 'computation' as a basic conceptual category, since the very notion of *computing something* is murky. Instead, a wholesale reconstruction is recommended, covering the full spectrum of active, intentional, physically embodied systems. No part of this edifice is co-extensive with computer science as we know it, though everything we have learned about computing has a natural place within it.

Strong conclusions, in need of strong defense.

A — Introduction

1 Meaning and Mechanism

Computation involves a dialectical interplay of meaning and mechanism.

The realm of **meaning**—or *intentionality*, to use the philosophical term—includes representation, language, symbolism, expression, signification, models, reference, theory, and so on: the general category of phenomena that are directed towards or "about" other things. Meaning is essential to computing—to any symbolic or intentional system, including all the information-processing machines we have built and programmed to do our bidding. Meaning in this broad sense is not limited to that which we have called computational. It is equally fundamental to our human ability to think, talk, calculate, reason, and imagine. To have a thought is to be in a state "about" (intentionally directed towards) some situation or state of affairs in the world—a knock on the door, the demise of the democracy, an inner pain. Similarly, to say that a machine computes a sum or calculates a trajectory, that an algorithm can determine whether a proposed distribution of resources is equitable, or that a computer is monitoring someone's blood pressure, requires interpreting a pattern of bits in or produced by a machine as meaning something.

In the case of digital computers, we use patterns of '0' and '1' to classify the states and configurations of the devices that perform these computations—the states and configurations that are in turn about the computation's subject matter. The numbers themselves are not the states; numbers are abstract, and there is only one number 1; states are numerous, and concrete (their occurrences have causal consequence). Per se, uninterpreted arrangements of bits are legion, and boring. What matters about those configurations—whether they denote numbers, represent answers to decision problems, can be taken as evidence of flight delays, or have any other significance, require acts of intentional interpretation. If we take the output of

a program to indicate that some route is the shortest, or that a particular refugee should be granted asylum, or that a given number is prime—all these things are interpretations. Even to say that a given set of bits is an *instruction* for a machine is an intentional claim, transcending the brute mechanical facts about machine's physical makeup and behavior.[1]

If meaning is essential, the realm of **mechanism** is if anything even more fundamental. It too has much wider scope than computation. Read broadly, the idea of mechanism, or of mechanical workings, along with associated concepts of causality and effectiveness, underlies our understanding of the entire physical world. Issues of mechanism have to do with the interactions of concrete physical configurations, with time and dynamics, with temporal evolution, with impact and action, with whether it is possible for a device to trigger an action or respond to an input—with whether it can be impinged upon by an antecedent cause or have a consequent effect.[2]

Issues of meaning and mechanism are identified front and center by Turing himself. "The computable numbers," he says in the very first line of his classic paper, are "real numbers whose expressions as a decimal are calculable by finite means."[3] The

---

[1]Suppose the memory M of a digital computer C is divided into two parts: one (M_1) taken to contain the program, and the other (M_2) to contain data. If the contents of M_1 are held constant, as assumed in this interpretive division, and those of M_2 varied, C's behaviour will in general change. But if the contents of M_2 are held constant, and those of M_1 varied, C's behaviour is also liable to change. That the first constitutes running the same program on different data, and the second running a different program on the same data, are acts of external interpretation; nothing about the bit patterns per se warrants making such a distinction.

[2]Someone might suggest that a computer must be such as to be capable of "responding to a *signal*." That characterization combines both dimensions: *responding* is an issue of mechanical capacity; whether an impinging causal disturbance is a *signal* is a question of meaning or intentionality.

[3]Turing (1936/7); p. 230. By 'decimal,' as becomes clear in the course of his paper, Turing means numbers represented in any positional notation, not specifically those with a radix of ten. Issues of radix, binary vs decimal,

notion of *expression* implicates meaning and representation; that of *calculation*, mechanisms to achieve certain ends. Because numbers are abstract, they cannot be touched by mechanisms per se. What Turing machines demonstrate are mechanisms to manipulate concrete, effective representations or proxies of numbers—*numerals*, to be precise. On a classical view, that is, computers should be understood as *numeral* crunchers, not as number crunchers—though it has been a long time since the domains represented by configurations of computation-internal bits has been restricted to the arithmetical or mathematical.

Informally, the ends to which we put computational mechanisms are typically not formulated mechanically, but in terms of the entities that their symbols represent or mean: adding numbers, deciding problems, drawing inferences. This practice of classifying computational structures and phenomena indirectly via the interpretations of their symbols and states[4] is not restricted to describing their inputs and outputs, or to stating the goals they are devised to meet. We also routinely use such indirect classifications in describing their inner states and what they are doing overall. Without assuming a numeral-number relation, you could not say that computers so much as add or count, let alone calculate distances or solve equations. Without assuming a relation between configurations of bits and employees you could not say that a program had developed a schedule for part-time workers. Without adverting to (or at least assuming) representation you could not describe an internal procedure as calculating a square root, or as verifying a user's identity, or as translating English into Japanese.

Reference to such interpretations are crucial in order for programmers to know what software to build, to figure out

etc., are all facts about representations of numbers, not about numbers themselves.

[4]'Interpretation' in its logical and lay sense, meaning something like *reference* or *denotation*, not the sense that the term 'interpretation' has been given in computer science—e.g., in claims that Python or Java are interpreted languages.

how to build what they select, and to interpret their results. Programming, in fact, can be understood as an exploration in the space of meaningful or signifying mechanisms—a search for mechanically (physically) implementable designs that meet meaningful (semantic or intentional) goals. Sans semantic interpretation, all we could claim about computers is that they are machines that rearrange stupefyingly complex uninterpreted inner states, which produce and respond to bewilderingly complex physical flutterings at their boundaries.

The emphasis on the interplay of meaning and mechanism has permeated computer science from beginning. The devices we take to be the earliest examples of computing or "calculating devices"—abaci, Babbage's Analytical Engine, programming cards for Jacquard looms, etc.—are all understood under interpretation, in terms of what their mechanical arrangements mean. The brilliance in their design typically stems from the ingenuity embodied in the ways their mechanical configurations are constructed to allow them to meet those interpreted goals (that is: goals formulated in terms of the states of affairs that the states represent). Although he claimed to be focused on numbers and mathematical functions, Turing's originality lay in how he was able to devise mechanisms to manipulate representations of those numbers and functions—mechanisms with which to achieve goals framed in terms of the numbers and functions that those representations mean.

It was soon realized, after publication of Turing's paper, that computational "meaning" was not limited to relations between concrete states and abstract numbers—or perhaps, to construe the same point from a perspective that Turing might have preferred, that the meaning relation defined over computational states need not "stop" with numbers; those represented numbers may in turn be taken to represent arbitrarily more general domains. As Newell noted in his landmark "Physical Symbol Systems" paper, he and Simon emphasized this wider

interpretation throughout their work:[5]

> "[W]e insisted that computers were *symbol manipulation machines* and not just *number* [sic] *manipulation machines*. The mathematicians and engineers [in the 1950s] responsible for computers insisted that computers only processed numbers—that the great thing was that instructions could be translated into numbers. On the contrary, we argued, the great thing was that computers could take instructions and it was incidental, though useful, that they dealt with numbers. … [O]ur aim was to revise opinions about the computer."[6]

This was the insight that unleashed artificial intelligence and the computer revolution. Even if computers were originally described in mathematical terms, at heart they involve the effective or mechanical manipulation of meaningful symbols of unrestricted variety—symbols representing entities, problems, and phenomena in arbitrarily diverse domains.[7]

It follows that computer science—the study of computing—should encompass three things: (i) a study of mechanism, at least at the level of abstraction at which it is relevant to the fundamental meaning/mechanism dialectic; (ii) a study of meaning, or anyway an account of the sorts of meaning that are relevant to this dialectical interplay, perhaps including a story about what sorts of configurations of effective ingredients can serve as effective mechanical vehicles for such meanings; and (iii) a study of how meaning and mechanism interact, both in practice and in theory—i.e., a study of how mechanisms can be structured and led to behave in such a way as to support useful

---

[5]Newell (1980)

[6]Newell (1980), p137. Emphasis in the original. All remarks in this paragraph suggesting that computers process *numbers* can only be interpreted as meaning that they processes *numerals representing numbers*.

[7]While I applaud Newell and Simon's broadening of the conception of computing, in §7.4 I take exception to their characterization of meaning and semantics as referring to internal states of affairs.

meaningful interpretation.

Needless to say, all three parts of the story should apply to those meaningful mechanisms we call computers—or to the meaningful behaviour of computational processes, to characterize the situation more actively, or however the point should be formulated so as best to reveal the interplay of the two foundational notions.

This immediately raises a question.

If, on the one hand, the devices we know as computers are a proper subtype of a broader general category of meaningful mechanisms, a category that presumably includes at least people and animals[8]—or if computational activity is a proper subtype of active meaningful mechanical behaviour more generally, or, again, however one wants to put the point—then a theory of computing should say what that distinguished computational subtype is, and account for how it is to be distinguished from the general class. If computation is *special*, that is, then computer science should say *how* it is special, and explain the consequences of that "specialness." If, on the other hand, computers are not a distinguished subtype—if computers are *not* special, and the term 'computation' has simply come to be a name for what all meaningful mechanisms do, or 'computer' to be a name for all active meaningful mechanisms (or at least active meaningful mechanisms that people build)—then computer science should make that fact plain, and present itself as a theory of the structure, operations, and behaviours of meaningful mechanisms in general.

Unless computation can be shown to be special, that is, the account should be a theory of the structure, operations, and

---

[8]The point is not to deny, especially at the outset, what I call the "computational claim on mind": the thesis that people are computational. Rather, it is to recognize that that claim is viewed as a substantive thesis. The idea that we are meaningful mechanisms is trivial—obvious to anyone who is not a substance dualist. Humans manifestly mean; and by 'mechanism' I do not denote anything more specific than being physically embodied.

behaviours of all meaningful mechanisms *qua meaningful mechanisms*—i.e., a theory of meaningful mechanisms simpliciter.

What might such an account look like? Regularities underlying mechanism have to do with causal profiles and effective properties—in the computational case, with concrete configurations of computational devices and the operations they engender or undergo. Focusing on mechanisms and effective properties fits seamlessly into our contemporary intellectual worldview—our concern with causes and effects, physical bodies, with "how things work," as explored since the 17th century under the head of the age of mechanism and the development of the natural sciences.

Because it not only highlights the interpretation of physical configurations but also typically ignores differences between distinct physical configurations that support the same interpretation—i.e., because interpretively equivalent configurations can typically be constructed or assembled out of different materials—computer science's conception of "how things work" is typically more abstract (more "coarse-grained") than that pursued in the physical sciences. In most cases, computational analyses and individuation criteria abstract away from specific considerations of material, energy, heat, etc., often classifying the states and configurations of interest mathematically. Yet no matter how abstractly formulated, it is physical causation that ultimately grounds the difference between what can and what cannot be done, even if it is mathematically modeled.

Even if computer science examines effective properties through a mathematical lens, that is—even if it theorizes them as if they were themselves abstract—it is clear on analysis, and is foundational in the understanding of programmers, that the constraints on what can be done stem from the capacities and limitations of mechanisms *qua* concrete mechanical devices. It is this effective focus that gives computer science its claim on being a *science*.

This is why the fundamental theory underlying computer science is called a theory of *effective* computability. The

limitations theorized in the computability and complexity re-
sults that are central to our current theoretical conception of
computer science derive from what are ultimately mechanical
properties of computational substrates.[9] If one changes the
physics, or the encoding schemes by which the numbers and
functions are represented in the mechanisms, or otherwise
messes with the meaning-mechanism relation, it is straightfor-
ward to alter the mathematically-formulated computability
limits—claims about non-computability, about complexity
classes of mathematical relations, etc. (If all constraints on the
representation relation between concrete mechanical configu-
rations and abstract numbers are relaxed, for example, it be-
comes trivial to solve the halting problem.[10]) This manifest de-
pendence on substrate and encoding makes it evident that the
limits are not fundamental to the abstract mathematical realm,
but to how they are represented in concrete mechanical form.

As well as giving it scientific credibility, this mechanical fo-
cus has allowed computer science to restrict its attention to
proximal properties of computational processes. By concentrat-
ing of what is effective, that is, computer science has been able
to focus solely on matters "within the machine" or immediately
impinging on its boundaries—i.e., to theorize local properties
of effective structures and operations upon them. Metaphysi-
cally, this proximal focus stems from the spatiotemporal local-
ity restrictions of causal effectiveness.

I will argue that this restriction of theoretical focus to local
mechanical issues is ultimately untenable (insufficient for ex-
plaining computing as computing), but it has had obvious prac-
tical benefit. While, as we will see, mechanism alone cannot

---

[9]This can be shown even if the theory of computability is described en-
tirely in mathematical terms. The functions classically taken as primitive
(increment, equality, etc.) all have straightforward mechanical implemen-
tations, as Turing emphasized. Other functions could lead to other com-
putability and complexity results.
[10]See "Solving the Halting Problem, and Other Skullduggery in the
Foundations of Computing" (Smith, 2014).

explain what computation is, understanding the structure of the mechanical substrate is critical for knowing how to construct what we think of as computers and implement those processes we deem computational. While it leaves many theoretical questions unanswered, a focus on effective mechanism is a useful foil for those who want to build. It also explains why many computer science departments are in engineering schools.

The regularities underlying meaning are very different in kind. At least in general, meaning involves deferential relations to that which is *distal*—relations to situations and states of affairs at least potentially lying beyond the reach of the causal and the effective. Relations to abstract numbers, satellite trajectories, the organization of far-flung corporations, hypothetical situations, future and past events, and the like, are not *causal*, one might say, but I believe it is better and more precise to use the computational term and say they are *not effective*. Semantic relations to distal subject matters, including reference relations, cannot be blocked by putting up a physical barrier. If I were to launch a computational process modeling proton decay, simulating traffic problems in Bangalore, or calculating the first million digits of π, the process would still do these things even if it were placed in a lead vault, or run on Alpha Centauri.

Needless to say, it is metaphysically fortunate that the inaccessibility and even non-existence of a situation that a computation is about does not entail the ineffectiveness or non-existence of the computation itself. The non-effectiveness of reference in the philosophical sense (the sense in which the word 'Matterhorn' refers to a pyramidal mountain in the Alps) is utterly essential to anything we can imagine as computation. It is so fundamental to our understanding of both the human and the computational case—to the very notion of intentionality, to all intentional phenomena, including understanding itself, and to the fundamental character of "aboutness"—that it is impossible to imagine its not being true. If reference were restricted to being an effective relation, the world would disappear,

reasoning become useless, hypotheticals impossible, counterfactual reasoning inconceivable, and fantasy lives metaphysically banned.

Some will argue that meaning is at least partially constituted by effective relationships and causal behaviours in which the effectively implemented meaningful structures play a causal role. This intuition permeates what is called embodied cognitive science, it is a theme in various pragmatist ideas about human thought and language (such as that we "do things with words"), and it is hugely popular throughout contemporary humanities and social sciences—e.g., in so-called "new materialism," in theories of radical embodiment, and the like. The intuition also meshes with widespread contemporary endorsement of naturalistic theorizing. In all these theoretical efforts it has become *de rigueur* for theorists to emphasize the critical relevance of effective material properties, physical bodies, causal interactions, and constitutive materialities of intentional phenomena. In parallel—though for at least superficially different reasons—focusing on the causal behaviour and the effective character of meaningful structures will resonate with many computer scientists.

Yet no matter how constructive or embodied one's view, no matter how pragmatist one's epistemological predilections, no matter how much such "embodiment" intuitions hold true, no matter how materialist or concrete one's metaphysical sensibilities, many of intentionality's most fundamental properties, including not only truth but reference as well, can never be entirely constituted locally or effectively.[11] On the contrary, they paradigmatically reach beyond the limits of the effective. This is an ultimate and irrevocable fact—even if it does not occupy a prominent position at the forefront of present-day theoretical

---

[11]Even in mathematics, Gödel's second incompleteness result, the failure of Hilbert's formalism project, and the concomitant recognition that "semantics cannot ultimately be reduced to syntax," reflects this intrinsic dependence of mathematical expression on facts beyond the reach of the effective properties of their expression. See §6.4.

imagination.

No accumulation of embodied, causal, physicalist, or materialist intuitions, that is, can undermine the fact that meaning and intentionality are constitutively oriented (directed) *beyond* the effective—beyond what is physically or immediately proximate. We would not even know about the world, if that were not true. Not only would we not be able to get home, once home disappeared over the horizon; we would not be able to think about home—would not be able to think about or have a notion of home, or indeed think about anything else—if the reach of meaning and aboutness was foreshortened to the boundaries of effect. In fact I would even go so far as to say that "reaching beyond the effective horizon" is intentionality's *point*.

2 Deference

Why call reference relations *deferential?* Because the norms that govern the effective states and mechanical operations of the devices we build (norms of correctness, e.g.) are determined at least in part and often wholly by facts about and relations to and among those distal situations—by facts about the situations towards which they are intentionally directed. If I write an algorithm to enumerate prime numbers—if, that is, I devise a strategy to cause a machine to produce a list of decimal numerals representing prime numbers—and my program prints out «38»,[12] then it has made a mistake. *That* it is a mistake, however, cannot be determined by close inspection of the sequential characters «3» and «8». That «38» denotes thirty-eight, for starters, depends on our assumption that we are using a base-ten representation scheme; if we were using base-eleven, «38» would denote forty-one, which *is* prime. Plus, even assuming the standard base-ten representation, the fact that the number thirty-eight is not prime, and therefore that the computer

---

[12]Because both single and double quotation marks are elements of many computer languages, including the Lisp dialects to be discussed in this book, I will use guillemots ("French quotes"), as here, to demarcate, and thereby quote, computational expressions referred to within English text.

has erred, is a fact about the distal, abstract number that the two-character numeral «38» represents, or is anyway taken to represent. And since the stated goal was to "enumerate prime *numbers*," it is an abstract fact about that non-proximal number thirty-eight that matters—the abstract, non-effective fact about primality that establishes the norm that my algorithm is mandated to honour.

Deference is a fundamental norm underlying all thinking—including all of science. If a theory parts company with the facts, then the theory is wrong, and needs to be rejected or repaired. More generally, *right* and *wrong, true* and *false, correct* and *incorrect,* and the like all have to do with whether effective states and operations correspond appropriately to that which they mean or that to which they refer—i.e., with that towards which they are intentionally directed. As I say, this is not to reject pragmatism, or the Wittgensteinian maxim that meaning is use, or any other view that takes meaning to *include* material activity, and/or to depend on facts about situations and states of affairs in addition to those towards which they are primarily directed. Those ancillary situations and activities, too, *qua* being intentional, typically reach beyond the local and proximal "bumping and shoving" of pure mechanism. (Wittgenstein was no solipsist, physical reductionist, or any other form of psychological internalist.[13])

The moral of "deference to that which is distal" holds just as true in computational contexts as in any other.

I call these relations normative because, in the context of use, the roles played by the relata are of unequal value or worth.[14] *True, right,* and *correct* are good properties to have; we

---

[13]Wittgenstein would likely also be appalled at the idea that the meaning of words could be modelled, as in many contemporary AI systems, by the frequencies and associations with other words. The language game of brick-layers involves bricks, mortar, hods, and trowels, not merely the words 'brick,' 'mortar,' 'hod,' and 'trowel.'

[14]'Value' and 'worth' are normative concepts; I am not attempting to define normativity in non-normative terms, which is likely impossible.

strive for them, and work to build computers that achieve them. *False*, *wrong*, and *incorrect* are bad; we strive to avoid them, and design our machines to avoid them as well. Again, this is not because there is something intrinsically right or wrong, per se, with the mechanical configuration of the machines (or our brains) that end up in such states. Rather, the entities and states of affairs towards which those states are directed "hold the cards" with respect to the norms that govern our machines and our brains. If they are not as they are represented as being, then the machine or brain states in question are censurable. Even in that narrow class of cases when it might seem as if all applicable norms could be locally and effectively defined—"arrange a group of unary numerals in order of increasing length," for example—the fact that it is someone's goal for the numerals to be in that order will in general rest on distal (non-local) facts and considerations. It is not an intrinsic, machine-internal fact about the list of unary numerals, even after they are arranged, that being so arranged was their or anyone else's goal, or that, per se, being so ordered is good. The goal might have been to arrange them in order of decreasing length, in which case the machine would again have failed, in spite of being in a mechanically indistinguishable state.

Because they are constituted by distal relations, the non-effective aspects of meaning cannot play an immediate causal role in how computations proceed—cannot exert a direct influence within the local space-time envelope of the mechanical goings-on.[15] This is the deep metaphysical truth underlying the widespread idea that logical inference must be "formal," on at least one reading of that problematic term. Yet, to state what is perhaps the most fundamental tenet on which this book is based:

Certainly I disagree with attempts to characterize it in terms of evolutionary fitness and adaptability, as has become common in theoretical biology. If anything, I believe it requires adversion to objectivity and a conception of "world as world," as indicated in *The Promise of Artificial Intelligence*.

[15]As argued in §6.2, formal or syntactic properties are constrained to be effective.

Even though it cannot make an immediate effective difference, meaning is nevertheless essential to computing as computing.

With nothing representing anything—nothing normatively governing its operations, nothing determining whether or not what it is doing makes sense—our devices would be as far from *computing* as random splatterings of ink on paper are far from *language*, are far from *meaning anything*. Moreover, as we have already seen, meaning even establishes "what has happened" and "what has been computed," when those "what" phrases are understood, as a philosopher would put it, "under interpretation"[16]—in terms of what it is that the concretely manipulated structures mean or refer to or represent. As I keep saying, to claim that a computational procedure has *added numbers, figured something out, made a decision, calculated an optimal route, obeyed an instruction, determined a likely outcome*, etc., is to describe that computation through such an intentional lens.

Issues of *meaning*, in sum, have to do with the relations of effective arrangements of marks and mechanical parts to distal subject matters or task domains—to what they are about, to why they matter, to what they signify. Issues of *mechanism*, in contrast, underlie what is immediately and proximally accessible, what can be concretely done, how it needs to be arranged in order to do it, how hard it is to accomplish a given result. Because of the locality of physical space-time (at least at any scale relevant to contemporary computing) issues of mechanism tend to reach "inwards," or at least "nearby," to the nature of and relations among the effective ingredients of the computational process itself, and to surface perturbations or causal couplings of the system to its immediately liminal environment. Issues of meaning, in contrast, in general[17] reach "outwards," into the

---

[16]'Under interpretation' again in the philosophical and traditional semantic sense of interpretation.

[17]Many of the reference relations in reflective systems do not reach outwards (even if the norms governing them do)—one reason reflection is an ideal site for studying computation as computation within what would

world or context in which the computation is relevant, unlimited by that which is immediately causally accessible.

Mechanism stays local, to put this cryptically, determining what can be done. Meaning reaches widely, governing why we care. It is only because of their interplay that computation is useful, powerful, and important—only because of their interplay that computation is *computing*.

3 Computer Science

To date, the focus of theoretical computer science has been skewed. Computational theory—computer science as explicitly formulated—has focused almost exclusively on the mechanical side of the dialectic, on what is local, on what is effective, on what configurations it is possible to construct a device to causally manifest. Responsibility for meaning has been relegated to the sidelines, shouldered by the implicit practices and presuppositions of theoreticians and programmers—has been assumed, that is, and presumed to be unproblematic. Mechanism has been on center stage; meaning banished to the wings.

That is not to say or imply that meaning is either invisible or unimportant. It is utterly standard to speak of computations as adding numbers, as making decisions, as calculating routes, etc.—phrasings that betray the ever-presence of meaningful interpretation. But as regards direct theoretical attention, meaning relations play only a supportive role in theoretical analysis—crucial, but outside the spotlight.

The depth of this asymmetry, and the extent of the confusions and infelicities to which it has led, have been difficult to appreciate, because of three intricately correlated assumptions and practices. Some readers will immediately rebel against the claim, arguing against the singular importance of the mechanical by citing, as evidence contrary to what I am arguing here, object-oriented languages, modeling languages, purely mathematical formulations of computability and complexity theory,

---

appear to be a traditional computational context.

and a variety of other features of contemporary practice. Rather than undermining the point, however, I will argue that these considerations merely show how pervasive has been the influence of these three ultimately distracting assumptions. Disentangling them, documenting the resultant confusions, and clearing the way for a more adequate account, are among the aims of this book.

3a Mathematical Vocabulary

One source of misunderstanding has already become apparent: computer science's near-ubiquitous use of mathematical vocabulary. It is not unusual for computing, computability, and complexity to be described in wholly mathematical terms, both in introductory treatments and in advanced treatises, with no explicit reference to mechanisms at all, or to encodings, representation, or anything else concrete.[18]

Like so many other things, this mathematical focus was evident in Turing's original 1936/7 paper. Ostensibly, the paper's subject matter was the computation of *numbers*; specific

---

[18]An example is the definition of the term "effective computability," commonly presented in university-level Theory of Computation courses.

> Let $\mathbf{N} = \{0,1,\ldots\}$, $\mathbf{N}_k = \mathbf{N} \times \ldots \times \mathbf{N}$ (with k factors)
> A partial function
> $\quad f : \mathbf{N}_k \rightarrow \mathbf{N}$
> is effectively computable if there is an effective procedure or algorithm that correctly calculates f. An effective procedure is one that meets the following specifications. Firstly, the procedure must consist of a finite set of "simple" instructions and there must be no ambiguity concerning the order in which the instructions are to be carried out. Secondly, if the procedure is given a k-tuple $\mathbf{x}$ in the domain of f, then after a finite number of steps, the calculation must terminate and output $f(\mathbf{x})$; if the procedure is given a k-tuple not in the domain of f it must not output a value. See also Church-Turing thesis. (Daintith and Wright, 2008).

Note that this explanation rests on the undefined terms "simple" and "carried out".

machines were modeled with mathematical (abstract) quintuples; and so on. The property of being 'computable' is officially predicated only of abstract entities: numbers in the first instance, and subsequently functions. All the considerations about numbers adduced throughout the paper, however, and about the functions defined over them, have to do not with numbers themselves, but with numerals—i.e., with their concrete representations: with what can be done locally, with what is and what is not directly comparable in a sensory and/or effective way, with what is finite and what can be done by finite means. Turing is explicit about this: "According to my definition, a number is computable *if its decimal* [i.e., if a standard-form numeral representing it] *can be written down by a machine*."[19] Think too of his invocation of intuitions about the resources available to a "human computer": things visible, things close at hand, things that can be stored in a finite and limited memory, and the like. Consider as well the strategy invoked to define the universal machine (and language "interpretation," in the programmers' sense, at the same time): of *writing the quadruples down on the machine tape*, which of course means writing down representations of those quadruples. Yet although the mechanical properties of numerals are doing all the work, it is numbers that predominate in the attendant imaginary.

3b Classification

A second complicating issue is also evident in Turing's paper, keeping the asymmetry of current computational theory at bay. I said above that we use the labels 'o' and '1' to classify computational states, using a phrasing that suggests that what we use as classificatory devices are numerals. As far as it goes, that is correct; think of internet addresses such as «192.168.1.254»—labels that do not denote numbers at all, and that play their individuating role as concrete structures.

     We also use the numbers themselves, however (that is, the

---

[19]Turing (1937).

numbers that the numerals denote), including the numbers zero and one,[20] to *classify* computational states—or, as might be said, to *model* them. Classification in this context, to simplify egregiously, is an epistemic practice we theoreticians engage in—using objects useful for our theoretical understanding and referents of terms in our linguistically framed theories to model or correspond to the entities in the domain that the theory is ultimately about, even though those objects are not themselves what we are talking about. Thus we denote the length of a board by saying that it is "4 feet 7⅜ inches long," where 'feet' and 'inch' are classificatory devices we use to characterize lengths; the length in question would be the same identical length, within the applicable margin of error, if we denoted it by describing its extent as 1 meter, 40.65 cm. "The length doesn't know from feet and inches," one might say; the length is what it is, independent of such classifying externalities.

This implies that there are two ways in which concrete computational states are related to abstract numbers: one in which *states represent numbers*, as for example when computers are taken to add or perform calculations; a second in which the representation relation runs the other way, with numbers being used to represent states, as for example in the suggestion of using 255 as a mask to block or reveal the lowest eight bits of a memory location.

Among other differences, the two cases involve distinct directions of deference. If addition is at issue, then we hold the bit patterns accountable to representing the sum. If we add three and five, using the numerals «3» and «5»—if, that is, we issue (what we interpret as) an instruction leading the machine to do what we call "adding" to the binary representations of the numbers three and five, and assume that the machine should as a result produce a representation of the sum of the two

---

[20]To minimize confusion, I will write out the names of abstract numbers in English, in contexts when the use of decimal numerals ('1', '2', etc.) might confuse matters as to whether numbers or numerals are at issue.

numbers that those two numerals represent—and the computer prints out «7» (i.e., the numeral representing the number seven), then the machine has erred, or the program has a bug. The computer or program is normatively enjoined to return «8» (the numeral representing the number eight). If on the other hand we recommend or instruct the computer to use the numeral representing the number eight to mask the lowest three bits of a binary storage location, then it is *we* who have erred; the operative norms in this case are on the binary bit pattern, not on the abstract number that we use to classify that pattern.

Though fallible, understanding the direction of deference can be a useful guide to identify the direction of representation. In some cases both directions of representation are combined, complicating discernment of the underlying semantic situation. One hears talk, for example, about "adding two eight-bit numbers"—a phrase which on the surface is manifestly oxymoronic. Numbers do not have numbers of bits; "eight-bit" is a property of representations. Addition, however, is not *au fond* an operation on representations; strictly speaking, it is defined only over numbers. The idea of an "eight-bit number" thus relies on a derivative extension of bits to numbers and of addition to numerals, via an assumed numeral-number representation relation.[21]

3c Redefinition

More serious than the ubiquitous use of mathematical language and the collapsing of numeral-number relations is a third complication standing in the way of our proper understanding of computing. In an unremarked development that disturbs the foundations of science, the vocabulary and terms of art that have historically been used to talk about meaning and intentionality have been systematically redefined, in computer science, to refer to *effective properties of mechanisms*—not to the

---

[21]Note that in such constructions (such as "eight-bit number") the classifier is a *number* of bits, not a *numeral* of bits—which would make no sense.

distal situations, or to relations to those distal situations, to which they would classically have been taken to refer. That is, such ur-semantic notions as *reference*, *value*, *semantics*, *interpretation*, etc.—even *meaning* itself—have been reconstrued in computational discourse to refer (in its classical sense) to properties and relations within the proximal, effective realm of the computation qua mechanism. This terminological transformation, symptomatic of a historic mechanization of the topics of intellectual discourse more widely, has complicated the understanding of the discourse of computer science by surrounding disciplines. Suppose a computer scientist writes or talks about "the semantics of program α," "the meaning of computational configuration β," or "the referent of program identifier γ." It would be unexceptional for an outsider to misunderstand such phrases as being about deferential, non-effective relations to distal or abstract task domains—numbers, functions, people, sets, employees, satellite trajectories, organizations, future events, etc. In contemporary computational discursive contexts, however, *the meanings of all these terms have been folded back inside the machine*—used to refer to the effective consequences of running the program or operating with that structure (even if the structure is mathematically modeled).

It is as if computer science has wrapped all its devices in inwards-facing mirrors, which reflect all traditional semantical reference back inside the machine. In a program to determine raises, for example, the official "semantic value" of an identifier like «EMPLOYEE-153» would be taken, in computer science, to be a (potentially abstract model of a) *memory record*, not a living and breathing employee. Similarly, if «$USER» were a computational identifier for the currently logged-in user, its reference would likely be a concrete login identifier, not the flesh-and-blood person to whom, as outsiders, we would take that identifier to refer. What is considered to be the *denotational semantics*—not just the "operational consequences" or "behavioural import"—of an instruction to compare the identities of the university having the most students with that having the highest-paid basketball coach will be to return "true" (i.e., the

computational identifier «TRUE») if two *internal identifiers* are the same, not if the two institutions are in fact the same.

Suppose Michigan State and Montana State were accidentally assigned the same label «MSU», leading a computer to return «TRUE» to a question about whether two students attend the same university,[22] when in fact the correct response would have been «FALSE». The point is not that this might not happen; such mistakes are legion (and unavoidable). What is problematic is that the «TRUE» response would be deemed *correct*, according to computer science's reigning analyses of what is called the *semantics* of programming languages (not just its account of their *operation*). What computer science takes to be the standard semantical account, that is, "reaches" only as far as what is internal to the machine; the question of what universities those identifiers actually name is not only considered to be extra-theoretical, but, surprisingly, to be beyond the reach of semantical analysis.

It is left to the programmers, or to the users of these systems, or to theorists' tacit understanding, to realize that «EM-PLOYEE-153» *also* signifies or denotes or in some other way represents a person, that «INSTITUTION-2714» also signifies or represents a university, and that the instruction will return «FALSE» if the identifiers «INSTITUTION-2714» and «INSTITUTION-699» are distinct, without reference to the people or institutions involved. Or, to invert the example, if one university were to end up identified by two distinct labels, it would again be left to programmers, users, etc., to understand that the system would return «FALSE», and would be deemed "semantically correct" in doing so, even if the correct answer would have been «TRUE». By the same token, it is left to designers, programmers, and users to understand whether "adding" «10,000» to a financial

---

[22]We take the question to be about students. The idea that the program would take it to be a question at all is, of course, also a projection that we make on it. Even assuming that interpretation, the program would treat it as an issue of the respective identities of the labels associated with the universities that are tied to the labels associated with the students.

record merely represents an update to a computational record, or signals that a live human being has being given a raise. And of course the fact that «TRUE» and «FALSE» signify truth and falsity is similarly atheoretical (to say nothing of being non-effective), on current semantical accounts, relegated to programmer or theorist understanding.

How could a machine do something different? Or perhaps more pointedly: how could computer science do something different? Computer science could take a lead from logic, and, so long as the program is operating as expected, warrant the computational behaviour not to be correct, but to be *sound*—to follow from the assumptions, *including the semantic assumptions*, on which it is based. It could provide facilities for the definition of constants («CICERO», «DUKE-UNIVERSITY», «USER-271», etc.) which it recognized as having referents in the real-world task domain, without assuming that the identity of the symbol was necessarily a reliable proxy for the identity of that to which the symbol referred. Or if the programmer or user were explicitly prepared to warrant that symbol identity could be taken to match referent identity—i.e., could affirm a one-to-one correspondence between computational identifiers and real-world entities they denote (called a "closed world" assumption in AI)—that fact could be explicitly stated, and pointed to in demonstrations of how programs meet their semantic goals.

It may be pointed out that even if these issues about the semantic reach of computational structures into task domains remain theoretically untreated, computer science has nevertheless triumphed in its first fifty years. It may be argued, that is, that even if it is left to programmers to shoulder tacit understanding of these issues, they are clearly capable of doing so adequately. But masons extoled the building of St Mark's in Venice, too, hundreds of years before the development of Newtonian mechanics. The claim of an eleventh-century engineer that they had no need for a theory of forces, torques, mechanical advantage, etc., would ring hollow today. It is also widely recognized that programming is becoming increasingly bewildering; any help we can provide programmers to steward these issues

will be beneficial. And think of such famous bugs as the one
that bedeviled the Hubble Telescope when it was first launched
because two numerical values were referenced to schemes for-
mulated with different units.

In addition, we are not going to be able to rely on the tacit
understanding of human programmers and human stewards of
programs' semantic assumptions in the future. As software de-
velopment is increasingly delegated to automated algorithmic
development—including "software 2.0," as automation
schemes based on machine learning are being called—we may
increasingly need to depend on explicit representations of the
computation-to-world correspondences.

4 Project

My aim in this book is to disentangle the complexities de-
scribed above, in search of a clearer theoretical framing—a task
made more difficult than it might otherwise have been because
of the reconfigured vocabulary issues discussed in §3c.

Because computer science has redefined the terms that phi-
losophers, linguists, and cognitive scientists have used for hun-
dreds of years to talk about the deferential, distal, non-effective
aspects of intentional meaning, it is difficult to write about the
issues in a way that is comprehensible to both audiences. It is
especially difficult to explain classical conceptions of semantics,
including issues about distal reference, non-effective relations,
and governing norms to computer scientists, since all the terms
that were classically used to name such phenomena ('reference',
'meaning,' 'truth,' etc.) have been "used up" in the new compu-
tational jargon for internal phenomena, and as a result the is-
sues classically considered to be semantic have disappeared.

Inevitably, it follows that all statements made here are vul-
nerable to being differentially interpreted by diverse audi-
ences—even those to which the book is primarily addressed.
For starters, the notion of semantics is likely to be given an ex-
ternally-directed, non-effective, non-temporal reading by phi-
losophers, and a computationally *internal*, effective, processual
reading by computationalists, with cognitive scientists and

linguists potentially spread out across a variety of intermediate views. I will do my best to avoid confusions stemming from such terminological misalignment, and within practicable limits will erect signposts to signal what is going on, but it is an issue with which the reader will inevitably have to contend.

Partly to mitigate such difficulties, the investigation will start internally, with problems recognizable as problems within computer science, and slowly build towards more externally reaching and widely encompassing frameworks. But throughout, the bottom line is straightforward. Although computation involves an interplay of meaning and mechanism, contemporary discourse about computation, as embodied in present-day computer science, has projected the realm of meaning onto the effective wall of the cave. All that can be seen on that wall is a shadow of the intentional richness that is in fact constitutive of computing as computing. We need a thorough understanding of the whole phenomenon, not merely of its projection onto a mechanical wall, if we are to understand computation and move computer science forward.

Meaning and mechanism are the primary characters in this drama. Their interplay is the primary subject matter. The powers and limitations of the effective are a major theme. The adequacy of science for explaining intentional phenomena, including computing, are the stakes.

What we need is an investigative strategy.

B — Strategy

The book is written in the form of a detective story. It starts by examining the evidence alluded to at the outset: telling issues in our default conception of computing that came to the fore in the 1970s, during a series of experiments exploring computational reflection. The discussion will include not only the design of 3Lisp, the world's first "reflective" programming language, but also the subsequent failure to develop 4Lisp, 3Lisp's

envisaged successor.

5 Reflection

A reflective system is one that is able to represent, reason about, and act appropriately in regard to its own operations, structures, and behaviour. Reflection therefore intrinsically involves both meaning and mechanism. "Represent and reason about" are intentional capacities; "act appropriately in regard to" involves effective operation. While all computational systems exemplify both dimensions, reflective systems are distinctive in the fact that the dialectic applies twice. Not only must a reflective system *itself* be meaningful and mechanically effective; it must also represent itself *as* being meaningful and mechanically effective.[23] Meaning and mechanism are constitutive parts of both the system as signifying and the system as signified.

More specifically, a reflective system $\mathcal{R}$ must both represent (describe, reason about) itself, and be able to act effectively and appropriately in regard to itself, in terms of a representational schema or model $M_{\mathcal{R}}$—something I will call the system's **reflective model**. $\mathcal{R}$ must both *instantiate $M_{\mathcal{R}}$* and *represent itself as instantiating $M_{\mathcal{R}}$.* That self-referential aspect of reflection, coupled with the dual-aspect nature of computation, generate a four-part criterion on reflective systems—something which as a whole I call the **reflective integrity criterion**: a reflective computational system must be able to

1. Represent its ability to represent;
2. Represent its ability to act;
3. Act on its ability to represent; and

---

[23]Because the notion of reflection introduced with the publication of 3Lisp (Smith, 1982) was not understood (as documented here in §2.2), the term 'reflection' in modern programming languages has lost its intentional character—i.e., its involvement of issues of meaning and reference, as discussed here. As I argue throughout the book, however, I believe that this practice is a follow-on consequence of misunderstanding, and that considerations of meaning nevertheless play an implicit if unacknowledged role in programmers' understanding of the notion.

4. Act on its ability to act.

Moreover, it must do all of these things "correctly."[24]

It is these interlocked strictures that make reflection an ideal site to analyze both dimensions of the dialectical meaning/mechanism conception of computing. It is because both aspects play such central roles that the book starts out looking at reflection.

I attempted to meet the reflective integrity criterion in stages. For expository simplicity, I used '1Lisp' as a blanket label for all prior Lisps—from McCarthy's Lisp 1.5 through Sussman and Steele's Scheme, a higher-order version of Lisp that attained prominence in the 1970s and 1980s.[25] The next step, called 2Lisp, which met criteria 1–2, was designed to manifest the two dimensions of computing explicitly—both meaning and mechanism. Though not itself reflective, the fact that 2Lisp was based on such a model made it "reflection-ready." The design of 2Lisp allowed the subsequent definition of 3Lisp to be relatively straightforward. In fact 3Lisp is conceptually simpler than 2Lisp (though trickier to implement).

Although 3Lisp in particular, and the idea of reflection in general, was acknowledged in the literature, I believe it is fair to say that 3Lisp was never fully understood. It was not 3Lisp's reflective character that was opaque, however—at least not in the first instance. Rather, it was the underlying two-factor "meaning/mechanism" model of computation, exemplified in the 2Lisp dialect, that proved theoretically inscrutable.

This book is not a study of computational reflection.

---

[24]I.e., it must represent its representational capacities in a way that satisfies the conditions that it represents those capacities as meeting. If it represents its own representational capacities as context-independent and static, then it must represent itself as such, and so on.

[25]In retrospect I think I should have used the label '2Lisp' for Scheme, to make the higher-order version a distinct second step in the sequence. What ended up being called 2Lisp, 3Lisp, and 4Lisp would then have been labeled 3Lisp, 4Lisp, and 5Lisp.

Rather, reflection serves merely as a crucible in which to subject our underlying models of computation and representation to merciless scrutiny, in order to reveal lurking inconsistencies and conceptual ambiguities. The book is an (English-language, discursive) reflection on the fundamental nature of computation, and of the interplay of meaning and mechanism. It explores these issues first in the context of computational reflection, then more generally across computing and computer science, and ultimately in the full range of intentional systems.

6 Difficulties

What were the difficulties? What troubles have lurked under the hood of computation, only to be exposed by reflection and the design of 3Lisp? And why was the design of 4Lisp never completed, intended as the next step in the series?

Evidence of trouble has already surfaced, discernable in the history of computer science's technical vocabulary. The most basic terms of the field—*symbol, data, meaning, value, interpretation, information,* etc.—were inherited from 17th century discourses about logic and reasoning. These terms are not like *mass, force, energy, charge, momentum,* etc.—properties manifestly applicable to moving bodies, physical forces, processes, energetic effects, and such. That is: the origins of computer science's most fundamental terms did not historically have to do with mechanical concerns. Rather than stemming from the "empiricist" side of the Cartesian divide, it might be said, the terms derive from the "rationalist" side, having to do with the sorts of intentional entities studied in logic: expressions and processes that are *about* things—entities "intentionally directed" towards entities and phenomena (objects, processes, states of affairs, etc.) that they are about, entities and phenomena in a subject matter or task domain.

On the surface, this origin story does not seem problematic. We have already seen that computational structures are intentionally directed: they represent numbers, mathematical relations, elements of task domains (employees, salaries, corporations, classroom usage, etc.), other items in the computational

realm itself (web pages, code bases, memory locations, data base entries, etc.), and instructions to perform this or that operation. As soon as one presses, though, it becomes clear that complexities lie just below the surface.

An almost trivially simple example will serve as a "smoking gun" in subsequent analysis. In all classic dialects of Lisp, «3» is taken to be the value both of «(QUOTE 3)»[26], which makes sense, since quoting an expression is a classical way to name or represent it, and also of «3» itself. The item «3», that is, is said to be "self-evaluating."

Per se, «3»'s being self-evaluating is not unintelligible. Nor is it unintelligible that, given «3» as input, the Lisp interpreter[27] returns «3» as output—i.e., returns as output the very structure it is given as input. What else, after all, should it be expected to do? What is problematic is calling «3» its own *value*. That phrasing suggests that "evaluation" is being used for something other than 'semantic value' in the traditional intentional or logical sense (of 'reference'). To speak classically:

1. If the computational entity «3» is taken to be a *numeral* or *symbol* (e.g., on a "formal symbol manipulation" construal of computing), then «3»'s semantic value is surely the abstract number three—not something that a language processor can return, in fact not a "computation-internal" entity at all.

---

[26]«QUOTE» is typically used in Lisp-like languages to create list data without evaluation. For example, «(+ 1 2)» evaluates to «3», whereas «(QUOTE (+ 1 2))» (or «'(+ 1 2)») evaluates to a list with three elements, the first of which is the atom «+». The need/use for quotation is due to the fact that the same data structure, a list, is used for both code and data. There is not a clear correlate in most other programming languages (with the notable exception of machine code, which uses byte sequences for both code and data). In Python, for example, one can create a string of an expression «"1 + 2"» – and even use special-purpose methods like «eval» to attempt to evaluate it as Python code – but there is always a clear distinction between code and data structures. See §2.4 for a discussion of why a Lisp-like language was chosen for the study of semantics and reflection.

[27]More on computer science's use of the term 'interpreter' in §3.4, below.

2. If, alternatively, «3» is taken to be a *number*, on the view that computation is itself abstract, then a number does not seem like something that should have a value at all. There is no problem, on an abstract conception, with numbers *being* values. But abstract numbers *having* values seems awry.

In other words, the term 'value' is being used for two different things: (i) the structure that the language processor *returns*, given an input, and (ii) the entity that the input *signifies* or *designates*. Whereas the former must be computational, the latter need not. The entities, and the original structure's relations to them, are different. Clarity demands that they be labeled with different terms.

3. Someone might suggest a third possibility: that when we describe a computationally-internal element using the term '«3»'[28] we are *indirectly* referring to a computational structure—to something concrete, something with effective properties, something that can play a mechanical role—by classifying it in terms of the abstract entity that would classically be called its semantic value. This fits with the practice of saying that an algorithm has "figured out the shortest route to Albuquerque," or that a decision-theoretic algorithm has "determined that Alex should be given the job."

But in terms of the current issue this third suggestion is problematic. According to it, "the value of «3»" should then refer to *the value of that which we are classifying with its value*—which, if not oxymoronic, is either empty or ambiguous. Any English term denoting that value would *also*, by stipulation, denote (via indirect classification) the computational structure of which it is the value. If we use a structure's value as a way to classify it, that is, we are left

---

[28]The five-character '«3»' expression is a English-language reference to a *single* character token of the Lisp numeral 3. (See note 11 on the use of guillemots ('«' and '»') in §1.2.)

with no way to talk about the relation between that structure and its value.

If the structure «3» has anything to do with the number three, in other words, then none of these three options license the claim that «3» is *self-evaluating*.

To complain about «3» and «(QUOTE 3)» both evaluating to «3» may seem petty and fastidious—nothing over which to make heavy weather. So thought I when I built 2Lisp, designed in terms of what seemed an "easy fix" to the problem. But the issues ramify. What seemed trivial, on the surface, proved to be the tip of a large conceptual iceberg.

This example simultaneously illustrates two common practices in contemporary computational theory:

1. Computational structures are often conflated or identified with what they mean, represent, or denote, in virtue of either (i) being directly identified with them (treated as the same thing), or (ii) having the relation between them left implicit, out of theoretical view.

2. When the phrase 'the semantics of' (or 'the semantic value of') is used explicitly, it invariably functions to name a computationally internal structure or behaviour.[29] It may name a structure directly, in which case that structure will be an effective structure of the sort that can be returned as the output of a language processing regimen—e.g., as when the semantics of «(QUOTE A)» is said to be the atom «A». Or it may name the structure indirectly, via an intermediating abstract classifying entity, such as a number, function, or truth-value.

Return to the example of the numeral «3». What «3» means, represents, or denotes is unarguably the number three—the

---

[29]This is too simple; the 'semantics' may include side-effects or other results of processing, for example. The domain of semantics, however, even including such activity, is always computation internal or peripheral—not something that reaches out into the distal world.

abstract number with which in informal practice (and in programmers' minds) it is invariably identified. How can we be sure? Because «3+2» will invariably return «5», and the only rationale for that behavior is because the number five is the sum of the numbers two and three. Informally, one might think "the value of «3»" or "the value of «5»" would be English terms naming the respective numbers. But that analysis does not withstand scrutiny. It is belied by the claim that "«3» is self-evaluating"—a statement that betrays the fact that the abstract number is being used to classify something concrete.

Even if such conflationary practices seemed innocent in Turing's time, perhaps because of the clear mathematical structure of the domains with which he was concerned, they are problematic for a more general account of computing. In three distinct ways they effectively "disappear" various properties, phenomena, and vocabulary items that we need to understand if we are to plumb the depths of computation:

1. Ontologically, identifying a structure with what it represents or means, without theorizing the representation or meaning relationship explicitly, hides the entire issue of meaning and representation from theoretical view—the first of the two axes constituting computation as computation. Differences between representation and represented are minimized, if noticed at all—including, in particular cases, issues of cardinality, identity, contextual sensitivity, and the like.[30] These intentional complexities are critical to the proper functioning of real-world computational systems. Hiding them from theoretical view prevents our understanding some of the most critical issues in contemporary computing (including, for example,

---

[30]E.g., whether symbol identity can be used as a proxy for identity in the task domain, whether the number of entries in a data base can be assumed to be the same as the number of entities in the domain that the data base represents, whether the fact that an internal identifier has not changed, such as "disk in drive 0" or «$USER», is evidence that the item that this identifier names has not changed, etc.

issues of ethics, bias, coverage, etc.).

2. Epistemically, using the word 'semantics' and 'value' for what is returned by language processors, and for other effective relations and operations among computationally internal structures, also robs us of the theoretical vocabulary necessary for understanding the meaning dimension of computing, when it does come into view—for understanding how computing systems are related to the task domains or worlds they are about, and how they are more than merely uninterpreted state machines.[31]

Ironically, moreover, as well as obscuring the *meaning* dimension of computing, conflating or identifying representations with what they represent also hides from view the second of computation's constitutive dimensions: the notion of *effectiveness*. Computational structures play a role in computational activity in virtue of exemplifying effective properties. Yet it is often true that the entities that those structures mean or represent are not themselves effective. The case of numerals (effective) representing numbers (not-effective) is a case in point. If representation and represented are not rigorously distinguished, then not only does the relation between what means and what is meant get hidden, but the role of effectiveness also eludes theoretical view.

The problem to which this leads is not that computer science has ignored efficacy; rather, it is that it has too blithely assumed it. Current computational theory seems to assume that everything of interest is effective, even if it is mathematically modeled. While this would take work to document carefully, I

---

[31]Another example. Logicians, philosophers, and cognitive scientists might say that when we understand the computational structure «3» as "the number three," we are understanding it *under interpretation*. As noted above, that is not the notion of interpretation from computer science. In computational discourse, the interpretation of any computational structure must be another computational structure (or effective operation on such structures), nothing genuinely abstract. So the term 'interpretation,' along with 'semantics' and 'value,' needs reconstruction.

believe that computer science is in the grip of something I discuss in chapter 7, under the label of **blanket mechanism**: an ontological-cum-epistemic assumption that the entire discourse relevant to computation—the entire ontological field relevant to computer science—can be restricted to what is mechanically effective. Blanket mechanism is essentially a methodological commitment that privileges mechanism, disappears meaning, and comprehends only half of the nature of computation.

Blanket mechanism is the name of the wall in the cave.

7 Discussion

All sorts of rejoinders suggest themselves.

Re arithmetic, some will suggest that relations between integers and their canonical representations are so close—virtually isomorphic, pace an exponential difference in complexity—that it is otiose to distinguish them. No one, they will argue, gets confused. Consider a case already encountered: the idea of a "16-bit integer." This term presumably means something like the following: "an integer (i.e., an abstract number) ranging from -32,768 to 32,767, representable, using standard two's-complement binary encoding, in two bytes of binary computer memory." The qualifiers "16-bit" and "in two bytes" apply to numerals (representations), whereas, at least in the first instance, "ranging from $-32,768$ to 32,767" applies to numbers so represented. Numerals, per se, do not "range."

Whether people are ever confused about the number/numeral relation is hard to say,[32] but a reflective system might be.

---

[32]Some people are indisputably confused about numeral/number relations in regards to complexity results. Schoolbook multiplication is said to have complexity on the order of n^2—where n refers not to the size of the abstract *numbers* being multiplied, but to the size of their binary (decimal, etc.) *representation*, which in turn is proportional to the logarithm of the represented number. Strictly speaking, therefore, the complexity of multiplying integers m_1 and m_2, using the schoolbook method (and assuming the usual positional notation), is proportional to the product of the base-10 logarithms of m_1 and m_2.

Suppose one tried to represent, in a Lisp-like language, the fact that the numeral «3» represents the number three. Lisp is not a knowledge representation language, so facts cannot be represented as such—at least not in any way that the official "semantical" account of Lisp would recognize. But if Lisp were extended to be able to represent facts, the natural form for this would likely be something like «(REPRESENTS '3 3)»—i.e., an abbreviation for «(REPRESENTS (QUOTE 3) 3)». But there are two problems with this suggestion. First, if subject to anything like traditional Lisp evaluation, the term "«(QUOTE 3)»" would be converted to «3», turning (the meaning of) the suggested formula into a false claim that «3» represents itself. In order for the suggestion to work, moreover, this version of semantics would have to be defined independently of Lisp's notion of "semantic value," leading to two separate accounts of semantics, concerning different things, using overlapping terminology. A sound reflective architecture must be clear and rigorous in regard to semantical issues—in no small part because reflection is fundamentally a semantical notion ("a reflective system is one that it is able to *represent*…its own operations, structures, and behaviour"). As soon as one brings representation within the system, even in as simple a case as that which relates numerals to numbers, relying on external human expertise to manage unadmitted conceptual complexity is not a winning strategy.

More generally, as Newell and Simon emphasized half a century ago, numerals denoting numbers is far from the only example of representation in computational systems. Consider object-oriented languages. It is standard to define classes or types in object-oriented programs to correspond to the concepts in terms of which we understand their non-mathematical task domains: *teacher*, *student*, *instructor*, *course*, *grade*, etc., in a university registration system; *institution*, *account*, *balance*, *transaction*, and so on, in the case of a financial system; and the like. According to the accepted practice of how theoretical computer science assigns semantics to such languages, their instances would be understood to have *memory records* (or mathematical models thereof) as their semantic values, not live

human beings, brick buildings, transactable dollars, etc.

As in the case of numerals vs. numbers, someone might argue that in simple cases people are unlikely to be confused, especially when the ontological structure of memory mimics that of the task domain—i.e., when the program is conceived as a direct model or simulation. But problems arise as soon as the program-world relation departs from being one-to-one, or from any other form of isomorphism. Suppose a university student is registered as a student in a course of which they themself are the instructor—something that happened in the case of AI philosopher John Haugeland immediately after he submitted his Berkeley doctoral dissertation.

Consider how this situation might be computationally modeled. Suppose, not unreasonably, that students and instructors were labeled using different schemes—students by student IDs; instructors by personnel numbers. In such a situation, a programmatic query as to whether the instructor of «COURSE-27» was a student in «COURSE-27» might well return false (something like «FALSE»), on the assumption, reasonable in general but incorrect in this case, that the types "student" and "instructor" (sets of students and instructors) would be extensionally disjoint. Or a system allocating classrooms might reason that the room for the course that Haugeland was teaching needed one more chair than was in fact required. These are not uncommon forms of bug: it is routine for systems to perform incorrectly due to failures in tacit assumptions—including the common assumption that identifier identity can stand proxy for identity of that which is identified.

The difficulty is not that it is up to computer science to ensure that programs cannot make or be constructed in terms of incorrect assumptions. That would be impossible—like requiring that a logical system never process or produce a false sentence in the course of making an inference. Rather, what betrays the problem is the discrepancy between (i) what we humas understand such constructs as

(IN (INSTRUCTOR (COURSE-27)) (STUDENTS (COURSE-27)))[33]

to mean—i.e., to be a representation or query about real people and real courses in the world—and (ii) what computer science takes the *semantics* of that statement to be, which in contrast has to do with computational-internal relations among computational structures. We take the proposition represented above to be *true*, because Haugeland was in fact a student in his own class. The computer science account would deem the computer warranted in returning «FALSE», because the "semantic values" of the constituent terms—i.e., the internal identifiers—are distinct. And because computer science has redefined the term 'semantics' to name this internal relation, there is no vocabulary left with which to say that the statement is in fact true.

The moral generalizes. By moving the representation-to-the-world relation out of theoretical view, by off-loading, onto programmers' and onto theorists' tacit understanding, all responsibility for ensuing appropriate system behaviour, which in general will be defined in terms of it, and by suggesting that the simple case of sign-signified isomorphism can be taken as paradigmatic—and beyond that by vainly hoping for the best—computer science abrogates its responsibility to provide a sound theoretical framework in terms of which to analyse and understand the systems that we build.

8 2Lisp

To deal with these and similar oddities—to provide a conceptual framework in terms of which to reveal, as clearly as possible, issues of meaning, mechanism, and their relation—I set aside the notion of a computational *value* entirely, in 2Lisp, and split the classic notion of evaluation into two roughly orthogonal notions:

1. One of *designation*, according to which 2Lisp structures were taken to have denotational, representational, or referential content (the "meaning" side of the dialectic); and

---

[33]In Python, «COURSE_27.instructor in COURSE_27.students».

2. One of *simplification*, according to which arbitrary program expressions were reduced to simpler, "normal form" co-designating expressions.

2Lisp simplification was reminiscent of term-reduction in logic and of α- and β-reduction in the λ-calculus. Simplification—a form of term-rewriting, where 'term' was extended to cover all effective computational structures—was taken to be the fundamental operation underlying the processing of 2Lisp programs.

Drawing a denotation/simplification distinction seemed eminently reasonable, given that simplification is clearly an operation that "stays within the boundaries of the machine," whereas designation is a relation that faces no such internalist restrictions. Simplification is manifestly a function from expressions to expressions (or, rather, in a computational context, from internal structures to internal structures). This made it unexceptional to assume that the domain and co-domain of 2Lisp simplification were the same, elements of which were produced and responded to in virtue of their exemplification of effective properties.

Borrowed from our understanding of logic, natural language, and human reasoning, in contrast, designation or reference was taken to be the relation between thoughts, words, symbols and ideas, on the one hand, and what we think about, talk about, hypothesize, and imagine—i.e., between intentional entities and the world towards which they are intentionally directed.

Depending on program and user context, 2Lisp expressions could designate anything at all (just as we can talk, think about, and imagine just about anything at all)—abstract numbers and functions, people, routes, and other arbitrary objects and situations past and future, as well as other internal structures, such as data structures, program fragments, storage locations, and so on.[34] The example above of a computational structure

---

[34]As explained in §2.5, there was to have been a fourth dialect in the sequence, called 4Lisp, in which the representation of external entities in

representing an academic course, with elements for instructor, students, topic, grades, etc., and a theoretical recognition that those real-world entities were what the computational structures designated, was perfectly in line with the 2Lisp approach.[35]

I viewed distinguishing designation from processing as a straightforward matter of conceptual hygiene—a necessary prerequisite to defining a clear, cogent, tractable notion of reflection. That is, I felt and argued at the time, it was necessary in order to satisfy the two-by-two reflective integrity criterion. In addition, I expected it to be relatively transparent, and self-evidently sensible.

I still believe the first claim—that distinguishing meaning, reference, designation, and other relevant aspects of the intentional nature of computation from the much narrower set of effective relations constitutive of computational mechanism is prerequisite to a coherent notion of reflection. On the second claim, though, about the move's being self-evidently sensible, I was wildly mistaken.

Far from clarifying matters, separating designation and processing gave 2Lisp (and 3Lisp, the reflective dialect defined in terms of it) a conceptual structure totally unfamiliar to programmers—so wholly at odds with reigning conceptions of programming languages as to be opaque to programmers and

---

task domains would be explicitly theorized. Data structures in current programming languages (including 2Lisp and 3Lisp) can of course represent such phenomena; the difference in 4Lisp was that this representational relation was to be theorized as part of the language design, and therefore formulable as something to which programs could be held normatively accountable.

[35]More accurately, they were in line with what was to have been the 4Lisp approach. 2Lisp licensed external real-world reference, and so technically speaking all these examples were 2Lisp structures. The 2Lisp processing regimen, however—and that of 3Lisp defined in terms of it—was defined only with respect to such external referents as numbers, truth values, sequences, etc. (as well as with respect to references to all computationally internal structures). See §2.5.

theoreticians alike. In particular, it opened up cracks among:

1. The official story about the nature of computation, running from Turing's 1936/7 paper through to the present day, and built into the tacit assumptions underlying contemporary theoretical computer science;

2. The tacit understanding of computing on which programmers rely in their daily practice, managing the relation between the code they write and the demands on their code stemming from requirements framed in terms of the projects and task domains for which the programs are being written; and

3. Long-standing and deeply rooted understandings of how symbols, languages, and representations stand for and orient us towards entities and phenomena in the world around us—the very tradition on which computer science is historically founded.

9 Approach

To reach these conclusions, as noted above, the investigation starts by identifying problems that arose in the development of 2Lisp and 3Lisp: problems having to do with the nature of programs, their (human) interpretation, and the frameworks that computer science used to analyse their "semantics."

The book then steps serially through various different conceptions of *program* and of *semantics*, in search of tenable conceptual footings. I first examine programs understood as effective specifications of computational behaviour, then as ingredients in algorithmic manipulations of structures representing task domains. The exploration systematically documents unresolved questions about programs, about data structures, about relations between representations and what they represent, about both effective and non-effective properties of computation-internal entities, and a spate of similar issues.

In chapters 2-5, the case is made that no incremental adjustment of our current theoretical framework is powerful enough to generate a satisfactory solution. Chapter 6 backs up

to review the conceptual structure of logic—the intellectual forebear of computer science, on which so much of computer sciences' approach and vocabulary are based. One goal of doing this is to show that, whereas computer science has come to focus almost exclusively on the mechanical and the effective, logic—even formal logic—was never so restricted. On the contrary, though it is not usually formulated in these terms, logic developed a valuable understanding of something I call the **core intentional architecture:** the architecture of systems explicitly exemplifying both meaning and mechanism, interdefined but neither conflated nor confused. As explicated in chapter 6, the effective conditions constitutive of the operation of logic-based mechanical systems are associated with syntax and derivation or proof ('⊢'). Issues of meaning, including non-effective representation, relations to both abstract and concrete elements, etc., are theorized under semantics, interpretation, and entailment ('⊨'). The dimensions are related in two of the most important notions in logic: those of *soundness* and *completeness*.[36]

Systems of formal logic, and active inference systems based upon them, are exceedingly restricted—profoundly incapable of dealing with the suffusing complexity, circumstantially dependent semantics, causal impact, process control, side effects, and myriad other facts routinely encountered and dealt with in contemporary computational systems. But the underlying core

---

[36]That the dimensions of both meaning and mechanism are considered to be constitutive of a logical system is evident in the fact that inference and proof, in logic, are understood to be defined over the *formal* or *syntactic* properties of the expressions. The *logical* properties would be understood to include both the syntactic and the semantic. To say that inference or proof was defined over the logical properties of sentences is either inexact or confused.

In contrast, it is commonplace to talk about the *computational* properties of programs, data structures, etc., taken to include only the effective properties, analogous to the syntactic and proof-theoretic properties in logic. This fact alone betrays the fact that genuinely semantic (non-effective, distally oriented issues of interpretation and intentional directedness) are sidelined in current computational theory.

intentional architecture is based on a profound insight into the nature of intentionality—an insight that remains applicable to present-day computing, in all its wooly complexity. Unfortunately, the insight has disappeared in contemporary discussion, rendered invisible by the shift in vocabulary and computer science's singular focus on the effective. The fundamental insight at the heart of logic has to do with how an intentionally constituted system can be effectively realized and normatively governed by non-effective representational relations to distal subject matters. That insight has fallen by the wayside in our theoretical imaginations, but it remains of enormous significance—as applicable to present day computing as it ever was to purely mathematical and logical deliberation. That *formality* and *syntax* constitute the mechanical dimension of logical systems may not be immediately evident, in part because the general question of what it is to be a logical system is not normally theorized as such. What receives theoretical attention, in logical treatises, are usually particular logics or particular families of logics: first-order quantificational logics, predicate logics with equality, and the like. What it is to be a syntactic property, for example, is not typically theorized; rather, the syntactic features of the logic(s) under investigation are typically posited or demonstrated ostensively.

Still, there is widespread tacit agreement on a variety of background norms and standards on logical practice. Some are relatively specific, such as that syntactic and semantic properties be defined in such a way as to support what is known as *compositionality*—the idea that the semantic value of a composite expression be a function of the semantic values of its syntactic constituents. Others are more general, including that the syntactic properties and formal rules of inference be such as to make semantical sense. What is most important for our present purposes, as argued in chapter 6, is that syntactic properties *must be effective*. Any attempt to propose a logical system in which a non-effective property was deemed syntactic would be censured—considered to be a cheat.

Systems of logic, that is, satisfy what in *The Promise of*

Artificial Intelligence (henceforth *Promise*) I call the REPRESEN-
TATIONAL MANDATE—six conditions constitutive of all core
intentional architectures:

Conditions (DIALECTIC)

C1   An intentional system must work, locally, in virtue of the
     effective properties manifested by its embodiment.

C2   Overall, it is normatively directed towards the world as a
     whole, including much that is not effectively available.

C3   Being physically embodied, it has no access to the non-
     effectively-available states in terms of which the norms on
     its operation are defined.

So what does it do? (ARCHITECTURE)

C4   It exploits local, effective properties that it can use, but
     does not (intrinsically) care about

C5   To "stand in for" or "serve in place of" properties and re-
     lations of states of affairs it cannot effectively access

C6   In order to behave appropriately towards those non-ef-
     fective states that it does care about, but cannot use.

C1 is an acknowledgement of the constraints imposed by an
overarching physicalism and the limits of effective causality; C2,
of the fundamental nature of intentionality; and C3, of the dia-
lectic that follows from C1 and C2. C4–C6 are effectively a sum-
mary of the notion of *representation*, with C4 framed to include
both internal representations (memories, data structures, con-
figurations of ingredients, etc.) and external representations
(maps, signs, images, external language, and the like).

   As noted above, logical systems, and especially those stud-
ied under the guise of formal logic, are extraordinarily more
specific than required by the core architecture. Among other
things, they typically assume: that syntactic inference is inde-
pendent of semantic interpretation; that semantic interpreta-
tion is independent of inference and all other contextual factors
(such as changes in external state, the passage of time, and so
on); that inference systems are state free (so that whether an
inference is legitimate does not depend on what other

inferences have taken place); and a host of other things. The importance of formulating the core architecture in this investigation at a broad level of generality, however, is to make a simple point: the computational systems we build and use in the world are *also instances of this same architectural type*.

I warrant that all programmers have a deep appreciation of C1-C6—in their bones and in the regularities in their practice, if not (yet) in their explicit understanding.

The fact that the core intentional architecture applies as much to computing as to logic suggests that its generality stems from fundamental facts about intentionality, embodiment, etc., thereby at least beginning to erode the idea that computation is "special" in the sense suggested in §1.1. The argument to this conclusion will be buttressed by an analysis of how we have ended up in the situation we are in—how computer science came to focus so exclusively on effective mechanism, how all of our semantical and intentional vocabulary came to be redefined to deal with local, computation-internal, effective matters, how such remarkable progress has been made in theoretical results that have been developed to date even while more far-reaching intentional issues have been sidelined, etc. A full explanation belongs to intellectual history, but at least the following contributing threads can be identified:

1. Pressure to fit the study of computing into our conception of "science," complete with its naturalistic presumption of the explanatory adequacy of causal explanation;

2. A focus on the things that need to be understood in order to construct intentional systems—i.e., an abiding interest in what it takes to *implement* or *build* computational processes;

3. The fact that computer science started out as, and in many cases remains, an engineering practice, in which programmers and theorists alike understand full well that computation involves issues of representation, but also understand, at least tacitly, that computer science as it stands

does not theorize those relations;

4. The fact that, in the first half of the 20th century, what was monumentally significant was the recognition, by now taken as a truism but at that point far from obvious, that it was possible to construct a physical device that could be understood as taking rational steps in a logical argument, under seemingly obvious and self-evident regimes of semantic interpretation;[37]

5. The fact, in spite of Newell and Simon's reminder that computation is broader than this, that numerical computation has remained the primary focus of theoretical analysis—a case in which, even if the effective structure (of numerals and representations) and the structures they represents (numbers, functions, etc.) are not isomorphic, there is sufficiently widespread agreement in practice on which representational schemes are being assumed that confusion has been largely avoided;

6. The fact that a pact has been tacitly agreed, by programmers, theorists, and users alike, that it is up to them to shoulder responsibility for navigating the complex intentional and representational nuances and subtleties that accrue to contemporary systems, relating the constructed systems to their task domains; and

7. Perhaps also, as will be highlighted in chapter 8, a tacit recognition that the subtleties underlying the representational and intentional properties of contemporary computational systems are almost stupefyingly complex, and that dealing with them adequately in compelling theoretical terms will be extraordinarily challenging—far beyond the capabilities of any currently available semantic or even ontological theory.

---

[37]John Haugeland emphasized this historical fact—which underlay his widely quoted (but in my view also widely misunderstood) claim that "if you take care of the syntax, the semantics will take care of themselves." See §6.4.

This book will tell a story about this subject matter, in an attempt to hew an intelligible path through a thicket of theoretical issues and forge a comprehensible proposal for how we can move forward—a story of how we can open a new era of understanding of computational and intentional systems. But while that will be the end of the road for this book, it will not be the end of the road for inquiry. The book's ultimate aim is not to bring one era of analysis to an end, but to open up a new one that is clearer, more encompassing, and more accurate.

C — Conclusion

I would be the first to admit that the territory that this investigation explores has been only inchoately mapped. At issue are intricate relations between the mechanical and effective operations in virtue of which embodied intentional systems work, and the non-effective semantic relations of directedness they bear to the subject matters they are about—relations pertinent to the norms they are enjoined to honor.

The philosophical literature is replete with accounts of some human dimensions of this issue. Of signal importance is Frege's 1892 distinction between cognition-oriented *sense* and world-directed *reference*.[38] Other proposals have been presented that, in various ways, attempt—to put it in Wittgensteinian terms—to account for both meaning and use: accounts of pragmatism, activity theory, inferentialism, and the like. None, though, are candidates for serving as general models of intentional significance. Some, such as "conceptual role semantics" or "two-factor semantics,"[39] assume that denotation or reference is always to an external world, and that all effective operations are internal—a pair of assumptions inapplicable to

---

[38]Frege (1892).
[39]Also called inferential role, functional role, procedural semantics, semantic inferentialism, and other labels.

computer systems, which abound in internal denotation and external interaction. Some frameworks, such as those that take two sentences to "mean the same thing" if, intuitively, they license the same inferences, are defined with respect to disqualifyingly high-level intuitions about human reasoning. Some apply only to external language, whereas anything appropriate for computing in general will have to account for arbitrary internal states (the computational analogue of "mentalese," as it were). More seriously, none has been worked out in anything like the technical detail required to serve as the basis for architectural or language design in computer science.

On the computational side, efforts to theorize relations between computational systems and the world, such as knowledge representation languages, resource description frameworks (RDF), models used in requirements engineering, etc., are not only too ontologically simple to do justice to the needs of "computation in the wild," but are also designed for much coarser-grained application (overall goals, classes of inputs and outputs, etc.) than the intricacies of detailed program code. In addition, they typically fail to address issues of contextualism, the impacts of use and processing on denotation and meaning, etc. One of the strengths of contemporary programming language semantics is that its accounts, at their best, are designed to deal with just such complexities. The difficulty is that what they account *for* is restricted to what is ultimately mechanical, effective, and internal.

Progress will undoubtedly be made integrating these approaches and techniques, developing new ones, and framing accounts that do justice to computational systems as unrestricted ("non-special") intentional systems embedded in unlimited worlds. Numerous challenges will need to be addressed en route, including the development of an ontological framework adequate to accommodating the surpassing complexities that programmers deal with, albeit tacitly, in the programs they write. But the daunting nature of the task should not weaken our theoretical resolve. It is five hundred years since the beginning of the Scientific Revolution and consequent efforts to

theorize the "natural," mechanical, physical world. We should not expect the task of developing an equivalently comprehensive understanding of the realm of meaning, intentionality, and reference, etc.—the realm of significance—to be any simpler or more straightforward.

2 Reflection

Reflection is an invaluable site for analyzing computation because notions of meaning and mechanism arise doubly. It must represent and be able to act; and it must represent and be able to act upon its ability to represent and to act. That is, it must satisfy what in the Introduction I called the *reflective integrity criterion*.

This chapter will focus on reflection, in order to develop intuitions to which to hold subsequent analyses accountable.

A reflective system is one able to represent, reason about, and direct intentional actions towards its operations, structures, behaviour, and engagement with the world—and to do so in a dynamic, integrated, and on-going fashion. Reflection is a substantial capacity in its own right. Among other things, it is a prerequisite to the possession of any capacity that could be called genuine intelligence. But inchoate reflective capacities have been recognized as powerful in a wide range of less exalted computational settings. Most contemporary programming languages provide at least rudimentary capacities of this sort. While not as central as recursion, reflection has become a staple ingredient in the conceptual arsenal of contemporary computer science.

Just what reflection is, however, remains an open question. No general theories of computational reflection have been proposed, for reasons that reach deep into our understanding of computation itself. As noted in the Introduction, this makes reflection an ideal site for rigorously examining the nature of computing—and in particular for assessing the adequacy of

current theoretical frameworks, for uncovering their limitations, and for developing more adequate alternatives.

Reflection's importance is not restricted to the computational realm. It is also an instructive example with which to challenge a variety of reigning assumptions about mind, consciousness, and other topics in artificial intelligence (AI), cognitive science, and philosophy. These larger topics are not the subject matter of this book, which focuses on the computational case. Still, some comments on them will be made in passing, since computing can be viewed as a small distillation of many of our deepest theories about the nature of intentionality and mind—especially if computing turns out not to be "special" in the sense discussed in §1.1. And even within the computational realm, the aim here is not to understand reflection in a narrow sense, but to use its analysis as a strategy for revealing foundational issues in programming, implementation, and computing in general.

1 Preliminaries

Reflection can be understood as a form of self-reference, so long as both *self* and *reference* are interpreted broadly. At issue is not simply the possibility of a single expression, data structure, or event denoting or referring to itself, in the narrow form exhibited in such familiar logical paradoxes as the famously contradictory "This sentence is false."[1] Rather, the notion of reflection

---

[1]"This sentence is false," known as *The Liar*, is self-contradictory because if it is true, it must be false; and if false, true. See e.g. Barwise and Etchemendy (1987).

Similar forms of narrow self-reference characterize the sorts of logical puzzle made famous by Raymond Smullyan (e.g., see Smullyan 1978). Though it is not *referential*, and thus ineligible for being called self-reference, I would count the cyclicity of set-theoretic non-well-foundedness, where a set may contain itself as a member, as similarly "narrow," since the cyclicity is *local*—a single binary relation between two unitary entities (unitary for the purposes of establishing the cycle).

So too recursion. I do not believe that recursion involves self-

to be explored here has much wider scope, both intensionally and extensionally—including not only the ability of a system to refer to or designate itself or its ingredient structures (including the ability to *name* both itself and its parts),[2] but also, employing descriptive and theory-laden capacities, to *find itself richly intelligible*—as a complex entirety, in fine-grained componential detail, and as both dynamic and contextually embedded.

For a system to count as reflective, I will argue, such capacities must be integral to a system's overall intentional (representational, descriptive, etc.) and active capacities, rather than be separate from or external to them, of the sort that might be employed by a detached observer or theorist, or be exemplified in a modeling system used to model itself.[3] Moreover, in order to achieve any kind of power and generality, the idea is for a system to represent, reason about, engage with, and modify itself *in the same way, and using the same resources*, that it uses to represent, reason about, engage with, and potentially modify the external world or task domain in which it is embedded. As well as increasing generality, elegance, and power, it is crucial for the resources for self-representation (reasoning, etc.) to be the

---

reference, but there is nevertheless a related cyclicity in recursive definitions; they too I would count as narrow.

[2]This characterization crosses the "personal/subpersonal" (or analogous "system/subsystem") divide. It can certainly be argued that people's introspective or reflective capacities exist primarily only at the personal level; we are able to consider ourselves, our behaviour, etc., or aspects of ourselves or behaviour, *as whole people*. Famously, we lack direct introspective access to the components or physical configurations of our brains, or to other fine-grained aspects of mental implementation—though, needless to say, we have no difficulty referring to individual thoughts and beliefs, to physiological aspects of our person (limbs, torn muscles, sciatica, etc.), to such psychological aspects as confusion, paranoia, and happiness, and to aspectual facts about our phenomenology influenced by subpersonal issues of implementation, such as being over-caffeinated, inebriated, or drugged.

[3]Thus to construct a model of a general modeling system $\mathcal{M}$ within $\mathcal{M}$ would not make $\mathcal{M}$ reflective, for reasons explained below.

same as those used for representing the external world or tradi-
tional task domain so as to enable the system, in full reflection,
to comprehend itself as an integrated, intelligible part of that
world—to be able to see itself as *in* and *of* the world as a whole.

It is in part because of this wider compass that reflection is
such a useful crucible in which to test the mettle of any pro-
posed understanding of computation.

At its base, reflection builds on a suite of familiar technical ca-
pacities: quotation and disquotation, logical "reflection princi-
ples,"[4] "representation theorems," facilities for semantic ascent
and descent, the ability to construct representations of primi-
tive structures and operations, facilities for "treating programs
as data," etc.—i.e., the very sorts of resource that underlie a va-
riety of forms of narrow self-reference. As architectures grow
more complex, however, reflection's demands reach further—
to include a system's ability to construct (and act upon) descrip-
tions, theories and models of itself, not only as a static or passive
structural entity, but as a complex, dynamic, contextually em-
bedded system that is, in turn, complexly, dynamically, and
contextually described. Moreover, it is only in that more de-
manding context that the stuff and substance of reflection
comes into view.

Perhaps the best way to understand computational reflec-
tion is as a technical analogue of what *self-knowledge* means in
everyday discourse (not as it is theorized in philosophy, where
self-knowledge, too, is often understood in a narrow sense[5]). It
is one thing to know one's own name, to be proficient with the

---

[4]Though related to the notion of 'reflection principle' in logic, the can-
vas of reflection as understood in this book is much wider.

[5]Philosophical accounts of self-knowledge largely treat no more than
the relation between a subject's being in a mental state and *knowing*
that they are in that mental state—e.g., between being in pain and
knowing that one is in pain; between believing that there will be a third
world war (to use Evan's famous example), and *believing* that one be-
lieves that there will be a third world war (Evans, 1982).

first-person pronoun, to comprehend the world from a single perspective, and to recognize and recall what one said a moment earlier—all of which, on the human side, are well within the capacities of young children. It is another to be dispassionately observant about one's conduct and impact on others, to have insight into one's motivations and behaviour, to be effective in deploying self-understanding gained through years of psychoanalysis, self-discipline, and/or considered experience. While no computational system yet constructed approaches such levels of prowess, the notion of reflection examined here and the computational architectures I explore were designed from the outset with those more ambitious accomplishments in mind.

As we witness unrelenting progress in AI, neuroscience, machine learning, cyborg engineering, and the like, such a substantial notion of reflection will only grow in importance.

2 Failure

In the 1980s I formulated a concept of "procedural reflection"—a capacity of a programming language to include elementary reflective capacities. Among other things, I wanted to show that reflection was a coherent and well-behaved notion, to illustrate its power, and to discharge the sense of mystery that surrounded the notion, because of its relation to such genuinely challenging phenomena as consciousness. Though the project was conducted within the confines of a simple traditional programming language, I also undertook it as something of a "design study" in order to establish more adequate footings for constructing more ambitious reflective systems in knowledge representation and artificial intelligence more broadly.

The primary focus of that original work was architectural. In my doctoral dissertation and a number of subsequent papers I showed how an extremely elementary form of reflection could be implemented in a purpose-built dialect of Lisp. As well as presenting, explaining, and assessing this language, the dissertation also explored, at a more general level, some of the

conditions, theoretical as well as practical, that any system must meet in order to be counted as genuinely reflective.

Among other things, reflection puts special attention on the intentional dimension of a computational system—on the "aboutness" of the system, on what is traditionally (outside of computer science) understood under the label of 'semantics'— for the simple reason that a reflective system must be capable of representing or reasoning about its own construction and behaviour. That is, a reflective system must be capable of manifesting or engaging in behaviours that are *about* its self. To put it bluntly: *reflection is a semantical notion*. The architecture and intelligibility of a reflective system is thus singularly dependent on the notion of semantics or "aboutness" in terms of which that characterization is framed. As such, reflection serves as an distilled case in which to assess reigning notions of semantics.

One of the larger claims I argue in this book is that all contemporary understandings of computational semantics are inadequate.[6] They are inadequate for many reasons, including some that relate programmers' tacit understanding of programs with the dictates of contemporary computational theory, as well as more specifically being inadequate in terms of which to define a notion of aboutness adequate to the concept of reflection.

More specifically, present-day notions of computational semantics restrict their focus to mechanical configurations and mechanically induced behavior within the boundaries of machines—on the causal underpinnings of computational processes, on the "mechanism" side of the fundamental dialectic. That is, semantics as it is understood in computer science is restricted to the realm of the *effective*. To put it in computational

---

[6]I have in mind (and this book will primarily discuss) the semantics of computational systems themselves, of the programs we write to control them, etc. The focus will not be on what is sometimes called the "computational semantics" of (human) language and mind— though the morals and insights drawn out here are relevant to those topics.

terms: "semantics" as currently understood in computer science is more about implementation and causal behaviour than about the nature (especially the intentional nature) of that which is implemented. This remains true even when computation is studied in abstract or mathematical terms.

This computational understanding of semantics is ultimately so different from the traditional understanding, which has to do with *non-effective* long-distance aboutness, that when the differences are salient, I will refer to them using different labels, using semantics-L for semantics as in logic—i.e., the non-effective aboutness discussed in the Introduction, constitutive of the meaning side of the fundamental dialectic; and semantics-C for semantics as that word is currently used in computer science, which focuses instead on the perhaps mathematically modeled behavioural (effective) consequences of a computational structure or operation. (More generally, when confusion might arise, I will append an 'L' to intentional words used as in logic and natural language, and a 'C' to those same terms as used in computer science.) As recognized in some contemporary philosophical discussions,[7] computer science treats computing as a physical, mechanical, or effective notion, in spite of its ubiquitous use of the term 'semantics.'

To serve as a background and reference for this discussion, I have assembled relevant parts of the original dissertation and some subsequent papers written in the 1980s into an accompanying volume (*The 3Lisp Legacy*). Along with the original texts, that book includes annotations tying specific discussions to the issues explored in this volume. I have brought them together to serve as a background for the present discussion, in part because I believe they incorporate some insights of enduring value. But the catalyzing motivation, not just for that volume but for the present one as well, stems from the fact that the dissertation and these papers all ultimately *failed*—both

[7]Piccinini (2015).

practically and theoretically. The reasons for that failure have yet to be explicated. It is those reasons, and their consequences for computing more generally, that I believe to be of contemporary significance.

The problem is not that the proposed reflective dialect of Lisp, called 3Lisp, was never used. While true, that was to be expected, if for no other reason than 3Lisp was a test demonstration—a vehicle developed to make a theoretical point.[8] Rather, 3Lisp, and the specific architecture it was meant to exemplify, was both: (i) *unusable*, for all intents and purposes, not just in pragmatic form but in fundamental conception; and (ii) *unintelligible*, given the discourse of computer science into which it was introduced. The reasons for the dysfunction and the incomprehensibility are different, telling, and not easy to fix.[9] As I will argue here, they stem not from local facts about 3Lisp, or indeed from any issues about 3Lisp in particular, but

[8]I doubt that many of the other languages that were subsequently proposed as models of reflection—including Brown (Friedman and Wand, 1984), Blond (Danvy and Malmkjaer, 1988), M-Lisp (Muller, 1992), etc. —enjoyed much use, either, for a similar reason: they were more theoretical demonstrations than practical proposals. That does not mean that they, or 3Lisp, lacked influence. The presence of at least inchoate reflective capacities in numerous contemporary languages, including Java and Ruby, stands witness to the impact that reflection has had since these special-purpose demonstrations were published.

[9]Those familiar with 3Lisp are likely to suggest that the dialect's evident dysfunction has to do with its excessive strictness about distinguishing signs and symbols from what they signify or denote—what philosophers would call issues of *use* and *mention*—such as its insistence on relentlessly discriminating among numbers, numerals, and quoted numerals, and its distinction between representations of sequences of entities and sequences of representations of those same entities (cf. §2.2).

Many of these accusations are true. Less obvious, though, is one of the issues to be explored here: *why* 3Lisp imposed such strictness, and why this semantic rigour, which seemingly added conceptual clarity, ultimately contravened, rather than facilitated, practicability. See chapter 8, and "The Correspondence Continuum" (Smith, 1987).

from inadequacies in the conception of computation in terms of which it was designed—inadequacies which can only be addressed via the wholesale reconfiguration of computer science recommended in chapter 8.

The general design of 3Lisp, including many of the aspects that made it unusable, grew out of an attempt to repair confusions, particularly semantic confusions, that infected (and continue to infect) our understanding of the nature of programs and programming. These confusions, normally glossed over in practice, come to light under the scrutiny required in order to construct a system that is reflectively sound.[10] Cleaning up these confusions, which I believed to be a prerequisite to defining an elegant reflective architecture, hardly seemed like a contentious goal.[11] Yet doing so, and thereby achieving what from a classic philosophical perspective would be considered a degree of theoretical elegance, yielded a programming language that, far from being unequivocally better than previous dialects, was in many ways worse.

Philosophical clarity, that is—at least clarity in terms of reigning philosophical norms—was achieved at a price of such semantical strictness that it vitiated day-to-day practicability. More seriously, arranging for 3Lisp to hew closer to the conceptual frameworks in terms of which we understand language and logic pulled it away from how we presently understand computing. Aligning the notion of semantics in terms of which 3Lisp was designed more closely with traditional notions of semantics-L, that is, one of my design goals, required distancing it from reigning computational presuppositions about semantics-C. Diagnosing why those things happened, in turn—that

---

[10]See §4.V4 on the applicable notion of soundness.

[11]At the time 3Lisp was designed, I was convinced that it was theoretically cleaner than Scheme, which had just been developed at MIT. I fantasized that it, or at least 2Lisp (a non-reflective precursor to 3Lisp that nevertheless cleaned up the confusions I felt were endemic to Scheme), might replace Scheme as the paradigmatically clean and distilled version of Lisp. It was not to be.

is, why resolving evident confusions can hinder, rather than help, both practical usability and intellectual comprehensibility—reveals a wealth of yet additional difficulties, the theoretical measure of which has yet to be taken. While some of the complexities have to do with reflection *per se*, most stem from more basic unresolved issues in computing that come most prominently to light when reflection is at point.

Three examples, to give a taste:

1. The discrepancy between the semantics of *particular programs* and the semantics of the *programming languages* in which they are written. It is no accident, given the current state of computer science, and the exclusive focus on the semantics-C construal of semantics, that we presume to some understanding of the latter, and virtually none of the former—and sobering, too, that our accounts of the latter do not even provide adequate resources in terms of which to develop the former.

2. A related gap between our "official stories" of what programs mean and the tacit understanding of them that, so far as I can tell, working programmers universally share—perhaps even *must* share—in order for programming to be a humanly viable endeavour.

3. A profound inability of all currently accepted formal models of semantics—across philosophy and logic as well as computer science—to deal with the ontological profusion that permeates what I have called **computation in the wild**: real-world programs engaging with genuine, concrete task domains.

Some of 3Lisp's inadequacies can be understood in terms of its attempt to address these issues. The problem is that its approach was strong enough to tear away the blinders that for decades have kept us unaware of the conceptual difficulties underlying all of them, but too weak to serve as the basis for a viable alternative. So it stands as something of an unstable waystation en route to a better framework.

So that is one failure: 3Lisp achieved (what at least seemed like) semantical clarity at the expense of usability. But that was not the only problem. Equally telling is the fact that the papers presenting 3Lisp failed *communicatively*.

It is not that the work on which they report received no recognition. On the contrary, interest in reflection burgeoned after descriptions of 3Lisp were first published,[12] with a number of follow-on dialects developed, papers written, and workshops held. It would be an exaggeration to claim success on one of the project's original goals: to make reflection as ubiquitous and unproblematic as recursion. Still, as mentioned above, whereas the word 'reflection' was essentially foreign to computational discourse in the 1970s,[13] the majority of contemporary programmers would now recognize the term, and have at least a rudimentary sense of what it would be (indeed, what it is) for a computational system to be reflective—at least in the sense in which they understand it.

In spite of this notice, however, the fundamental *idea* for which the original 3Lisp papers argued—the insight that I still feel to be crucial to understanding reflection—died stillborn. The problem was not so much that it was rejected, as one might have expected had it merely been a bad idea, but that it was almost wholly ignored in all follow-on work, even work that claimed to be inspired, among other things, by 3Lisp itself.

Four decades on, I still believe that the idea is important—not only for any architecture laying claim to being reflective (even if the form in which it was exemplified in 3Lisp was simplistic and fatally flawed), but for a more general understanding of self-knowledge and even consciousness, and indeed of all forms of human understanding and engagement. The idea

---

[12]Especially the 1984 Principles of Programming Languages (POPL) paper, entitled "Reflection and Semantics in Lisp" (Smith 1984).
[13]Reflection was recognized as a concept in logic, of course, in notions of reflection principles. One of the goals of this text is to demonstrate how, and why, introducing an analogous notion into computational contexts is hugely more complex.

remains essential, for example, in any effort to take the measure of contemporary developments in machine learning and second-wave AI. The difficulty is that it was rendered invisible by the theoretical commitments that then governed, and still do govern, the entire field of computing—commitments that I believe are ultimately intellectually untenable, and that block progress across all of cognitive as well as computer science.

Understanding what rendered the idea invisible requires pressing hard on what turn out to be surprisingly divergent understandings of computation currently prevalent in AI, computer science, logic, and philosophy of mind. And once one presses hard, cracks start to appear, undermining confidence that theorists and practicioners in these various fields are even talking about the same thing. These difficulties erode any sense that we are close to comprehending the space of meaningful mechanical systems.

In addition to whatever merit it has in its own right, in other words, reflection serves as an unparalleled site for exploring intellectual parallels, collisions, and disjunctions among computing, philosophy, and cognitive science. One might have thought it would be uncontentious to mesh a reasonably straightforward model of semantically disambiguated representation with an equally familiar conception of a programming language. But as soon as the two were brought together, the combination started to tear itself apart.

3 Anecdotal History

A personal digression on the history of 3Lisp may provide some useful context.

In the 1970s, while a graduate student in the Artificial Intelligence Laboratory at MIT, I set forth on an incurably ambitious project: to design and implement, from the ground up, a new kind of knowledge representation system. I had in mind something small, elegant, and powerful—more of a kernel or virtual machine than a full-fledged reasoning system. From the get-go, I imagined that the calculus would integrate declarative and procedural dimensions in a conceptually integrated way.

Rather than consisting of two interacting parts, that is—a declarative representational language tied together with a procedural programming language—I envisaged a single, semantically cohesive architecture, in which all behaviour was integrated with employment of its integrated representational capacities. Modulo various logical intricacies, that is, including limits on explicitization, the system would only *do* what it *represented itself as doing*.[14]

The project of developing a kernel architecture or calculus has antecedents. For more than three centuries the differential calculus has served as an elegant basis in terms of which to formulate classical physics—not merely because of its general mathematical power, but more specifically because it so deftly incorporates, in its very architectural design, ontological commitments about its subject matter shared across all its domains of applicability. Upon enrolling as a student at the MIT Artificial Intelligence Laboratory, I was analogously impressed, especially through Sussman and Steele's contemporaneous "Lambda papers,"[15] by how Lisp (about which more in a moment) could play an analogous role in computer science—serving as a compact basis for numerous seemingly disparate programming constructs, well beyond its evident support of recursion and functional programming. In setting out on the ill-fated dissertation project, I thought—and to some extent still do think—that I could imagine an equally distilled active representational or descriptive computational calculus that could serve as the heart of wide-ranging systems capable of knowing

[14]The normal reaction to the (Lewis, not Michael) Carroll paradoxes is that some behaviour must "happen directly," rather than be engendered by the processing of behavioural description. Reflection provides a novel approach to the issue by allowing that truth to obtain only in the limit, without imposing a finite bound on the number of levels of descriptive ascent accessible in any given circumstance.

[15]Sussman and Steele (1975), Steele and Sussman (1976), Steele (1976), Steele (1977), Sussman and Steele (1978), Steele and Sussman (1979), Steele (1980); note they were being released at the time.

and reasoning about the world.[16]

The system was to be known as 'Mantiq' ('منطق", approxi-
mately the Arabic equivalent of the Greek 'λόγος', or *logos*). It
arose in part out of my involvement, over a number of sum-
mers, in Winograd and Bobrow's project at the Xerox Palo
Alto Research Center (PARC) to develop a "Knowledge Repre-
sentation Language" known as KRL.[17] Though in some ways
Mantiq was inspired by the aims of KRL, it was equally moti-
vated in contrast. Though I was sympathetic to KRL's attempt
to fuse descriptive, procedural, and architectural capacities,[18] I
was dismayed at its mushrooming complexity, impatient with
the initial version's retention of a distinction between a "declar-
ative" representational language and the "procedural" program-
ming language (Interlisp) used to specify behaviour, and suspi-
cious that the designers' proposal to add procedural and meta-
level facilities "later" (i.e., in some imagined future version), ra-
ther than designing them in at the outset, would sentence the
system to irredeemable baroqueness.[19] I imagined something

---

[16]It is not hard to think that one can imagine something, of course.

[17]Bobrow, Winograd, and the KRL Research Group (1977), pp. 213–
222.

[18]Just one example: KRL attempted to implement default reasoning ar-
chitecturally, rather than explicitly. Suppose for example that β (say,
that *x* flies) is intended to be a default entailment of α (that *x* is a bird).
One of our goals, in designing KRL, was to construct a system that,
when presented with α, and absent any explicit countervailing indica-
tion, would conclude β without having to consider a representation
whose content was along the lines that "β was a default consequence
of α." That is, the aim was for KRL to *conclude, by default, that β was a
consequence of α*, rather than by to (explicitly) *conclude that β was a
default consequence of α.*

[19]If the system was to be adequate as a descriptive framework, why
not use it to describe any behaviour that was to be engendered? Alt-
hough this was a sentiment with which Bobrow & Winograd agreed,
they did not deem it possible to pursue in KRL-0. Eager to move for-
ward towards such a goal, the project I took on, as a student member
of the project, was an attempt to represent KRL descriptive structures
within KRL, as a first step towards incorporating a procedural

orders of magnitude simpler, yet at the same time far more *generatively* powerful—a core, integrated calculus that would wear a combined essence of reasoning-cum-representation on its sleeve.

In 1978, as a design study en route to Mantiq, I decided to work out, within the familiar context of a programming language, the notion of reflection discussed in the papers included in *Legacy*—the kinds of self-directed reasoning and representation that I knew would need to be a fundamental architectural dimension of the ultimate Mantiq system. Once reflection and "a few other details" were in hand, I planned to turn to Mantiq itself.

For multiple reasons, I chose Lisp as the programming language to work with. Lisp was simple and expressive, for starters—at least in its core versions, including Lisp 1.5 and the initial versions of Scheme that Sussman and Steele were designing at the time. Second, as mentioned, Lisp embodied the aesthetic that, though difficult to articulate, was so important, and something I took to be a criterion on the design of Mantiq: of syncategorematically expressing, in relatively distilled form, at least some of the essential structure of its domain of applicability.[20]

---

language within the overall descriptive framework. There is no doubt that the 3Lisp language arose in part from this experience. Note that 3Lisp can be considered to be approximately the dual of that KRL subproject. Rather than represent procedural capacities within a descriptive framework (my task for KRL), my goal in 3Lisp was to instill the type of semantical framework appropriate to a descriptive calculus into the scope of a procedural language.

[20]Set theory is simple, too—and one undoubtedly could (some people do) model the behaviour of programming languages in set theory. But with respect to the domain of programming languages, set theory does not embody the aesthetic I was after. The reason I had been so impressed with the differential calculus as a kernel formalism in which to express classical mechanics and dynamics had to do with how many substantive physical insights were embodied in the formalism itself—including, for example, the idea of point-to-point temporally varying features (in the Strawsonian sense), not requiring any notion of a

Third—and contrary to subsequent misinterpretation—I also felt that, in its structures and its behaviour, both explicitly and implicitly, Lisp was admirably free from distracting influences of its implementation on von Neumann architectures. This would allow reflection in Lisp to be genuinely *about Lisp itself*— i.e., about the nature of Lisp (about the "pure Lisp virtual machine," to put it in computational vernacular), not about details of its underlying (physical or computational) implementation.[21] Fourth, Lisp was not bogged down by irrelevant issues of complicated syntax—a benefit for the "programs as data" focus of reflection and self-reference. Fifth, early versions of Lisp attempted to be higher-order, and Scheme actually achieved that capability—a fact that was relevant to a number of other long-term Mantiq design goals. Sixth, around MIT in the 1970s Lisp reigned supreme, virtually unchallenged as something of an AI programmer's *lingua franca*: *the* language in which to build AI systems. Seventh, and by no means least, from the first meta-circular interpreter for Lisp, called M-Lisp, formulated in McCarthy et al.'s legendary *Lisp 1.5 Programmers Manual*, Lisp provided built-in mechanisms for (structural) quotation and for dealing with program structures as data, thereby famously allowing informal play with (at least narrow) self-referential programming.

So I selected Lisp not merely as a language in which to implement a prototype reflective system—which, though it happens to be true, is from a theoretical point of view irrelevant— but as the basis for a design that was *itself reflective*. The distinction is critical for various theoretical reasons, especially

---

discrete individuated object for their exemplification. Set theory, in contrast, does not embed substantive theses considered to be universal to programming languages—does not architecturally build in those features that are and must be shared among all candidate programming languages (including behaviour, representation, effectiveness, and instruction-following).

[21]Garbage collection, for example, is in my book a fact about Lisp implementation on von Neumann machines, not about Lisp itself.

including semantical ones, but it had an additional side benefit. The study provided a dual opportunity: not only of understanding reflection well enough, or anyway so I hoped, to incorporate it in Mantiq, but also of filling in a gap in our understanding of Lisp itself. As noted, it was well understood in the 1970s that Lisp provided capacities for quotation and other (narrow) forms of self-reference; it facilitated ready demonstration of meta-circular interpreters; it seemed unproblematically to support (pace subsequent debates) macros, backquote, and other forms of programmable program modification; and Lisp 1.5's (McCarthy's) analysis of Lisp in Lisp suggested its use as its own meta-language. Yet I had always felt, since my very first encounter with the language in the 1960s, that in all then-extant dialects of Lisp these self-referential powers remained tantalizingly undeveloped and under-theorized. Designing a reflective dialect of Lisp, therefore, struck me as a worthy side project—offering the possibility of bringing to successful completion the self-referential promise that Lisp had inchoately offered from the outset.

As the work got underway, the subsidiary goal took on a life of its own: to deepen our understanding of Lisp's vaunted self-referential properties. Embarrassingly, until close to the end of the 1970s, it remained my intention to write a doctoral dissertation on Mantiq, incorporating a raft of metaphysical, ontological, and epistemic proposals. The full Mantiq project, however, was not just unrealistic; it was delusional. Fortunately, developing 3Lisp proved sufficiently tractable to serve as an adequate basis on which to obtain a doctoral degree.

4 Project

Numerous considerations recommend reflection as an area of study.

First, if a knowledge representation system (such as the envisaged Mantiq) is to be integrated, not only in the sense of not imposing a sharp procedural/declarative divide (such as by using different languages for each, in the manner of KRL), but also, relatedly, in allowing behaviour to be specified in virtue of

its explicit description, rather than just through bare imperatives, then the system's structures, operations, and behaviour must be able to be represented and described.

Moreover, if the reflective mechanisms are to be capable of modifying or affecting the system—rather than just being powerful enough to model or describe it, as for example ubiquitously demonstrated in incompleteness and uncomputability proofs—then those structures, operations, and behaviour must be represented *effectively*, in two importantly distinct senses of that critical term. For starters, throughout the course of the system's operation, what is the case about the system, not just in general, but specifically *what is going on at that very moment*, must be presentable to the system using the system's own descriptive/representational resources. That is, as I put it in Smith (1986), the system must be capable of on-the-fly *introspection*.[22] In addition, for the reflective deliberations to be of any use, they must be what I came to call *causally connected* in such a way as to support consequential intervention and action.[23] Whether one takes the use of the words literally or metaphorically, that is, a reflective system must be capable of self-directed "perception" and "action"—must act, in Searle's phrasing, in ways that honour both world-to-word and word-to-world norms[24] (though in both cases the "world," in 3Lisp's case, was restricted to the innards of the system itself in addition to associated mathematical entities). In sum: by itself, the goal of integrated control is strong enough to mandate effective,

---

[22]"Varieties of Self-Reference," Smith (1986).

[23]Another way to describe this is directionally, in terms of the ubiquitous metaphor that meta-level descriptions are "above" the base-level they describe—e.g., as reflected in the phrase "semantic ascent." A reflective system must implement an effective version of semantic ascent, or be able to "reflect upwards," as well as to implement an effective version of "semantic descent," or be able to "reflect downwards," by moving from a meta-level description of a possible state of itself to being in the state thereby described.

[24]Searle (1983).

self-referential access to a system's representational structures. So from the outset, it was clear that a genuinely integrated representational system must be reflective—must honor the reflective integrity criterion.

A second reason to focus on reflection, independent of considerations of integrated control, is more pragmatic. As all programmers know, such proto-reflective capabilities as quotation, disquotation and meta-level facilities, on both sides of the procedural/declarative divide (i.e., both in procedural programming languages and in such purely representational systems as formal logic), supply remarkable power at minimal cost. In Lisp's case, explicit access to the famed EVAL and APPLY, along with program quotation, are crucial to its ability to embody the "distilled kernel" aesthetic described above. Lisp's support for macros, similarly, allows the language to be used for its own extension,[25] increasing expressive power with little increase in core complexity.[26] Code optimizers, genetic algorithms, debuggers, etc., are behaviours that can all be considered to fall within this class.[27] And as mentioned earlier, if such capacities are ultimately going to be part of an integrated representational system, rather than build them in at the outset (which would be

---

[25]'Extension' not in the logical sense, but in the ordinary language sense of *allowing the system to be extended*—by adding features, defining new aspects and properties, etc.

[26]The notion of "increased expressive power" is informal but crucial. There are widely accepted proofs that, on the procedural side, any machine capable of implementing a Turing machine is "equally powerful," and on the descriptive or representational, that any system capable of expressing the truths of arithmetic is as "just as expressive" as that of any other language whose semantic structures can be arithmetically modeled. I support neither notion of equivalence, and reject their excessively-indiscriminate use of "equivalent power."

[27]The point is not that macros, debuggers, optimization routines, etc., are simple or distilled. Rather, it is that a simple, distilled kernel with quotational and other very simple meta-level capabilities (explicit EVAL in Lisp, for example) can provide a facility for defining all of these sorts of capabilities, without needing to complicate the kernel.

brittle and baroque), facilities for their integration should be part of the kernel architecture, so that they could be added in a theoretically and practically elegant way.

A third motivation for studying reflection is best framed theoretically. It has to do with the treatment of what is perhaps most generally subsumed under the notion of the "meaning" or "sense" or "intension,"[28] as opposed to the *denotation* or *extension*, of a procedural or declarative structure. Considerations of meaning and intension are ubiquitous in logic and philosophy of language, to deal with referential opacity, modal operators, quantificational contexts, variable binding, etc., where the semantical import or role of an occurrence of a term is other than simply to designate or point to its referent. Computer science has at least analogous contexts, where the computational significance of an expression differs from, or goes beyond, its ordinary denotation or "value." Most famous are the "left-hand side contexts" in assignment statements,[29] but there are other cases as well, such as in variable declarations, some forms of parameter passing (e.g., when the name of an array is passed as an argument to a function rather than the array as a whole), etc.

These intensional issues are relevant to reflection because of their intimate relation with quotation and semantic ascent. Lisp is especially free in using quotation to deal with intensional issues.[30] Cf. for example the way that LAMBDA terms are used to

---

[28]This is not the place to distinguish and disentangle the relations among these three terms. Arguments can be made that the notion of *intension* (as opposed to extension) is logic's partial attempt to deal with what, at least since Frege, philosophy of language has called the *sense* (as opposed to reference) of a natural language expression or thought.

[29]If x=2 and y=10, for example, the Python statement

    x = (x + y)/2

is an assignment statement that results in variable x being assigned the value 6; it is not a false claim that 2=6. Similarly, in

    array[7] = array[7] + 1

the right-hand-side occurrence of 'array[7]' is generally taken to be a

create closures in Scheme, which essentially involves quoting their bodies, or Lisp 1.5's use of APPLY, MAPCAR, etc., where the programmer must explicitly quote a functional expression in order to enable it to be applied to arguments in a different context, as for example in:[31,32]

```
(mapcar '(lambda (x)
           (+ x 1))
         '(10 20 30))              ⟹ (11 21 31)
(apply '(lambda (j k l)
           (* j (+ k l)))
         '(7 8 9))                 ⟹ 119
```

Even the simple case of setting a variable, which in many language is a primitive construct (e.g., $y = y + 1$), involves intensional issues—one reason why the construct bedevils initiate programmers, since the so-called "left-hand side value" (the variable 'y' in this case, in its occurrence on the left side of the equals sign) is, as it is said, "not evaluated." It is not incidental that the corresponding Lisp expression is:

```
(setq y (+ y 1))
```

where the 'Q' in 'SETQ' is short for "quote," the expression being an abbreviational equivalent to the more transparent:

---

reference to the *value* of the seventh element of the array named 'array', whereas the left-hand-side occurrence names that element, in such a way that execution of the statement can cause the value to be 1 greater than it was before that execution.

[30]As a logician might put it: dialects of Lisp prior to Scheme (and 2/3Lisp) used *hyperintensional* mechanisms to deal with intensional issues.

[31]These are Lisp 1.5 versions; in each case, the Scheme analogue would omit the quote mark before the «(lambda...)» term, but retain it before the second argument.

[32]Again, Python does not require the same use of quotation as Lisps, but approximately equivalent expressions would be:

```
list(map(lambda x: x +1, [ 10, 20, 30 ]))      ⟹ [ 11, 21, 31 ]
```
and:
```
(lambda j, k, l : j * (k + l))(7, 8, 9)      ⟹ 119
```

```
(set (quote y) (+ y 1))
```

or, as it is universally written,

```
(set 'y (+ y 1))
```

The general point is that whereas more complex languages may provide special constructs (assignment statements, special syntax for type declarations, etc.), Lisp's simplicity and "kernel calculus" character lead it to use quotation as a mechanism for treating sophisticated issues about meaning—a step, I believe, though it is never so described, towards dealing with *reference to meanings* (i.e., the ability to construct and use terms whose *extensions* are the *intensions* of other terms). Anyone proposing to deal seriously with intentionality (with a 't') and therefore anyone proposing to design a comprehensive representational system, thus needs a firm grasp of the relationship between quotation and self-reference, on the one hand, and fine-grained intensional (with an 's') contexts, on the other. Reflection is a great site in which to work such issues through.

A fourth reason to focus on reflection, though still technical, has a different flavour.

In the late 1960s and early 1970s, when 3Lisp was designed, it was common for researchers to construct AI systems to deal with simulated worlds—"toy worlds," as they disparagingly came to be known, because of the patently unrealistic simplicity of the simulations commonly used. To foreshadow concerns to be taken up later, one problem (of many) with these "toy-world" simulations was that they effectively "baked in" their designers' conception of their task domain's ontology, characteristically taken to be clean and unproblematic. My sympathies lay with a then-growing cohort in AI who felt that it was intellectually important—fundamental in order not to "cheat"—for an AI system to engage with its subject or task domain "for real," rather than merely "as imagined."

The problem, of course, is that dealing with external task domains "in vivo" typically requires engaging with sensors and effectors, perception and action, etc.—i.e., requires that one

take up real-world robotics, suffused in expense, challenges, and complexity. Taking the "internal structures" and "internal behaviour" of a computational system to be its represented subject matter thus allows a system to be *genuine* without requiring the resources of sensors, effectors, and general robotics. That is not to say that the structure of a calculus or virtual machine might not still be considered "toy," because of its extreme simplicity (and formal neatness), especially as compared to the external world. That simplicity has its advantages, though—it allows the theorist to pursue the architectural and semantic issues in greater depth than would likely be practicable were the base case or ground condition to be realistically rich. Having a task domain be simple and cheap, but still genuine, makes it an ideal initial site for exploring issues of representation, behaviour, and semantics.

A fifth reason to study reflection was different in flavour. Stemming from the human side of things, the motivation stemmed from a recognition of the importance of the self to the very notion of intelligence, and the corresponding urgency, for AI and cognitive science generally, of understanding it.

The view that the ability to reason about oneself must be an integral part of any system able to engage in full-fledged reasoning is widespread—from the Delphic "know thyself" injunction, to considerations of the phylogeny of the prefrontal cortex, to Popper's indelible "we think so that our hypotheses can die in our stead," to AI's recognition of the importance of metalevel reasoning in planning and error recovery. The recognition is so widespread that it need not be belaboured here, except to make one personal comment relevant to the reflection project.

I was suspicious, in the 1970s, and remain suspicious today, of excessive emphasis on explicit "self" versions of reference and consciousness, blinding one to what I take to be the most important fact about meaning and semantic directedness—that it can leap over time, space, connection, and even possibility, so as to be directed towards *alterity*, towards the world beyond, towards the wider reality in which we all live.

There is not the remotest sense in which the 3Lisp architecture, or the more general notions of reflection explored in this volume, probe questions of the depth required to do justice to all of these issues. But it is important to understand that the theoretical framing of the 3Lisp project, and to some extent the subsequent analysis of its capabilities, were influenced by this motivation. Even in the 1970s I believed a proposition to which I remain committed today: that if we develop technical models or analyses of concepts and behaviours fundamental to our humanity, we should do so with respect for the "real case," of which our technical systems are but simple, inchoate models. Perhaps the staggering advances in processing power witnessed in the intervening four decades, including present enthusiasms for various machine learning schemes and second wave AI, will make the point more obviously consequential than it was in the 1980s. However that goes, I believe that it is important to keep in mind, throughout the technical intricacies of the discussions of representational semantics, even in the arcane details of the technical papers included in the accompanying volume, that serious issues are at stake—much more serious than surface details might suggest.

There was yet a sixth motivation, relating the reflection project being examined here to the wider issue of the self as theorized in knowledge representation, AI, and cognitive science. In the 1970s, as this work was being undertaken, the topic of consciousness was just beginning to emerge from a long period of intellectual taboo (at least in scientific circles) into legitimacy and even prominence. Here, too, I want to make just one familiar point: that it was (and remains) popular, in discussions of consciousness and even of awareness, to point to *self-consciousness*—to the ability to be aware of being aware, to the capacity of engaging in meta-level reasoning, etc.—as being not just characteristic attributes of consciousness, but perhaps even defining, or at least definitive of what makes human consciousness special.

For reasons mentioned above, I found much of this focus on

the "self" aspects of consciousness excessive. Needless to say, self-awareness is decisive in rational, ethical, emotional life. Yet it was common, in the 1970s, to hear incautious exhortations not much more sophisticated than the following: "If only we could construct a system that was *aware of itself*—that could reason about *its own nature and behaviour*—why *then* we would have a system that was truly conscious!"

I doubted it. I have almost unbounded respect for what it would be to develop a system that can genuinely be said to think about *anything*—for all sorts of reasons, including many that figure in discussions of authentic and/or genuine intentionality. Similarly, though it is likely easier, I have the greatest appreciation of what it would be to devise a system that merits being called *aware*. But the *self* part made me uneasy—especially its fetishization. Sure enough, there can be something wondrous, and devilishly intricate, about the ability to step at least partially aside from one's own existence, in order to survey it along with all else that exists in the world. There is no denying its psychological importance, either—or underestimating the apparent difficulty in achieving detachment or dispassion in more profound form. But *architecturally*, the self did not seem to me to warrant the almost mystical awe in which it was held. The fundamental challenge for AI and cognitive science, I believed then—and continue to believe now—is not to be conscious of oneself, but to be *conscious of the world*.

An anecdote. In the late 1970s, a small group of philosophers and philosophically-minded cognitive scientists met from time to time in Cambridge, MA.[33] At one such meeting someone expressed the view that what would genuinely be challenging for artificial intelligence, and would almost be constitutive of achieving consciousness, would be to build a "self-aware" computer system, with the emphasis on the 'self'—one capable

---

[33]I have no clear sense of the membership, but I recall at least Ned Block, Dan Dennett, Jerry Fodor, David Israel, Dan Osherson, and Georges Rey being among the regular attendees.

of forming meta-level beliefs, desires and intentions about its own beliefs, desires and intentions, without restriction or limit.[34] I protested that this was patently untrue—that I could trivially build such a system, if someone would first just tell me how to build a system that could be aware of a *tree*. In fact, pace awareness, I had already constructed systems, I pressed on, with all the self-examining, self-referential, self-modifying properties that others were touting as the mark of consciousness—yet I was willing to guarantee, and would certainly have gone to court to argue, that these systems were *not in the remotest sense conscious of anything at all.*

   It would go too far to say that my subsequent study of reflection was intended to be deflationary, but it was definitely intended to be demystifying. The analogy with recursion is instructive. (Reflection bears more than a passing resemblance to recursion, though reflection is substantially more complex, in part, though by no means only, because of being semantically laden.[35]) To the uninitiated, recursion can seem mysterious,

---

[34]The "qualia" debate had not yet emerged—but even when such issues are taken into account, it is the qualitative character of being in the world that is challenging, in my book, not the additional case of qualitative character of oneself thinking or being aware.

[35]Recursion is sometimes described as involving *self-reference*, and so might seem like a semantically-laden notion, but I believe that the description of recursion as involving self-reference is an elementary semantical mistake.

From an extensional point of view, first, a recursive function is simply a mapping from a domain to a co-domain. There is nothing circular or looping, let alone referential, about it, *qua* mapping. Second, even intensionally, to assume for a moment that it is legitimate to talk about functions-in-intension, any looping or topological closedness is at most *mereological*—again not involving *reference* in any form. Issues of reference arise only when one *describes* or *denotes* or *represents* a function (recursive or otherwise). But in that case, the supposed "self-reference" is not *self*-reference—not reference *to the definition*. Rather, it is an embedded reference, within a (presumably compositional) representation of a complex whole, to that whole, rather than to a part of that whole—thereby violating a kind of

even somewhat magical. *Sans* underlying support, implementing recursive behaviour is somewhat complicated, as would be agreed by legions of low-level programmers who have struggled to ensure that re-entrant drivers, interrupt processors, and operating systems do not inadvertently trample on their own variables. And few are the students who find Church's Y-operator pellucid. Moreover, implementing efficient, seamless, transparent garbage-collecting memory management regimens, a premium in implementations of recursive behaviour, remains an active area of research.[36] Nevertheless, we *understand* recursion; virtually all contemporary high-level languages provide primitive support for it. No one is any longer theoretically perplexed; and programmers use it with abandon.

One of my original goals in studying reflection, by analogy, was to render descriptive and effective reflective (including self-referential) access as unproblematic as recursion, to enable it be provided by default in high-level languages, to unleash

---

mereological isomorphism that might otherwise be suggested by the governing compositionality. (Think about a technique employed in the days of early Lisp implementations of replacing the "recursive call" within a recursive procedure definition with a direct link back to the beginning of the procedure—making the definition, again, topologically closed or looping, thereby avoiding the overhead of a full procedure call, for a miniscule gain in efficiency.)

In sum, not only do recursive functions not refer to *themselves*; they do not refer *at all*. Functions are not semantic entities. The *definition* of a recursive function does not refer to itself, either; it contains, as a proper subpart, a reference to the function as a whole. To call a recursive definition *self-referential* is thus to commit an elementary use/mention confusion.

One can put the point even more simply. Extensionally, recursive functions involve neither self nor reference; they are merely mappings from domains to co-domains. Intensionally (or hyper-intensionally), *definitions* of recursive functions do involve *self*, but they do not involve self-*reference*; rather, they involve self-*use*. (The fact that we call recursive functions 'recursive' is an intensional or hyper-intensional fact about their finite representation.)

[36]E.g., in data purge algorithms in (so-called) solid-state disk drives.

programmers to use it without question—all in order to free us up to focus on more important things. Most of my published work, including not only the technical papers on reflection and 3Lisp, but also the more conceptual *On the Origin of Objects* and *The Promise of Artificial Intelligence* attest to that same orientation. As I have said several times, I firmly believe that the most difficult problems that AI and cognitive science must address do not have to do with self-reference and reflection at all, but with comprehending the world at large.

There is irony to the situation. In spite of spending years studying reflection, and developing something of a reputation, particularly in computer science, for being obsessed with self-reference, my deeper goal was to understand the self en route to *setting it aside*, as a theoretical problem, in order to focus on more challenging issues.

So that was the project: to explore issues reflection, self-reference, and substantial and effective self-description. Locally, the goal was to understand the topic, at least in a skeletal way, so as: (i) to clarify some of the promises that had lain implicit in Lisp and other programming languages and calculi that were able to "treat programs as data," (ii) to serve as an ingredient for a larger knowledge representation system; and (iii) if not quite to "solve" at least to advance our understanding of some of the formal/technical mystique surrounding consideration of the self. The approach also arose out of a sensibility: that in our theorizing about consciousness we stop fetishizing the self, and focus on the more substantive issue of building systems able to grasp their embedding world.

But topic, goal, and sensibility are not enough. Also required is an *idea*—an insight or guiding principle strong enough to allow these aims to be realized. That idea had to do with semantics.

5 Semantics

As I have said several times, reflection is an ineliminably semantic or intentional phenomenon. That is not to say that it is

passive, as if all that is required is reference or representation. As noted above, reflection in the sense I am considering requires effective, dynamic interaction between the reflective process and the system of which it is a part—self-directed analogues of action as well as perception. Notwithstanding these requirements of causal or effective engagement, however, the fundamental fact about reflection is of a system able to bear a semantic or intentional relationship to itself.

This stance of giving semantics the lead in any discussion of reflection is betrayed in the opening characterisation: that reflection is the ability of a system to *represent, reason about,* or *direct intentional action towards* its own structures, operations, and behaviour. Representation and "reasoning about" are paradigmatic (and paradigmatically) semantic phenomena—virtually defining of what semantics is. Directing intentional action towards something is by definition intentional; but even if the phrasing were simplified to 'deal with,' in the sense intended, the meaning would be the same. To "deal with" something is to be directed towards it—a caller at the door, a pressing email, an electrical problem. And a similar point can be made of other characterisations, such as the opening sentence in the (current) Wikipedia article on reflection, which describes it as "the ability of a process to *examine*…and *modify* its own structure and behavior."[37] Like 'reason' and 'represent,' the term 'examine' is semantically laden, denoting an intentional activity and orientation. One would not say that a bullet "examines" the window that it shatters, or that a magnetic field "examines" the objects within its compass in order to determine whether or not they will bend to its "will"—would not use intentional language, that is, when the phenomenon can be comprehensively explained using less conceptually demanding physical notions.

Like all semantic phenomena, as most obviously manifested

---

[37]"Reflective programming," Wikimedia Foundation, last modified September 8, 2024, 09:02 (UTC), https://en.wikipedia.org/wiki/Reflective_programming (emphasis added)

by reference, reflective activity is *oriented towards* some entity, event, phenomenon, domain, or subject matter (in reflection's case: some aspects of the system of which it is a part)—related to it through a "semantic arrow of intentional directedness," to put it in Brentanoesque terms.[38] This referential orientation, and more general semantical character, is underscored in informal discussion, and explains the almost automatic assumption that reflective reasoning must involve some form of "semantic ascent" or employment of meta-levels. In addition, as explained in the Introduction, and using terminology discussed in Smith 2019, it must also be *deferentially* oriented towards its subject matter. This normative requirement is fundamental. Notwithstanding their use to underwrite activity, change default behaviour, and the like, reflective descriptions, models, and even activities are *normatively accountable* to the system of which they are descriptions and models—towards which they are oriented. A reflective model or description, to put this baldly, should be *correct*, or at least descriptively or representationally adequate to the tasks in which it is being employed.

   This point about reflection's fundamentally semantic character seems so obvious as barely to deserve mentioning. It is certainly the case that discussions of reflection, in programming language circles as much as in logic, are drenched in semantical vocabulary: *self-models*, *meta-level reasoning*, *representations* of dynamic program state, etc. Reflection, it is said, involves "decomposing a system into a base-level and *one or more meta-levels that perform computation on the computation of the lower levels*," for example, to take a quote almost at random;[39] or "[c]omputational reflection…is defined as the activity performed by an agent when doing computations *about itself*.[40] Yet, as this book takes pains to argue, this surface agreement and use of common

---

[38]Or at least Brentano as interpreted by Chisholm, see Chisholm (1957). Also see McAllister (1974) and Dupuy (1994).
[39]Cazzola et al. (2000).
[40]Ancona et al. (1998).

semantical vocabulary mask profound underlying disagreement, having to do with what the word 'semantics' is understood to mean.

When I set out to design 3Lisp, I was innocent of the lurking semantical and terminological complexities. I based the design of the dialect on a conception of semantics that arose from my interest in knowledge representation and wider issues in cognitive science. Though I knew little logic at the time, the approach is also consonant with the classical (roughly realist) conception of logical semantics discussed in chapter 6. That is, without yet appreciating the profundity of the difference between them, I took 'semantics' to mean 'semantics-L,' not 'semantics-C.'

In particular, I took the requirement of distinguishing signs from what they signify—maps from territories—to be a minimum requirement on any system pretending to representational status. True, programming languages were not usually viewed as *representation* systems, but they are universally taken to be semantic—i.e., to comprise *symbolic structures subject to interpretation*—and so nothing seemed awry in approaching them from a representational vantage point. They involve the use of *numerals*, for example; the numerals surely designate *numbers*; and so on. Similarly for the other semantical terms with which computational discourse is rife: *names, identifiers, symbols, references, data, information*, etc. Theories of programming language semantics routinely map entities of all these sorts onto corresponding semantic values in what, at least on the surface, looks like the usual way (numerals onto numbers, memory addresses onto memory locations, etc.). And so it seemed innocuous, and certainly uncontentious, to embark on the project of designing a programming language, enriched with reflective capacities, that was based on a representational conception of semantics.

The details of the particular representational stance that I adopted were commonplace in the 1970s, though they now seem dated. In the dissertation I framed them this way, with reference to the overarching project of artificial intelligence

(note that the KRH is not about reflection, but about the underlying conception of computation and representation on top of which 3Lisp's model of reflection was based):

KRH Knowledge Representation Hypothesis (1982):
Any mechanically embodied intelligent process will be comprised of structural ingredients that (a) we as external observers naturally take to represent a propositional account of the knowledge that the overall process exhibits, and (b) independent of such external semantical attribution, play a formal but causal and essential role in engendering the behaviour that manifests that knowledge.

This was my attempt, in 1980, when 3Lisp was designed, to explicate the "formal symbol manipulation" conception of both computation and cognition that reigned throughout AI and cognitive science in the 1970s, in the heyday of its classical model (GOFAI). From a contemporary perspective, its formulation is problematic, *even if one continues to embrace a computational theory of mind*. For present purposes, what matters is the KRH's deferential attitude to semantics. I now believe that its claim that the mechanistic (formal, causal) operations are *independent* of the semantic attribution is too strong (false, in fact, of entities we take to be computers), but it certainly entails something I still believe: that there is *more* to the semantics than merely "what happens," mechanistically (formally, effectively, causally). And that "more" is enough to motivate the deferentiality of the stance: as explored in chapter 6, how things are (mechanically) treated is normatively governed by that semantical interpretation, with that non-effective interpretation having at least a degree of ontological and explanatory priority over the mechanically engendered behaviour.

For purposes of this book, and somewhat informally, I will label this deferential stance a **representational** model of computation—even if not necessarily restricted to be *formally* representational in the sense articulated in the KRH. There are hazards in using the term 'representation': it is used with a host

of different meanings in different fields; these days it is almost universally deprecated in contemporary discursive theorizing in all of them; and it is itself subject to the very same confusions that, as I will argue here, plague our understanding of semantics. As I argue elsewhere, however,[41] I find the deprecation excessive, blocking appreciation of profound insights into the nature of intentional mechanism. What matters most about representation here is merely that the ingredient structures of a representational computing system, as I use the phrase, are mandated, *in some way or other*, to bear a deferential non-effective semantic relation to that system's world or task domain.

A commitment to a representational model of computing was not, *per se*, the idea on which the proposed theory of reflection was based—the idea that I later came to feel had disappeared. But this representational model of computing was a critical precursor to and underlying assumption of that idea. Part of what rendered the reflection idea invisible was that an underlying deferential conception of semantics was something I simply *assumed*. It is not that I considered it unworthy of mention; the aim of articulating the KRH, above, was exactly to forestall any misunderstanding. But then, with that much said, I simply took a deferential understanding of non-effective representational semantics to constitute the theoretical background in terms of which the idea I wanted to express was framed.

The model of reflection, based on this assumption, was rather simple: the architectural form of a reflective system, I believed, was effectively dictated by two things:

1. The system's *nonreflective* architectural form—i.e., the system as it was defined for or used in normal, nonreflective situations; and

2. The overarching or governing semantical framework—

---

[41]See "Rehabilitating Representation" (Smith 1998), and also the 'Representational Mandate' in *Promise* (Smith 2019).

again, as framed so as to deal with the semantics of *non-reflective* representational structures.

That is, to make this as explicit as possible, far from being a mysterious or sophisticated add-on to a general (nonreflective) system, requiring devious additional design, the structure of a reflective system emerged, or anyway so at least I claimed, as a relatively straightforward consequence of two things: (i) its representational capacities; and (ii) its base-level architectural design. (The claim is a little strong—but there is something right about it—and it is that "something right" that I was struggling to convey.)

As this formulation makes evident, the substance or content of any given reflective system (the meaning or content of its reflective deliberations or states) is governed by the nature of the theory of self in terms of which the system found itself intelligible, acted upon itself, etc. A reflective system framed in terms of a modest theory of self (in terms of a modest "self-model," as some would put it) would, perforce, likely have modest reflective capacities. A system capable of perceiving, understanding, and acting upon itself in terms of a powerful, comprehensive theory of self would thereby be that much more *reflectively* powerful. Such a system, however, would need to be capable of powerful capacities for perceiving, understanding, and acting *in general*—which capacities would by my lights need to be available to be deployed in its mundane, base-level interactions with the world.[42] The nub of its reflectivity—the architectural intricacies required in order to make the system reflective, so that it could avoid tripping over itself in vicious loops, but nevertheless be sufficiently closely engaged with its own operations so as to be able to monitor the most acute particular details—i.e., the self-referential architectural structure that combined the

---

[42]The reason that those capacities would have to be available in non-reflective situations, rather than merely being likely to be available, stemmed from the aspects of reflection that transcended mere introspection. See chapter 7.

requisite degrees of attachment and detachment—*that*, I believed, was:

1. Relatively simple—certainly more complex than recursion, to repeat a point made earlier, but nevertheless something that could be understood once, demonstrated to be theoretically sound and pragmatically tractable, and then, as I put it above, "used with abandon";

2. Generic, in the sense that it could underwrite essentially arbitrary reflective systems, even if those systems differed substantively in having widely differing theories of self (just as the theory of recursion underwrites all manner of divers recursive function definitions); and

3. Effectively dictated, as stated above, by the combination of (i) its basic, nonreflective architecture, and (ii) the nature of the governing semantic/intentional theory.

Did I say as much in the dissertation? Perhaps not as summarily as I might have. The most compact statement occurs in a paper published in the *Principles of Programming Languages* conference (POPL) in 1984, where expression is given to a thesis that subsequently became something of the 3Lisp project mantra: that

RSS Reflection is simple to build on a semantically sound base

The phrase "semantically sound" in this statement does not mean 'sound' in the technical sense of formal logic,[43] but neither was it meant entirely informally—i.e., as saying no more than that a house should be built on sound foundations. Its specific content will be explained in chapter 8, having to do with the appropriate integration of declarative and procedural forms of content. One of the difficulties with the reception of 3Lisp had

---

[43]I.e., that derivable consequences of a set of premises are true just in case the premises are true. In the context, that would have been the natural interpretation, and so the wording was infelicitous.

to do with the fact that the very maxim on which it was formulated was itself subject to misinterpretation due to conflicts in underlying assumptions and vocabulary.

Nevertheless, RSS was always intended pragmatically. At a general level, the intuition was roughly this: that, because reflection is a semantic phenomenon, and because its intricacies can grow, in order to be intelligible and useful, it should be developed in a system whose base-level semantic framework is as clear and as simple as possible. Given that clear and simple base, the idea went, the ensuing reflective architecture would be simple enough to understand and to construct—i.e., would effectively "fall out" of the semantic framework, *once that semantic framework was properly understood*. Everything hinged on the words "properly understood." Whereas semantical awkwardness and complexity might be acceptable in a nonreflective system, the demands and strictures of reflection, or so anyway I believed, would amplify any ground-floor semantic infelicity to the point where the system would become unusably baroque and incomprehensibly complex.

Thesis RSS directly influenced the 3Lisp project. In spite of the reputed elegance of prior Lisps, including both Lisp 1.5 and Scheme, I did not feel that any of them were semantically clean enough for reflective purposes. A simple but particularly telling example in Lisp 1.5 and its descendants (such as MacLisp, the dialect in which 3Lisp was first implemented) has already been mentioned: their awkward use of quotation, in a dynamically-scoped language, to mimic higher-order behaviour—an awkwardness with which programmers had lived until the deficiency was repaired in Scheme. But there were other problems, detailed here, that in my judgment had not been repaired in any existing Lisps, including Scheme.

Perhaps the barest possible example of these problems, again already mentioned, is illustrated in the fact that in Scheme, as well as in all prior Lisps, the expression «(+ 1 (QUOTE 2))» evaluates to 3. This in spite of the fact that, at least to anyone of a representational bent, the expression seems ill-formed—a malformed attempt to add a number to a numeral.

And if one's basic semantic framework is confused on issues of *use* and *mention* (philosophical terms for the distinction between referring to something and referring to a representation of or to a reference to that thing), then the chances of a comprehensible reflective architecture being developed on top of it struck me as minimal to nil.

As detailed in the original 3Lisp papers, therefore, my strategy involved the presentation and analysis of three consecutive Lisp dialects. The first, dubbed 1Lisp, introduced purely for comparison purposes, was stipulated to be a distillation of, but was really just a name for, all prior Lisps, including not only McCarthy's original Lisp 1.5, but all subsequent versions—including, crucially, the lexically-scoped, higher-order Scheme. The second system, 2Lisp, was a dialect I called **semantically rationalized**, in which the aforementioned semantic infelicities were "corrected"—so that, among other things, «(+ 1 '2)» generates a type error. Details are provided in *Legacy*; for now it is enough to note that this semantical cleansing operation required replacing the default *evaluation* regimen that underlies other Lisps (except M-Lisp) with one of *normalization*, a form of simplification or term-rewriting, reminiscent of α- and β-reduction in the λ-calculus, which reduces expressions to canonical co-designating terms—a move that requires separating out (i) the notion of a symbolic expression's value or reference or denotation, and (ii) what happens to that expression when it is subjected to computational processing.[44] Making this separation, which turns out to lie at the nub of the misunderstanding of computing to be unraveled here, led 2Lisp, I felt, to be a cleaner version of Lisp than any that existed before, including Scheme.

The proposed model of procedural reflection was then demonstrated in the third dialect, 3Lisp, defined as a reflective

---

[44]Terms used to refer to "what happens to an expression when it is processed" vary by context: "evaluation" in functional languages, "inferential role" in logic, etc.

variant of 2Lisp, inheriting its alleged semantic cleanliness. The theoretical point was straightforward: the coherence and simplicity of the 3Lisp reflective model depended fundamentally on the semantic cleanliness of the rationalized version (2Lisp) on which it was based.[45]

There was to have been a fourth dialect, en route to Mantiq, called 4Lisp. From a representational point of view, 3Lisp was intended to (and indeed did) deal solely with *internal* issues: quotation, self-reference, and effective internal description— i.e., with what I call an "introspective" sense of reflection. While remaining fully procedural (though within an overarching representational conception, as always), 4Lisp was to meet two additional goals.

First, 4Lisp was to extend the semantic rationalization and semantic cleanliness of 2Lisp and 3Lisp to *data structures*, used in programs to refer to or represent objects, events, states of affairs, and the like, in the program's task domain—paradigmatically located in the world outside the machine (though of course data structures are used to represent internal entities and states of affairs as well). 2Lisp and 3Lisp provided some elementary data structures, but only "formally," as it were. With respect to the critical issue of (representational, and thus noneffective) semantic interpretation, beyond the obviously critical ability to refer to programs themselves, only simple mathematical entities were handled in 3Lisp: numbers, functions, sequences, boolean truth values, etc.—plus, in what was in effect a semantic "cheat," uninterpreted atomic symbols (traditional Lisp "atoms").

Needless to say, there is a sense in which the 2Lisp and 3Lisp

---

[45]A reflective 1Lisp, I believe, would have been a mess. Strikingly, all subsequent reflective dialects of which I am aware, with the possible exception of Muller's M-Lisp, are in fact just that: attempts to meld reflective capacities into/onto languages that are not designed in terms of a semantic model in terms of which reflection can (in my view) be defined.

data structures are computationally "complete."[46] A 2Lisp or 3Lisp programmer can readily construct vectors consisting of atomic identifiers, and other such usual forms; it is simple enough, in turn, to "implement" any other data structure one pleases out of them, such as a vector consisting of the capital cities of all the countries in the British Commonwealth. But that was *exactly not the point*. The whole idea undergirding 2Lisp and 3Lisp, in contradistinction to standard data structural practice, was to provide a programming language that, like logic, came *complete with (representational) semantical interpretation* (in the semantic-L sense). And the entities in the real-world task domain of a 3Lisp program, which would form the natural domain of interpretation of its data structures, lay beyond the reach of the semantical theory governing 3Lisp (or, for that matter, any other programming language at the time).

So the first half of the 4Lisp mandate was to provide full support—formal structures plus semantical-L interpretation, as in logic[47]—for representing concrete real-world entities: objects, events, situations, states of affairs, etc., together with the properties and relations that apply to them. Once this was in place, a second part of the mandate came into view: for 4Lisp to extend the 3Lisp reflective model so as to be able to represent and reason not only about its internal operations and structures, but also *about its relationship to and engagement with that*

---

[46]Computational completeness has nothing to do with the notion of completeness in logic; rather, it has to do with a form of mechanical/behavioral equivalence, according to a very broad metric of equivalence.

[47]As in logic, the language designer could not possibly provide the actual interpretation for all use cases. I imagined that 4Lisp would provide categorical constraints on interpretation (entities of a given structural class would denote or represent elements of the semantic domain of a corresponding type), with the actual interpretation function supplied, in the absence of semantics-determining robotic engagement with the world, by the user. The resulting structure would set up the preconditions for the language definition and implementation needing to be *sound*, rather than correct.

task domain (covering not only input-output and other phenomena at the liminal boundary, but also those states of affairs beyond the boundary normatively relevant to how and why it behaved as it did).[48] That is, in terms of the vocabulary of "Varieties of Self-Reference" (1986), 4Lisp was to move beyond mere introspection into the genuine world of reflection.

Like Mantiq, 4Lisp never came to pass. Among other things, its development would have required the prior overhaul of computer science recommended in chapter 8.

6 Reception

The dissertation on 3Lisp and reflection was completed in 1981, and two papers published in 1984 (both papers and excerpts from the dissertation are included in *Legacy*). As suggested earlier, the reception was more positive than any freshly minted graduate student could reasonably have expected. Four people deserve special appreciation: (i) Jim des Rivières, first and foremost, to whom this book is in part dedicated, who moved to Xerox PARC to help develop 3Lisp, co-wrote the paper on implementing reflection,[49] and developed an impressively-capable run-time 3Lisp compiler—someone who earned my enduring thanks, who tragically passed away before this book was completed; (ii) Dan Friedman, who at the 1984 Principles of Programming Languages conference (POPL) commented positively on the 3Lisp work, and who together with

---

[48]To put this another way, 4Lisp was to be designed to represent, among other things, all those external-world phenomena and states of affairs that would be adduced in an intentionally adequate explanatory theory—i.e., a theory that explained the semantics of and normative standards applicable to its structures and behavior—not just the array of causally-effective circumstances that caused such behavior to come into place.

[49]Jim des Rivières and Brian Cantwell Smith, "The Implementation of Procedurally Reflective Languages," in *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and Functional Programming* (Association for Computing Machinery, 1984), 331–347.

his colleague Mitchell Wand went on to publish a number of important papers on reflection and related topics; and (iii) Mario Tokoro and Akinori Yonezawa,[50] long-term supporters of reflection in computer science, who over the next decade played critical roles in forging and supporting the reflection community.[51]

Over the following decade a small community developed, additional papers were published, and a series of international workshops held (on reflection, first, and then on reflection and meta-level programming). Some other reflective languages were defined, notably including Friedman's "Brown," Danvy's "Blond," and Muller's M-Lisp; and theoretical analyses proposed such as Friedman & Wand (1988), Demers & Malenfant (1995), and Muller (1992).

Even more broadly, it is probably fair to say (as mentioned earlier), the notion of reflection—or at least some notion of reflection—has at least to some degree been made "safe" for programmers and theorists alike. That is not to say that the theoretical work is done—or, in my judgment, even seriously enjoined. In spite of a number of efforts, suggested above, I do not believe an adequate theory of reflection has yet been developed (by me or anyone else), or even that there is any accepted understanding of what such a theory would involve—in particular, as argued throughout this book, of what reflection even *is*, such that we might have a theory of it. In large part, I believe, the reason for this lack of theoretical understanding rests on the underlying confusion about the notion of reflection itself, based in turn on misunderstanding as to the nature of non-effective semantics in terms of which 3Lisp was framed, and underlying that about the nature of computation itself—the topic to which this book is ultimately addressed. Still, at least in broad outline,

---

[50]Then at Keio Univ; now Chairman and Founder of Sony Computer Science Laboratories.

[51]E.g., the IMSA92 International Workshop on Reflection and Meta-Level Architectures held in Tokyo, Nov. 4–7, 1992.

in the sense that the notion has come to be understood, the notion of reflection is no longer viewed as mysterious or even very suspicious.

In this vein, it is sometimes noted that the possibility of reflection was intrinsically present in the original days of machine-language programming, since in Von Neumann architectures the binary representation of a program is no different in kind, and no less amenable to access and modification, than that of any other form of data. Subsequent higher-level languages, however, tended to inscribe a line between the two, marking the *program* part of the machine as "off limits" with respect to dynamic modification, in part to facilitate compilation and other forms of program analysis and optimization. Since the 1980s, however, following the development of 3Lisp and other (reputedly) reflective dialects, and the demonstration of rudimentary reflective capacities in such much more mainstream languages as Smalltalk, it has become almost the norm for at least some reflective capacity to be provided in contemporary languages. Ruby, Java, and Python, for example, languages still in common use, provide some reflective facility. And so while reflection is nothing like as ubiquitous as recursion (though at the same time arguably more powerful, and admittedly more challenging to understand and to implement), one can reasonably say that it has taken up a pragmatically stable place, alongside other constructs, in the contemporary programming language designer's toolkit.

Why then not count the 3Lisp work as more of a success? The reason rests on a single point made above. The intent of the original exercise was not to demonstrate that a simple reflective system could be built and even used, but to make the theoretical point encapsulated (but by no means explicated) in the RSS mantra: that, *if one gets the semantics straight*, then reflection can be designed and implemented in a straightforward way. Preparing that cleaned-up semantical ground within the realm of functional programming languages was the point of 2Lisp. And it was not just the 2Lisp dialect itself, or its conception of semantical cleanliness, but the very idea of providing

(what I would consider to be) a semantically clean basis on which to build, and even the recognition that reflection is itself a fundamentally semantical or intentional notion, and thus, in my view, any real sense of what a reflective architecture is or would be—all of these things, subsequent to the publication of the dissertation and ensuing 3Lisp papers, vanished into thin air.

Why? That is the first mystery that we need to unravel.

3 Programs
Diagnosis · First Pass

Few ideas are as central to our understanding of computation as that of a *program*—a "set of instructions" directing an underlying machine to perform a set or sequence of operations. The idea is often associated with von Neumann's conception of "stored program" machines, in which programs are stored in the same type of memory as the data on which the program operates. This is the architecture of virtually every computer in existence today. But the concept is older, having developed in tandem with that of computation itself. It builds on a longstanding idea of using a mechanical apparatus to control a machine's behaviour, perhaps most famously in the design of the 19th century Jacquard loom. In his 1937 paper Turing used this idea to arrange for one portion of the computer memory to control the operations of the machine in his development of the concept of a "universal machine."

Programs are the paradigmatic subject of semantic-c analysis in computer science. On the face of it, it is not obvious why programs should be given higher priority than data structures, with respect to semantic analysis, since data structures and other non-program elements are the primary vehicles used to represent and carry information about programs' task domains—people, institutions, mathematical entities, machine-internal states of affairs (the length of lists, whether data has been updated, etc.), and so forth. On the surface—though I will question this below—the representational status of data structures might seem, at least in some senses, simpler than that of

programs. Nevertheless, the semantics of data structures is not a staple area of inquiry in computer science (though semantic analyses are applied in analyses of knowledge representation systems and databases). Rather, it is *programs*—or, significantly, *programming languages*—that form the centerpiece of the semantic analysis of computations.

Programs are semantically challenging in part because they exhibit two very different kinds of properties. The relation between them illustrates the fundamental meaning/mechanism dialectic in analytic focus throughout this book.

First, on the mechanism side, programs must be **effective**—capable of leading the machines on which they are implemented to behave in certain ways. I will eventually argue that effectiveness may be the most important notion in all of computer science. For now, though, and in rough terms, we can simply take it to mean that programs must have mechanical or "formal" properties that have a causal or in some other way determinative influence on the behaviour of the machines they are implemented on.[1]

Second, on the meaning side, and still speaking informally, programs are also *semantically interpreted*: taken to be meaningful (approximately linguistic) expressions that represent or are "about" the computations they effectively produce. Just how that aboutness works, and how it should be understood, will occupy us here for several chapters.

Effectively engendering something and semantically representing it are not the same thing. Spark plugs engender explosions without representing them. This is why they are analysed as pure mechanisms, rather than as signs or symbols. In

[1]Computer science has the notion of *effectively computable*, which is introduced by ostension and widely agreed upon, but there are no attempts to define what is or is not effective. See for example the definition from Daintith and Wright (2008) in footnote 17 of chapter 1. See Minsky (1967) for a more nuanced treatment of mechanism, which nevertheless is still more motivational examples than rigorous definition.

contrast, engines can be represented by numerous things—models, specifications, descriptions, blueprints, etc.—but in general it is not the functional role of such entities to cause to come into existence that which they represent. Again, on the mechanism side, the sun exerts effective control over, but does not represent, the orbits of the planets, whereas, on the meaning side, a paper expressing Newton's theory of orbital mechanics represents planetary motion, without exerting causal or effective influence on it.

The fact that programs cause the behaviour they represent is explicable neither purely semantically nor purely mechanically. Suppose we arbitrarily divide a computer's memory into two parts: p and $\bar{p}$. And suppose the bits in portion p are configured into some arrangement α_p, and the bits in $\bar{p}$ into arrangement $\alpha_{\bar{p}}$ (informally, α_p would be called the "contents" of portion p, and $\alpha_{\bar{p}}$ the contents of $\bar{p}$[2]). In general, arrangement α_p will exert an effective influence on the behaviour of the whole, since if α_p is changed, and $\alpha_{\bar{p}}$ held constant, the behaviour of the whole system is liable to change as well. But that fact alone is not enough to warrant calling p a *program*. By the same token, if p is held constant, and $\bar{p}$ changed, then again, at least in general, the behaviour of the whole will adjust—but that is not sufficient warrant for calling $\bar{p}$ the "data structures" over which p operates. The problem is that, from what has been said so far, the situation regarding p and $\bar{p}$, and α_p, and $\alpha_{\bar{p}}$ is entirely symmetric; so far we have no warrant for identifying one as program and the other as data structures. To count as a program, a configuration α_p must be able to be semantically interpreted as being *about the behaviour that it causes*. And by the same token, to label $\alpha_{\bar{p}}$ *data structures* requires that there be data that $\alpha_{\bar{p}}$ encodes—where data, similarly, in order to be data, must be

---

[2]'Contents' in the sense of the configuration of bits in that portion. Like everything in this realm, 'content' is a tricky word, having both a purely mechanical meaning (as in "the contents of this cup") and also a semantical one ("the content of what he said," meaning the meaning or substance).

meaningful, must be about something.

I will push hard, in this book, on what it is for programs to be *about the behaviour they cause*. At a general level, however, the fact that programs necessarily exhibit both effective and representational properties makes them an ideal site for the exploration of an ontological and epistemic issue of extraordinary generality—with ramifications far beyond computing. Since the Scientific Revolution and the rise of "mechanical philosophy," scientific theories have embraced causal explanations of material phenomena. This strategy has proved so successful that science is now often equated with "causal explanation." Historically, however, it has not been obvious that semantic or intentional phenomena, including logic, language, mind, and other meaningful phenomena—phenomena in what I am calling the **realm of significance**—fall within the scope of scientific explicability. For reasons to be explored here, there is good reason to doubt that they will ever be explicable purely in terms of the generalized "bumping and shoving" world of material causation.³

Needless to say, none of this is to claim that significant phenomena are *non-physical*, in a dualistic sense, according to some spooky metaphysics.⁴ Many if not most signs and symbols are

---

³By "the bumping and shoving" world of material causation I mean to include everything in current (and future) physical theory. Newton famously noted that neither gravity or magnetism worked by "bumping and shoving," in any sense of those terms used in his day, leading him to doubt that either were physical or mechanical forces at all (Newton, 1687). Our notions of physical force have developed, however, and especially with the recent LIGO results (laser interferometer gravitational-wave observatories), magnetism and gravity have been subsumed into physics' conception of the world—and so it is now unremarked to consider magnetism and gravity as kinds of "bumping and shoving." At issue here is whether representation and other semantic-L relations (including reference) will prove explicable within anything like what is currently understood as scientific theory.

⁴In particular, this statement is not by itself intended to deny what is called "global physical supervenience"—the idea that the total physical

plainly physical in their concrete occurrence. This is evidently true of oral language, written language, and computer data structures (though abstract and mathematical entities may also be interpreted semantically). Physical objects and processes are employed in scientific theories, as well as being represented by them. At issue, rather, is whether the analysis of the causal, physical, material, or effective (mechanical) dimension of signs, symbols, and other significant or intentional entities, especially their local causal, physical, material, or effective dimensions, is adequate for explaining what it is about them that makes them meaningful, significant, intentional.

It can be argued, I believe, that whether intentional or significant phenomena can be explained in causal (mechanical, physical, effective) terms is among the most fundamental challenges permeating current intellectual inquiry. It comes down to a question of whether the realm of meaning, interpretation, language, representation, significance, etc. can be *explained by science as we know it*—whether they can be *naturalized*, to put it in philosophical jargon; whether they can be *reduced to the physical*, to put it informally.[5] The issue bears on the power of neuroscience to explain the mind, on the explanatory potential of evolutionary psychology, on the prospects of building genuinely intelligent artificial intelligence systems, and on myriad other issues. While this book is not directly addressed at such large-scale questions, the discussion will be relevant to them for the following reason: many contemporary discussions of these

---

state of the universe, at the lowest and most fundamental level of physics, is sufficient to determine the state of everything in that universe, including its intentional phenomena.

[5]In saying "reduced to the physical" I do not mean to take a stand on specific forms of relation to the physical world—e.g., of the sort explored in philosophical discussions of the relations among type-reduction, token-reduction, supervenience, etc. My aim here is very general: to get at whether causal explanations, of the sort familiar from science (including discussions of emergent properties) will suffice to get at what matters about intentionality and significance.

issues rest, more or less explicitly, on presuppositions about computing and the explanatory frameworks we use to understand it.

In particular, there are many who take computer science to have demonstrated the adequacy of mechanical explanation for at least some intentional phenomena. If computation deals with intentional phenomena, which is certainly arguable, and if computer science deserves its label as a 'science,' which it certainly claims, then surely, such people reason, that must demonstrate that intentionality, in at least some sense, has been "rendered safe for scientific explanation."

In this book I will argue against that conclusion. On the first step, I agree: computation, at least what I have called "computation in the wild," does in fact deal with intentionality—and necessarily so. That is implicit in what I am calling the fundamental dialectic, between meaning and mechanism, where by *meaning* I include the realm of the intentional. It is the second step of the argument that is problematic. I will not only argue that computer science, in anything like its present guise, is incapable of explaining intentionality, but more strongly, that for that reason computer science, as presently constituted, *is incapable of explaining computing as computing*. As a result, the contemporary state of computer science cannot, in my view, be used to buttress any arguments that scientific explanation as we know it today—including genetics, neuroscience, evolutionary psychology, and artificial intelligence[6]—is capable of providing a foundation for understanding the realm of meaning, mattering, and significance.

I will not discuss these larger considerations further here, but their existence in the background underlines why it is so important to examine programs' constitutive exemplification of properties on both sides of the "meaning/mechanism" divide.

---

[6]At least artificial intelligence as currently theorized; see *Promise*.

1 Diagnoses

It quickly became clear, following the publication of the initial 2/3Lisp papers, which aimed for rigour in differentiating mechanism and meaning, that the theoretical framework underlying 2Lisp, and hence the model of reflection in terms of which 3Lisp was defined, was invisible or opaque to traditional programmers.

Over the years I entertained several hypotheses about the reasons for this epistemic blindness. My first diagnosis, discussed in the original 3Lisp publications, was that it arose from a conflict between two competing understandings of the nature of a *program*—a divergence of opinion on what programs are, what it is for a program to mean something or have semantic value, what "program interpretation" refers to, etc. It was clear, even at the outset, that the two readings of program were associated with divergent understandings of semantics, but at the time I felt that the incompatibilities in how programs were conceived had ontological or explanatory priority, with the differing conceptions of semantics following from it. Over time, in what I will here call a "second diagnosis," I came to feel that the semantical differences cut deeper, and that the discrepant notions of program reflected more foundational differences in understandings of the nature of computation as a whole—on what it is to have a semantic value in the first place, on the relation between programs and processes, and on other structural assumptions about the nature of the subject matter.

In the end, neither diagnosis suffices. Whether taken singly, or together, they are ultimately unable to explain the divergence in understanding that 3Lisp revealed. Rather, or so at least I will argue, both diagnoses are manifestations of instabilities at an even more basic level, having to do with metaphysical and methodological presumptions on which the entire field of computer science is founded—something I call the "third diagnosis."

This chapter attempts to motivate and explicate the first diagnosis: the two readings of the notion of a program. Though the result is ultimately unsatisfactory, articulating the two

conceptions is instructive, not only archeologically, regarding the history of the misunderstanding, but also substantively. Just one of the conceptions, it turns out, is conceptually compatible with the idea of a *reflective program* in the sense articulated in the Introduction, to which the 3Lisp project was dedicated: a program capable of reasoning about and representing the dynamic evolution of the program as it runs. Needless to say, this is the view on which 2Lisp and 3Lisp are based. It is the other conception, however, *incompatible* with reflection so described, that is the received view in computer science. This divergence goes some ways towards explaining the conceptual confusion that 3Lisp provoked.[7]

The second diagnosis is examined in chapters 4 and 5. Among other things, those chapters analyse the extent to which the second diagnosis predicts the first. Those two chapters also push on some unexamined but recognized oddities in programming language design. But though the second diagnosis cuts deeper, it too fails to lead to resolution. The reasons why the situation has never been sorted out not only reach deep into the foundations of computing, but into our basic ontological grasp of the world. In addition, the comparative analysis in chapter 5 also reveals that the workaday understanding of programmers parts company, in telling ways, with the understanding embodied in current theoretical computer science *and* with the conception articulated in the two diagnoses. This is one of the reasons why doing justice to the intentional character of computing will require a major overhaul of our entire ontological-

---

[7]As noted above, and explored in §4.VI, below, philosophical accounts of computing generally ignore the notion of a program—both the 'formal symbol manipulation' view as espoused in classical philosophy of mind, and the "pure mechanism" view of such contemporary writers as Floridi, Piccinini, etc. (Searle's "Minds, Brains, and Programs" (1980) and *Minds, Brains, and Science* (1984) talk of "programs," but he uses the term for computing as a whole, not making a distinction between that part of the computation that is a program and what a programmer would call data structures.)

cum-semantical-cum-mechanistic imagination.[8]

The third diagnosis starts out by setting readings of specific concepts aside, including both *program* and *semantics*, and instead assesses the situation from a perspective dealing explicitly with metaphysical assumptions undergirding present-day theoretical computer science. I argue in chapter 7 that the field is implicitly but resolutely committed to what I call **blanket mechanism**—a reframing of concepts originating in logic and philosophy to fit within a strict, narrow version of the philosophy of mechanism. To get there, we need to back up, in chapter 6, to review the intellectual foundations of logic, on which so much of our understanding of computing is based—especially historically. There are profound lessons in that historical understanding, I believe—fundamental to any deep understanding of intentionality, reflection, and the meaning/mechanism dialectic—that have unfortunately been lost in contemporary computer science.

In chapter 8 I sketch some preliminary requirements on what it will take to resolve the situation. It is well beyond the ambition of this book to develop the particulars of the solution; spelling out such an approach in detail will be an enormous undertaking. But I can at least sketch some of what will be involved, and describe some of the ingredients that will have to be included in any proposed solution.

The first *goal*, though, is to explain why the present state is inadequate.

Note to the reader: Since the first and second diagnoses are ultimately rejected, there is a sense in which the remainder of this chapter, along with chapters 4 and 5, could be omitted by a reader looking merely for the position that the book argues for, *rather than those that it argues* against. *The genealogy of the two rejected views,*

---

[8]See "The Correspondence Continuum" (Smith, 1987) for an inchoate analysis, and *On the Origin of Objects* (Smith, 1996) for a sketch of an alternative view.

however, reveals numerous nuances that are dealt with in the third diagnosis, so those optional sections are nevertheless recommended.

2 Two views of programs

The two views of program were explicitly distinguished in the original 3Lisp papers. The one that figures in computer science's theoretical analyses I called **specificational**, though I might now have chosen *prescriptive*. The other, which I labeled **ingrediential**, though a more illuminatingly term might have been *representational*, is more strongly connected to logic and artificial intelligence. As I will show, the specificational (prescriptive) view is procedurally powerful but semantically unclear. The ingrediential (representational) is semantically clearer, but procedurally inadequate. As argued in chapter 4, neither does justice to the intuitive understanding on which all programmers implicitly rely. Moreover—to repeat a standard refrain—the problem is not merely that contemporary conceptions are inadequate for purposes of reflection. Rather, analysing reflection forces us to recognize that all reigning formulations of program are inadequate.

More seriously, to up the ante, even when the inadequacies are exposed and explained, no straightforward solution presents itself. Not only does the field not currently have a conception of program, I argue, that is both semantically and procedurally adequate (i.e., adequate in terms of both meaning and mechanism), but what it would be to have such an account outstrips our present imagination. The resulting situation may tempt readers to throw up one's hands, or even to embrace the morass. But "Lo! It is imbricated!" is hardly helpful. The aim of chapters 7 and 8 is to point in a more fruitful direction.

2a Ingrediential View

2Lisp and 3Lisp were designed according to the ingrediential (representational) view, according to which programs are understood to be *effective constituents within the computational processes that arise from running them*. The view takes computations to consist of complex symbolic structures manipulated by

formal processes, much in the way that symbols in logical formulae are assumed to be manipulated by a logical inference or automatic deduction system, and (derivatively) in the way that philosophy of mind has classically understood computation—e.g., in classical philosophical descriptions of the "computational" theory of mind.

On the face of it, the ingrediential view makes no reference to a program/data-structure distinction—giving the conception something of a Von Neumann style. Programs are assumed to be dynamic computational ingredients, ontologically and referentially on a par, pace the question of task domain, with anything else "interior" to the computation—including all data structures. As with all symbolic structures, "referring terms" within programs—variables, identifiers, functional expressions, and the like—are assumed to be mapped by a "semantic interpretation function" onto (what philosophy of logic and language would take to be) their *referents* or *denotations*. As always, this interpretation is assumed to be *deferential*. The "formal" rules governing the symbolic transitions that are instantiated by running the computational process, that is, are not assumed to float free—"like frictionless wheels in the void," in McDowell's phrase. Instead, they are taken to be normatively governed by the deferential semantics.[9]

Moreover, and crucially—as suggested in the Introduction—the deferential semantic relation need not be, and almost always will not be, *effective*. In this logical/referential sense, that is, as explained more thoroughly in chapter 6, semantic interpretation simply "holds" or "obtains." It is not *active*, not a *process*, even if reference and interpretation are profoundly dynamic in the sense of *varying in time and through circumstance* (and even if, perhaps in a Wittgensteinian, Vygotskian, or Whiteheadian spirit, one were to argue that it is *processes* that

---

[9]The challenge raised above about how programs are distinguished from data structures is at least partly addressed in 2/3Lisp by the norms governing how programs are processed and what they represent; see §4.V.4.

have such semantic interpretations, and/or that the semantic interpretations are themselves processes, as might be the use of the language or tools).[10] It does not take time or energy, or require execution of a process, for the planet we call Venus to be the referent of the phrase 'the morning star,' for 'Mount Logan' to name the highest mountain in Canada, for the name 'al-Khwārizmī' to refer to a long-deceased Persian mathematician in whose debt we remain to this day. Even if the meaning of the term 'hod' is the use of a form of wooden container to convey bricks to bricklayers, it does not take time for that to be the term's referent (even if it takes time for a person to say or understand it).

Needless to say, this is not the computer scientist's notion of "interpreting a program," where that term is understood to be a gloss on what it is for the program to be *executed* or *run*. This alternative reading relies on the second, specificational or prescriptive reading, about which more presently (§2b). Even more profoundly, as explored in chapter 7, that conception relies not only on the specificational reading, but takes that reading (i.e., the program) to be neither more nor less than a causal precursor to the behaviour that results from executing it.

On the ingrediential view with which we are starting, in contrast, procedurally processing an expression (a "program," a "code fragment," etc.) would not in general be expected to involve effective (computational, mechanical) access to the

---

[10]In chapter 8 I will discuss semantical approaches in the spirit of Wittgenstein's injunction that "meaning is use," where the use is active, but that is a different issue. The present point is merely that, if β is the meaning of α, the connection between α and β is not active, not a process, not something that takes time (no matter how much β, or α, is itself an action or activity). Even if the meaning of a hammer is how it is used, it still does not take *time* for that to be what it means; it is simply *the case*. Similarly, if I describe a spiral staircase by moving my hand in a helical gesture, and the dynamic path of my hand is the signifier of the shape of the stairway, the *relation* between the active path and the shape is, again, not *itself* a process; once again, it merely obtains.

referent, which might be abstract (e.g., a real number), exterior to the machine (a mountain), or even non-existent (an imagined entity or hypothesized occurrence). Rather, as these examples suggest, and as more fully explored in chapter 7, the semantical interpretation of a symbol, in this context, is by default taken, as in natural language:

1. At least in the general case, to be determined to *be* the referent or interpretation by wider circumstantial facts, perhaps including the user's intentions or situations of use, but in any case (at least in general) not merely by local facts about the symbol's use;

2. To be established *as* the interpretation, at least in an ontological or explanatory if not necessarily temporal sense, prior to any issue of processing (the grounds of the deference); and thereby

3. To figure in establishing the *norms* that the procedural regimen is obliged to honour—the norms to which the regimen must defer—in the course of the symbol's treatment.

Terms in programs ingredientially conceived are most likely to denote or designate entities[11] in the program's (external) task domain.[12] The ingredient view thus fits naturally with a representational conception of computation discussed in the last chapter. In the following definition of factorial, for example, the variable «n» would be assumed to denote an abstract (perhaps Platonic) integer:

---

[11]At the moment I am not distinguishing 'names,' 'refers to,' 'denotes,' 'designates,' 'represents' or (most generally) 'has a semantic-L interpretation.' At stake is the general conception of semantics-L within which distinctions between and among these varieties could be drawn.

[12]This assumption especially likely to be applicable to functional languages, such as 2/3Lisp, but the same would be true for object-oriented systems, whose object types or classes are most paradigmatically defined in terms of the most salient ontological categories of the task domain.

```
(if (= n 1)
     1
     (* n (factorial (− n 1)))) 13
```

Or to take an only somewhat more complex example, imagine that a routine, called highest-paid-employee, given an identifier of an institution, is designed to return an identifier of the highest-paid employee at that institution. Thus it might be called in an expression of the following sort:

```
(highest-paid-employee institution-27)
```

The symbol «institution-27», in this expression, whether constant or variable, would be assumed, on this representational interpretation of the ingrediential view, to designate—name, denote, refer to—the real-world institution in question, such as Duke University; the whole complex term, the highest-paid employee of that institution (President, Dean of the medical school, basketball coach, whatever).[14] Procedurally processing the expression (*evaluating* or *executing* it, as is commonly said, though on this ingrediential or representational conception of program I believe that a phrase designating a form of term-rewriting would be more appropriate, such as the 'normalizing' used in 2/3Lisp) would invoke a call to the routine called highest-paid-employee (i.e., associated with the label or identifier «highest-paid-employee») which would "return"—this is the crucial bit—a *name or other identifying label for that person*.[15]

   To state the obvious, the designated person—the living, breathing human being—is of a profoundly inappropriate type for being "returned." Slightly less obviously, the identifier

---

[13]In Python:

```
def factorial(n):
    return 1 if n == 1 else n * factorial(n - 1)
```

[14]It would not, in particular, be assumed to designate a *memory record* containing information about that employee; see §4.V2, below.

[15]What it is to "return" something from a procedure is not a priori clear. Technical answers to this question are typically framed in terms of, rather than independent of, a governing conception of what a program or procedure is. My use of the term here is therefore necessarily informal.

«highest-paid-employee», rather than being a name of the computational routine that returned that person's name, would be taken, on the representational conception, to name a *function* or *mapping* from (actual) institutions to (real, live) employees—a function to which the computational routine associated with the name is mandated to normatively defer, by providing (returning) a formal or symbolic name, witness, representation, analogue or simulacrum.

Already some sources of confusion are emerging. On the one hand, I have said that the routine is *called* «highest-paid-employee», but also suggested that that term—«highest-paid-employee»—in turn names an (abstract) function or mapping from institutions to their highest paid employees. This ambiguity, about whether the term names a (computational) procedure or a (mathematical) function from institutions to employees, betrays the fact that there is some equivocation or conflation in its use—conflation that will need sorting out. However that goes, it is indisputable, as will come to the fore later, that the routine has something to do with the relation between real-world institutions and living people. That is the only conceivable reason why the procedure would be so named—and likely the only conceivable reason why the procedure would have been written in the first place.

It is sometimes suggested that the *intension* or *meaning* of the identifier «highest-paid-employee» should be associated in some way with the computational routine,[16] identified in turn as something like the abstract algorithm or step-by-step process by which it operates. That is: someone, especially one of computational persuasion, might argue that the *intension* of the term «highest-paid-employee», or perhaps the intension of the computational routine of which that identifier is the name,

---

[16]Or perhaps, in order to step away from issues of lexical or grammatical formulation, with the meaning or intension of the routine, if 'routine' is taken to name something like a program fragment as a piece of text, of a sort editable in a word-processing program.

should be taken to be a function or algorithm that takes identifiers of institutions as inputs and produces identifiers of people as outputs. *Extensionally*, in contrast, «highest-paid-employee»[17] might be taken to name the function (perhaps mathematically modeled—a common but distracting issue explored later) that the routine denotes or, as it is said, "computes." I will presently argue, however, that the practice of associating intension or meaning with "means of effective computation," though widespread, is untenably restrictive, and only defensible if one submits to the blanket mechanist vantage point explored in chapter 7.[18] Note additionally that such an intensional/extensional reading would be sustainable only if the procedure is *correct*, which cannot be guaranteed. In fact one of the roles of semantics, as understood in this first conception, is to *facilitate the determination of correctness*, not to presume it.

This attitude of viewing computation representationally, in a way that is analogous to logical inference (and natural language)—a stance which I am here associating with the ingrediential/representational view of programs—will be more familiar to those in the knowledge representation and database theory communities than to the majority of programmers. This is why, in the context of defining a reflective dialect, I would have

---

[17]I am intentionally being ambiguous as to whether I am speaking of the intension or extension of the identifier 'highest-paid-employee,' or of the procedure to which that identifier, as is said in computational discourse, is "bound." (If the identifier is taken to *name* the procedure, then the function from institutions to people would become the designation of the designation of the identifier. It is exactly towards sorting out such conundrums that the considerations in this book are addressed.)

[18]The idea is also philosophically awry; even if 'the morning star' and 'the evening star' have the same extension (the planet second-closest to our sun) but different intensions (or sense; I am not commenting here on the standard philosophical difference between the two notions), the intension would not normally be taken to consist of *words or symbols*; rather, the (abstract) intension or sense would be the "(intensional) meaning of those words or symbols."

done better to use the term 'representational' in place of 'ingrediential' as the predicate on programs conceived in this way, since reflection is likely to bring *any* notion of program within the compass of the overall computation (i.e., is likely to make programs into computational "ingredients," however they are conceived), whether or not they are viewed as interior in the *non*-reflective case. With respect to 2Lisp and 3Lisp, it is certainly the presumed representational character of programs (and thus their deferential semantics), rather than whether they are interior or exterior to the ensuing computation, that was viewed as most important fact relevant to the definition of a reflective dialect.

That I adopted an ingrediential/representational stance towards the constituents of reflection in defining procedural reflection in general, and the 3Lisp dialect in particular, was made explicit in the dissertation, excerpts of which are included in *Legacy*. In particular, based on the Knowledge Representation Hypothesis (KRH) presented in §2.5, I formulated what I called the "Reflection Hypothesis," as follows:

RH   Reflection Hypothesis (1982): In as much as a computational process can be constructed to *reason about an external world* in virtue of comprising an ingredient process (interpreter) formally manipulating representations of that world, so too a computational process can be made to *reason about itself* in virtue of comprising an ingredient process (interpreter[19]) formally manipulating *representations of its own operations and structures*.[20]

---

[19] The use of the term 'interpreter' here is that of computing, not logic or semantics; cf. the discussion in §3.4, below.

[20] There are problems with this formulation, which I would frame differently were I to write it now—for example the pronoun 'its' in the final phrase "representations of its own operations and structures," which is ambiguous as to whether it means the operations and structures *of the ingredient process*, or operations and structures *of the computational*

The passage does not refer to the ingredients of the reflective process as being a *program*—but since the development of 3Lisp arose directly from this mandate, and because the form of reflection that 3Lisp provides is of reflective programs, it is hardly surprising that that is the view of programs on which 3Lisp was founded.

2b Specificational / Prescriptive View

The contrasting view of programs, which in the 1980s I called *specificational*, takes a program not so much to be a *part* of a computational process, at least in the default case (reflection complicates the issue), but rather to be a constructive, effective specification *of what behaviour that process should exhibit*—what behaviour should result from running or "executing" the program. Again, as mentioned above, the 1980 choice of word was not ideal; 'prescriptive' would have been better, since the way in which *specifications* are typically understood in computer science (i.e., the way in which programmers understand the things to which they take the word 'specification' to refer), as well perhaps in mundane natural language, places no requirement on them to exhibit the first critical property for programs discussed above: of indicating not only *what* they specify, but of providing an *effective means* (algorithm, recipe, instructions, etc.) for obtaining that specified entity.[21] What motivated my choice of 'specificational,' though, was its assumption that, rather than being on a par with (other) data structures, programs on a specificational view were viewed as being ontologically and semantically distinct from data structures: prior to, external, and—most importantly—*about* them.[22]

---

processes that can be made to reason about itself.

[21]It is straightforward to specify an entity non-effectively: "the smallest odd number that is the sum of its proper positive divisors," for an arithmetic example; or "the candidate who, twenty years from the end of their term, would in retrospect be deemed to have done the best job."

[22]Were I to retain these two conceptions, I would undertake to rename them here, and start using the new terms here—or perhaps the simpler

(a) **Specification view**

(typical in theoretical computer science)

(b) **Ingredient view**

(typical in AI and cognitive science)

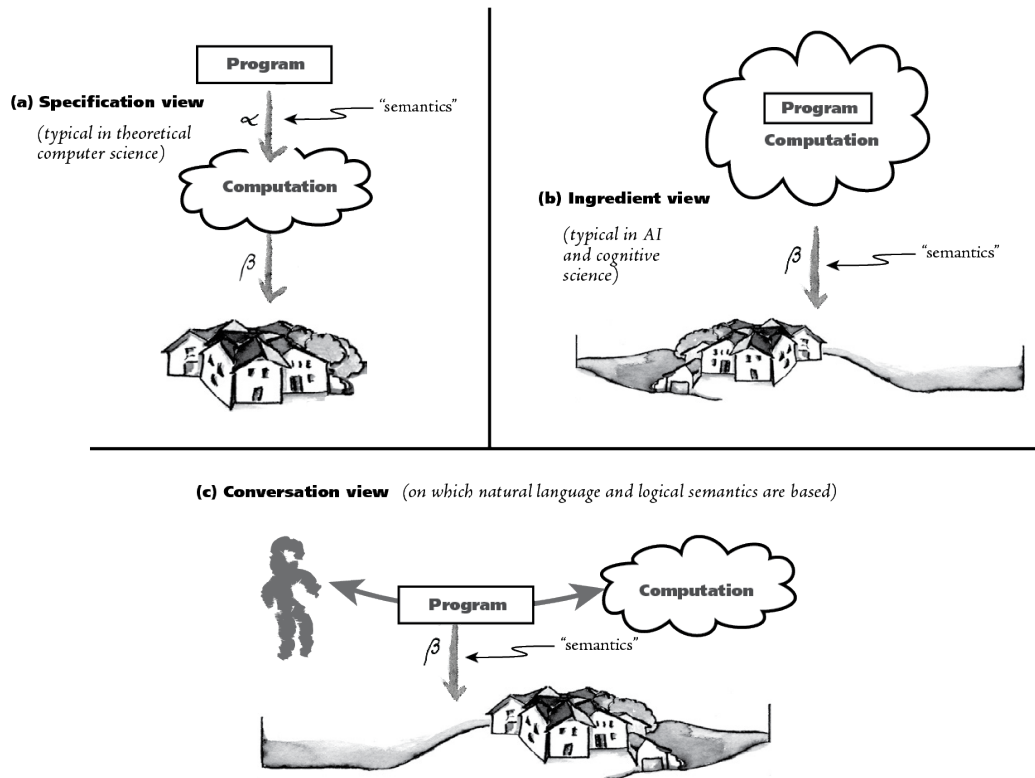(c) **Conversation view**   *(on which natural language and logical semantics are based)*

Figure 1

The difference between the ingrediential (representational) and specificational (prescriptive) views is depicted in several papers in *Legacy*, reproduced here as Figure 1. The diagram highlights two differences, not just one:

1. The specificational/prescriptive view (the standard view in computer science, I now believe) locates the

'representational' and 'prescriptive.' As already indicated, however, I will not do so. The difficulty in these terms is merely a surface indication of the issues at stake.

program *outside* the process of which it is the program, whereas the ingrediential/representational view (the view with closer ties to logic, knowledge representation, and natural language) locates the program *within* that process—within what I call the process' *structural field* (roughly: its memory or realm of data structures);

2. Semantically, the specificational/prescriptive view takes the program to be "meta to"—i.e., to be *about*—the structural field of data structures. To use the terminology of logic, the specificational/prescriptive view suggests that programs can only *mention* data structures, as opposed to the representational/ingrediential view, discussed above, where a program is positioned so as to be able to *use* data structures as ingredients within program expressions (in a "referentially transparent way," as a logician would say). That is: if «employee-134» is a computational constant naming or referring to a living, breathing person, then on a specificational/prescriptive view a program can at most *mention* that computational constant, thereby referring *to it* (to the constant), whereas on the ingrediential/representational view a program could *use* the constant to name or refer *to the person*.

The bottom line is that, on a representational conception of semantics, programs on the specificational/prescriptive view are taken to be at "one level of semantic remove" from programs understood ingredientially/representationally. Thus, as indicated in the examples cited above, on an ingrediential/representational view variables and other referring terms within programs are taken to designate entities in the program's task domain (which would paradigmatically be outside the bounds of the program itself,[23] unless specifically used in a meta-level

---

[23]One might say: "outside the computer," but many programs—email systems, compilers, network routers, etc.—have task domains that are internal to the computational environment as a whole, but "exterior" to

context, such as in reflection). On the specificational/prescriptive view, in contrast, they are *invariably taken to designate process-internal entities*: data structures or memory locations.

Thus suppose, to be pedantically clear, that, when executed, the program fragment «(highest-paid-employee institution-27)» in program fragment E2, above, were to return the identifier «ET0473». On a specificational/prescriptive view, the composite phrase «(highest-paid-employee institution-27)», rather than referring to a living, breathing employee who was *also referred to* by the identifier «ET0473», would be understood as designating or referring *to the identifier ET0473 itself*, or perhaps as identifying or corresponding to a memory location or abstract "record" associated with that identifier (perhaps a memory location or abstract record that the identifier «ET0473» was itself taken to *name*). Similarly—and somewhat perversely, as we will see— on this same specificational/prescriptive view, variable «n», in the factorial example, would be understood as *designating a numeral*, or perhaps a memory location "containing" a numeral, rather than, at least in the first instance, as designating an abstract number.

It is already evident that confusion lurks only minimally below the surface. It almost goes without saying that programmers understand the variable «n» in «factorial»'s definition as referring to a *number*, rather than to a numeral—or possibly (more on this presently) *as well as* to a numeral. Computational readers will also immediately point out that denotational semantical accounts of programming languages *also* take «n» to refer to a number. Crucially, however—and surprisingly—I believe that these last two claims (that programmers' and denotational semanticists' accounts both take «n» to refer to a number) are conceptually independent, as I will argue in chapter 5, based on different (in fact orthogonal) semantical "takes." In spite of its name, and pace virtually ubiquitous misunderstanding, denotational semantics *models* the numeral (or memory

---

the code manipulating them.

location) with the number, whereas in programmers' tacit understanding, the numeral *denotes* or *designates* the number. In this case the "results" are identical (the number two associated with the numeral «2»), but in the general case they diverge. In particular, and crucial to the present discussion: *the deference runs in opposite directions in the two cases*.[24] In programmers' implicit understanding, the numeral signifies, and is deferential to, the number. On the denotational semantics account, the number signifies, and thus defers to, the numeral (or other concrete operations or structures inside the machine).

While rough coincidence of analysis would normally be taken to be a sign of theoretical merit (the number two being associated with the numeral «2», without worrying too much about the directionality of sign vs. signified), I will argue in chapter 4 that in this case it is a distracting conflation—something that has papered over, rather than resolved, cracks in our understanding of computing.

3 Discussion

One natural reaction to the articulation of this distinction between the ingrediential/representational and the specificational/prescriptive views of programs would be to deem it arcane and otiose—to feel that «(highest-paid-employee institution-27)» should be viewed as denoting or referring to, or at least should be able to be taken to denote or refer to, *both* the (external) person *and* the (internal) identifier or memory location, depending on context or perspective.

I have considerable sympathy for this response. One goal of a project I call the "fan calculus" is to support, in a technically rigorous, scalable, and pragmatically usable way, exactly such contextual (and contextually multiplicitous) perspectival semantic interpretation. In fact I believe that understanding how

---

[24]In the "denoting" or "designating" case, the numeral '2' is normatively mandated to defer to the number two; in the modeling case, the number two, qua model, is mandated to be *used in a way that does justice to that which it models*.

programs are constructed and conceived in present-day practice requires a facility with contextual, dynamic semantic interpretation—and thus that the ability to deal with contextual, dynamic interpretation is a requirement on any adequate account of the semantics of computational systems. At present, however, no such contextually-flexible theoretical frameworks are available for such semantical analysis.[25] Sans a crisp account of how such flexibility works, nodding in the direction of flexible, contextually dependent interpretations strategies is not enough to lead to a coherent model of reflection.[26]

Moreover, to return to the case at hand, 2/3Lisp was anything but semantically vague, and—in spite of my current predilections—in no way supported contextually dependent semantics.[27] Rather than licensing multiple interpretations (in logic's sense), it came down strongly on one side—on the ingrediential/representational view—in line with logic, KR, and the Knowledge Representation Hypothesis. The fact that this choice runs contrary to the default understanding of at least most programming language theorists[28] contributed substantially, I believe, to the disappearance of the underlying 2/3Lisp idea.

I was perfectly cognizant, in 1980, that one could take different perspectives on the notion of a program, and that 2/3Lisp specifically adopted the ingrediential/representational view. I also

---

[25]In spite of the contextual complexity with which programming language semantics views are used to coping; this is an entirely different sort of context.

[26] The point is similar to the one made in the Introduction, about how the discursive critique, which would very likely endorse such contextual or multiplicitous forms of interpretation, has not engendered anything like a sufficiently worked-out proposal to serve as the basis for a workable language design.

[27]It did support a semantical analysis that dealt with context—but that is a different matter.

[28]The implicit understanding of working programmers will be taken up in chapter 4.

did my best to describe the perspective I was adopting,[29] argu-ing—and this is what mattered—that the ingrediential view was a necessary basis on which to build a sound conception of reflection.[30] What I failed to anticipate was the fact that the ex-tent to which that perspective departed from what other com-puter scientists took to be the norm was sufficient to "disap-pear" the claim that the ingrediential view was necessary for re-flective clarity.[31] But recognition dawned as soon as I encoun-tered the incomprehensibility with which programming lan-guage colleagues greeted the semantics even of 2Lisp—the "se-mantically rationalized" but non-reflective language on which 3Lisp is based.

An especially telling event occurred in 1984, when, proud of what I took to be its semantical cleanliness, I invited Joseph Goguen and José Meseguer,[32] well-known programming lan-guage semanticists of the day, to sketch a "denotational seman-tics" for 2Lisp. My plan was to use what they proposed for 2Lisp as the basis for the development (perhaps collaboratively with them) of a mathematical analysis of 3Lisp and its version of reflection, and then in turn to use that as a step towards the full-scale theory of reflection that I ultimately aimed to develop (and knew that I lacked).

When Goguen and Meseguer presented their analysis, I was flabbergasted. What they took to be a mathematically clean

---

[29]Cf. for example "Reflection and Semantics in Lisp" (1984), the first published paper on 3Lisp, in chapter 1 (p. 26); in particular, the two par-agraphs starting with "It is natural to ask what a program is…"

[30]I am not sure I believe this today, in part because I do not believe that semantical strictness (believing that semantics is perspective-independ-ent) is tenable. The present point, however, is that it is the ingrediential view on which both 2Lisp and 3Lisp were based.

[31]Departing more radically from theoretical computer science than from workaday programmers, however—or anyway so I believe. See chapter 4.

[32]Then at SRI International; now at the University of Illinois at Urbana–Champaign.

semantical analysis completely obliterated what I took to be es-
sential to 2Lisp's semantical clarity—conflating distinctions I
had taken pains to maintain, including those between and
among numerals and numbers and (what 2/3Lisp called) *han-
dles* and the data structures for which they served as names, se-
quences and the normal-form sequence-designating structures
I called *rails*, etc.[33] Entities I took to be concrete were treated as
abstract; the grounds on which I had rested my critique of the
Lisp conception of *evaluation*[34] vanished; and in general their
"theoretically clean" version of 2Lisp underwent a transfor-
mation that not only rendered it wholly unfamiliar to me, but
also "disappeared," at least in my eyes, its major contribution—
the very semantical cleanliness that I took to be its distinguish-
ing feature. Even more tellingly, the distinctions I took to be
criterial for defining and implementing reflection—distinc-
tions between signs and what they signify, and between things
abstract and things concrete—had also been erased.

I mean nothing indicting with this tale. Goguen and Me-
seguer were eminent theorists, and generous to a fault; I have
nothing but the highest regard for both. It was a perfect case of
what anthropologists would call *cultural clash*: two parties ap-
prehending a subject matter from different, even irreconcilable,
perspectives. Yet in spite of good will on both sides, the pro-
posed collaboration stalled. I never did develop a mathematical
account of reflection—and for reasons that will presently
emerge, no one else has, either.

The difficulty, to put it bluntly, is that whereas a divergence
in perspective on *programs* might, at least superficially, be con-
sidered a matter of taste, the same does not hold for reflection.
Striking the right semantical stance towards reflection—strik-
ing, in particular, a tenable version of what I am calling a defer-
ential semantical stance—is a prerequisite for understanding

---

[33]As described in §4.V4, rails are defined in 2/3Lisp to be the ordered
data structures that represent abstract sequences.
[34]Including my indictment of its blithe acceptance of «(+ 1 '2)»

what reflection even *is*.

As I grew clearer on the underlying issues, I began to be able—especially in conversation—to explain the perspective from which 2/3Lisp was designed (even if I failed to convince anyone of that perspective's merit). What was especially sobering was that success in describing the 2Lisp architecture required *not* using the ingrediential vocabulary employed in all of the *Legacy* papers. That is, it depended on my *not* saying that the design (and technical vocabulary) of 2Lisp was based on a view of programs as causally effective process-internal ingredients—even if I went to some pains to explain the ingrediential/representational vs. specificational/prescriptive distinction. Rather, I had to describe the 2Lisp architecture *from the specificational/prescriptive viewpoint*, which at the time felt alien to me: taking programs to be external, if nevertheless effective, process specifications. The difficulty was that, to the extent that that strategy worked, it thereby rendered the possibility of explaining 3Lisp, and the ensuing model of reflection, that much more difficult, if not outright impossible. The reason is explained in more detail in chapter 4, but among other things it comes down to this: adopting a specificational/prescriptive view "uses up" all standard semantical vocabulary in describing the program-process relation, leaving one without any words with which to describe what I felt to be the constitutive notion of "aboutness" in terms of which reflection must be defined.

Another anecdote is illustrative, this time from a mid 1980s conversation with Gordon Plotkin.[35] After failing, using my own terminology, to communicate anything about what mattered to me about 2/3Lisp, I decided to adopt his. That is, I attempted to "inhabit" the specificational/prescriptive view, and said that what I was interested in, in designing even 2Lisp, let alone 3Lisp, was *the semantics of the semantics of programs*."

---

[35]Now at the University of Edinburgh; this took place at Stanford's Center for the Study of Language and Information (CSLI).

To an extent the ploy must have worked, as I recall Plotkin nodding and smiling. But the differences were profound, and nothing further came of that conversation, either. Although I made some subsequent attempts to explain the differences in viewpoints, it seems safe to say that 2Lisp's and 3Lisp's deferential approach to semantics, on which the latter dialect's notion of reflection was based—including, for example, the idea of theorizing distinct procedural and declarative aspects of program meaning, assembled into a single encompassing significance function—was sufficiently divergent from consensual practice as to have achieved incomprehensibility.

4 Merits & Demerits

Before moving on to the second and third diagnoses, it is instructive to identify some of the merits, demerits, and questions facing this first one, based on two differing conceptions of program.

I will frame these positives and negatives in terms of the prescriptive/specificational view, since that is the "received" view in computer science. On the positive side, somewhat tautologically, the most obvious advantage of this view is the extent to which it meshes with contemporary theoretical discourse—how programmers talk, how papers in computer science are written, and so on. Whether that implies that it does justice to how programmers actually understand programs is a separate question, to be taken up in chapter 5. Nevertheless, at least the following points can be counted as in its favour:

PRO-1   The specificational/prescriptive view makes sense of why computer science calls language processors *interpreters*.[36] If one takes the semantic value of a program

---

[36]If the program is compiled, then it is not interpreted directly; but the resulting compiled code has to be executed. We do not typically use the term 'interpretation' for the CPU's processing of machine code, but if the program is compiled into another language, then that language may be said to be interpreted. Many Java programs, for example, are compiled

or program expression to be *the behaviour or results that eventuate from its execution* (theorized as "the behaviour or results *that it specifies*"), then, sure enough, processing it will produce what (from that point of view) it "designates," making sense of the computational use of this originally logico-semantical term.

PRO-2   To an extent, the specificational/prescriptive view explains why many theoretical computer scientists are interested in constructive mathematics. It is essential, in order to understand reflection, to realize that *nothing about computation per se mandates taking a constructivist approach to mathematics*. If, however, one (i) adopts the specificational/prescriptive view, (ii) understands program execution or processing to model or be paradigmatic of human mathematical activity,[37] (iii) takes computation *itself* to be a mathematical phenomenon,[38] and perhaps (iv) pays implicit obeisance to something like blanket mechanism,[39] then it becomes natural to focus on the restriction of mathematics to its constructive subset.

     A similar argument can be made about intuitionistic type theory, though in contrast to many

---

into byte codes, which are processed by what is commonly called the "byte code interpreter."

[37]One might argue that associating human mathematical cognition with the processing ("executing") of programs follows as a general conclusion of the computational theory of mind (CTOM), but I believe that is mistaken. Moreover, I suspect that those who subscribe to the idea believe it for much more local reasons.

[38]I.e., views computation, rather than being something concrete, such as a system of symbols that denote or represent mathematical entities, or, like any scientific phenomenon, as being a concrete system that can usefully be *modeled* mathematically, but rather as *itself a mathematical phenomenon*, on a par with numbers, sets, or morphisms—a stance I discuss in §7.1, below.

[39]See chapter 7.

constructivists and especially formalists, many intuitionists—including Per Martin Löf, whose type theory has received so much attention in computational circles—are not mechanists of any stripe. Nevertheless, if one wants a type theory in order to describe, from a program viewed as an effective (i.e., prescriptive) specification, the types of structures it is able (mechanistically) to "generate," there is clear sense in selecting an intuitionistic type framework for the purpose.

PRO-3   The prescriptive/specificational view also makes sense of an otherwise inexplicable passage in Newell's "Physical Symbol Systems" paper, based on his Turing award lecture with Herbert Simon, in which the authors attempt to characterize computing. To a classical logician or semantical or semiotic theorist, the passage verges on the daft:[40]

> "The most fundamental concept for a symbol system is that which gives symbols their symbolic character, i.e., which lets them stand for some entity. We call this concept *designation*, though we might have used any of several other terms, e.g., *reference*, *denotation*, *naming*, *standing for*, *aboutness*, or even *symbolization* or *meaning*. The variations in these terms, in either their common or philosophic usage, is not critical for us. <u>Our concept is wholly defined within the structure of a symbol system</u>. This one notion (in the context of the rest of a symbol system)

---

[40]Newell (1980), §4.1, p. 156; emphasis in the original, underlines added.

This paper is widely regarded as definitive of the "symbol manipulation" conception of computation—a construal not usually distinguished from what I call "formal symbol manipulation" (FSM)—though because of the issues of symbol and semantics being here discussed, from a conceptual point of view I take them as almost orthogonal.

must ultimately do service for the full range of symbolic functioning.

> Let us have a definition:
>
> *Designation:* An entity X designates an entity Y relative to a process P, if, when P takes X as input, <u>its behavior depends on Y</u>."

Four facts about this passage are striking.

a. Newell's apparent recognition of the centrality of issues of symbols and intentionality is both notable and salutary.[41]

b. Nevertheless, this account could never stand as a *general* account of symbolism (meaning, representation, symbolization, intentionality, etc.). In the general case, reference, denotation, meaning, etc., of symbols are absolutely *not* "within the symbol system." On the contrary, as emphasized in the <u>Introduction</u>, symbols, at least paradigmatically, are exactly a way of relating a system to that which is external—to that which is distal, beyond effective reach. I take such distal reference, and the normative deference accompanying it, to be constitutive not only of logic, but of symbolism in general—indeed, to be among the most fundamental facts about all of intentionality.

> To put it informally, one might almost say that the ability to denote or refer to that which is *not* "wholly…within the structure of a symbol system" is fundamentally what symbols are *for*.

---

[41]I say 'apparent' because, as argued in <u>chapter §4.VI</u>, it is ultimately unclear that Newell's concerns and intuitions in this passage ultimately have anything to do with intentionality, as opposed to being merely a use of intentional vocabulary (symbol, designation, etc.) to characterize purely mechanical entities and relations.

c. As is evident from both underlined fragments, Newell and Simon are operating within what in chapter 7 I call "blanket mechanism." The operative sense of 'depend' in their characterization is clearly *causal*, rather than being of the normative, non-local, semantical nature on which logic is based, and that I believe is fundamental to all intentionality—a conception spelled out in chapter 7.

d. Most immediately relevant, this causally-dependent conception of designation (reference, meaning, etc.) is not just applicable to, but must in fact be directly motivated by, the sort of "effective specification" view of programs under scrutiny here.[42]

PRO-4   The prescriptive/specificational view explains why programming language constants, variables, identifiers, etc. are so often taken to be names of *memory locations*—in spite of having natural language names most commonly intelligible in the realm of the program's task domain («institution-27», «next-floor», «current-

---

[42]Even on a prescriptive/specificational view, the final (underlined) statement is not clear: if a program denotes or designates a process-internal ingredient, such as a data structure, then the behaviour of the processor ("interpreter") will presumably *affect* the data structure (e.g., if a program prescribes incrementing a register, then the register will be affected). Newell and Simon's claim would be that the term in the program denotes that register just in case the behaviour (of the program? of the program's processor? of the process that results from the processor processing the program? It is not clear.) depends on the register. If the behaviour is taken to be "that the register is incremented," then that presumably does depend on the register, in at least some sense. But the behaviour, if individuated finely enough, also depends on what the contents of the register were—and one would not normally say that the term in the program (e.g., a memory address) denotes *the state of the memory register*. In spite of its fame, the paper is no beacon of clarity.

user», etc.). Some accounts take these memory locations to be concrete; others, as abstract mathematical entities, or anyway as mathematically modeled. Modulo these variations, however, the approach of taking names in programs to identify storage locations is universal.

PRO-5   The prescriptive/specificational view makes sense of what computer science takes a semantical analysis of a programming language to be—a conception that would otherwise be mysterious (*is* mysterious, in fact, if one views computation to be formal symbol manipulation in the style so widely assumed in cognitive science and philosophy of mind). Three examples:

a.   In contradistinction to what would be true of a logical or representational language, the semantical analysis of programming languages is never parameterized on an externally-supplied interpretation for the atoms, constants, primitive predicates, etc., or for any of the data structures;[43]

b.   Since the full semantics for the language is specified by the semantical analysis, with no room for any variation for specific programs and task domains, it can be assumed that the normative constraints on implementations are that they be *correct*, not merely *sound*;[44] and

---

[43]A semantical account of a logical or FSM system, for example, would typically assume that constants (e.g., 'SOCRATES', 'P' or 'Q'), predicates (e.g., 'MORTAL' or 'SUCCESSOR'), relation symbols (e.g., 'LARGER-THAN' or 'PRIME') would be given interpretations by a "user-supplied" interpretation function, which would then be used by the semantical analyses as a basis for building up the interpretation of complexes built up out of them, such as 'MORTAL(SOCRATES)' or '$\forall x [\text{PRIME}(x) \supset \neg\text{PRIME}(\text{SUCCESSOR}(x))$).' See chapter 6.

[44]Similarly, because the definition of the language includes the interpretations of all of its terms, there is in a sense no salient difference between

    c.   It is standard, in computer science, for denotational and operational semantics to be *proved equivalent*, without that constituting a proof that the system is "complete" (which is what it would imply if, as some philosophers incorrectly hold, operational semantics were an account of the "proof-theoretic" behaviour of the system, and denotational semantics an account of what, intuitively, the system represents or is about).

These facts, and others that could be cited, constitute strong evidence that the prescriptive/specificational view is indeed the "official" view of theoretical computer science.

Two things should be noted, though, before taking that evidence as justificatory—i.e., before taking the points enumerated above as suggesting that the specificational view is correct or even preferable. First, it has not yet been argued that this is how programmers understand programs, and in chapter 5 I will argue that, on the contrary, the specificational/prescriptive view parts company with programmer's intuitive understanding in several critical ways—casting doubts on its overall adequacy. Second, even if the foregoing points are thought to support the specificational/prescriptive view (in a moment I will argue that they do no such thing), the specificational/prescriptive view is accompanied by an equal or even greater number of demerits. If they do not constitute an argument against the prescriptive/specificational view, these infelicities at least suggest that it needs to be questioned.

In particular, here are eight reasons militating against the adequacy of the specificational/prescriptive view (to which others could be added):

---

material and logical implication (though because of *temporal* variation in the values of variables, the contents of memory registers, etc., and because those states are, among other things, relative to unpredictable input, implications may of course sometimes hold and sometimes not).

CON-1 On a specificational/prescriptive view, programs cannot add numbers, or do arithmetic, or in fact engage in any mathematical operations at all. At least this is so on anything like a *symbol-manipulation* construal of computing—or indeed any construal of computation on which computing is *concrete*. For if, according to this view, the act of a program interpreter is to *produce what is designated*, then the designation (i.e., interpretation[45]) of an expression such as «2+3»—or its Lisp equivalent «(+ 2 3)»—must be a *symbol*, not a number. Or put it this way: if, as commonsense would overwhelmingly suggest, the interpretation of «2+3» was taken to be the *number five*, then the interpreter would be mandated, on the specificational/prescriptive view, to *produce a number as its result* (rather than produce a representation or signifier of that number)—an operation that common sense, as well as most philosophies of mathematics, would consider to be metaphysically impossible.[46]

CON-2 Semantic analyses of programs, according to the specificational/prescriptive view:

a. Cannot make sense of why programmers use English or other natural language terms for their constants, variables, data structures, etc.—e.g., the «institution-27» and «current-floor» used in the examples above; and

b. Rob the programmer and theorist of theoretical

---

[45]As is customary, I take *designation* to be the species of interpretation applicable to *terms*.

[46]Cf. Dretske (1985).

Even if one were a hyperintuitionist, and felt that numbers were inscriptions, a program could at best return an *instance* (token) of a number, not a number itself. If a computer were given the expression «+ 2 3», and then damaged before the result could be read, it would be madness to say "the number five was lost."

language in terms of which to talk about the evident relation between those data structures and the corresponding task domain entities with which they are associated, because such semantical terms as *reference*, *designation*, etc. (i.e., all the terms in Newell and Simon's list) have all been "used up" to name the causal relation mediated by the language processor.

If, for example, as the prescriptive/specificational view claims, the name «institution-27» designates or refers to a memory location, then why does the programmer not give it the label «memory-location-43916»? How do we explain the fact that, if such a renaming were effected, the resulting program would be incomprehensible to programmers and maintainers alike? And if there is a three-way relation among (i) the symbol institution-27, (ii) the memory location memory-location-43916, and (iii) the institution designated as 27th in some enumeration, what should that three-way relationship be called?

It is telling, in this regard, that in 1982 Alan Newell, widely recognized as a champion of the symbol manipulation construal of computing, was forced to introduce the otherwise inexplicable notion of "The Knowledge Level," in order to accommodate the fact that computational ingredients are, in fact, employed in such a way that they bear a semiotic or semantical relation to the wider world.[47] The very existence of the "Knowledge Level" idea betrays recognition on Newell's part that the definition of symbols and designation in his and Simon's characterization of physical symbol systems was inadequate to comprehend both programmers' understanding and the genuine

---

[47]Newell (1982).

computation-world relations critical to the very notion of computing.[48]

CON-3 It is for the sorts of reason adduced above that semantical analyses of programming languages do not give rise to appropriate semantical analyses of the programs written in them. That is, these facts explain something noted in §1.6: that computer science has so construed symbols, semantics, etc., as to lead to the perverse situation that the semantical analysis of a *language* (a programming language, in this case) provides no wherewithal with which to understand the semantics of *texts written in that language*—i.e., no way of understanding the semantics of *programs*.

Whether or not the formal symbol manipulation (FSM) view of computation is correct, it is surely *correct enough* that any analysis of real-world computational use must provide means to understand the relation between computational ingredients and task domains entities. Most importantly: as Newell himself realized, it is the relations to the task domain which establish the deferential conditions to which the resulting program is held normatively accountable.

CON-4 The conception of semantics employed by the prescriptive/specificational view in order to account for the effective relation between a program and the behaviour that results from processing it is entirely restricted to the realm of the effectively mechanical. All issues of deference, normativity, etc., are excluded from the discussion—making it emblematic of the sorts of "blanket mechanism" restrictions discussed in chapter 7.

CON-5 Redefining (torquing, in my view) the notions of

---

[48]Or to serve as a way of understanding the warrant or correctness of its reasoning processes.

symbol and semantics to suit the requirements of the prescriptive/specificational view of programs leaves completely unanswered the deeper question of *what computation is*. It is "symbol manipulation," on the redefined notion of 'symbol,' only in the sense that passive or static symbolic programs are processed in order to produce the behaviour that they are (thereby?) taken to specify or designate. But nothing is said either about *the nature of that which is specified*, or about *what it is to specify*, beyond that what a symbol specifies or designates is *what it causally engenders when processed*.

In particular, Newell's characterisation of a 'symbol' as anything that has (partial) causal influence on that which it is taken to designate is such that one can no longer say that computational processes are *symbol manipulation* in any sense amenable to common sense, familiar to philosophy, or consonant with hundreds of years of logical understanding. Adjusting dials on an automatic milling machine serves to influence what metal parts it produces. Does grinding metal thereby count as computation, or adjusting the dials as programming? Or to move even further away from anything we would pretheoretically call specification, suppose that eavestroughs causally lead rain to flow through downspouts instead of over roof edges. Are eavestroughs thereby *symbols*? On Newell's characterization they would have to be counted as such.

CON-6 Though how much this would count against the specificational/prescriptive view would depend on one's theoretical commitments, it is at least noteworthy that programming language *interpreters*, on the conception identified in CON-1, above, cannot be *formal*, on the standard definition of formality, which requires that a symbol be treated *independent of its semantics*, since, on the view being proposed, the interpreter *causally produces that which it semantically designates*.

Even more telling against the prescriptive/specificational view than any of the foregoing six points, however, are two more general ones, which not only do some additional work in unsettling it, but which also up the ante on what will be required in order to provide a theoretically adequate account of computational reflection.

CON-7 Not a single one of the five positive comments about the prescriptive/specificational view listed above (PRO-1 through PRO-5) have *anything to do with the fundamental nature of computing*. Each speaks only to this or that aspect of the way in which computing is *currently theorized*. Moreover, as demonstrated by 2/3Lisp, none of them is necessary to the development of a programming language, or to the development or understanding of programs written in it.

That is, not a single one of conditions PRO-1 through PRO-5 is met by (i) any of the 2/3Lisp dialects themselves, (ii) the theoretical framework in terms of which those dialects were designed and described, (iii) the semantical account in terms of which 2Lisp was shown to be semantically rationalized and normatively sound, and (iv) 3Lisp's reflective model. In particular:

a. There is no need to call the process that generates activity according to the dictates of a program an *interpreter*. In my view, though the usage is common, the terminology only serves to muddy the semantical waters. Far better, I believe, to call such things *language processors*. That terminology leaves open the question of the programs' semantics, and of the semantical character of the execution process.

b. As stated above, nothing about computing compels one to endorse (or, for that matter, even to take a stand on) constructive mathematics or intuitionism. Symbol manipulation is compatible

with mathematical reasoning of all sorts. The no-
tion of computing is catholic as regards philoso-
phies of mathematical entities thereby reasoned
about.

c.   Newell and Simon's characterization of designa-
tion (and other semantical relations) is not just
entirely non-standard, but untenably mechani-
cal. Not only does it obscure, rather than illumi-
nate, the genuine structure of semiotic and se-
mantical phenomena; it also does nothing to ex-
plain the nature of programs (other than recog-
nizing, *along with things that are not programs*,
that they play a causal role in engendering behav-
iour).[49]

The characterization of language processes as
"interpreters" also hinders, rather than helps, in-
tellectual communication between computer sci-
ence and surrounding disciplines. In fact the en-
tire "physical symbol systems" (PSS) construal of
computation, I believe, has done more to hamper
the wider intellectual appreciation of computing
than any other single idea.

d.   To take constants, variables, data structures, etc.,

---

[49]Suppose we define relation $\varphi(X,Y)$ using Newell & Simon's characteri-
sation: that $\varphi(X,Y)$ with respect to process P if (and, we can assume, only
if), when P takes X as input, its behaviour depends on Y. What would
one imagine φ to be? Basically, it is simply any relation that X bears to
anything that affects X's treatment by P. For example, if P is a process of
lifting X (i.e., suppose P is something like a fork-lift, or fork-lift behav-
iour). Then Y could be X's mass, or the glue that holds X down, or the
distance of X from P. But do we want to say that X *designates* its mass, or
the glue that holds it to the shelf, or its distance from a fork-lift? Obvi-
ously not.

Unfortunately, there is no substance to the Newell and Simon ac-
count that prevents these and untold other causal dependencies from
being called "designation."

to refer to *memory locations* or *abstract memory records* confuses semantics with implementation, in my view, violating what is almost universally taken to be a goal of elegant computational theory: of abstracting what matters about a programming language away from the distracting influence of contingent matters of implementation (especially implementation on a Von Neumann architecture).

Note, in this regard, that the semantics of 2/3Lisp makes no reference to implementation of any sort—a fact I take to be to its credit. And contra what is often said, characterizing reflection as a technique for *making the implementation explicit* is not, I believe, an apt or penetrating conception of reflection.

e. It takes only a few moments to see that, from a general semantical or semiotic viewpoint, all five facts cited about the current practice of programming language semantics—PRO-1 through PRO-5—are semantical oddities. For example, it befuddles rather than benefits analysis to lose the notions of *soundness* and *completeness*—to remove, from the theoretical machinery, any ability to make reference to the genuine semantical interpretation of the constituent symbols.

The inapplicability of the semantical model implied by the specificational/prescriptive conception of program, mentioned above,[50] is a major liability for computational theory.

CON-8 Finally, above and beyond all of the foregoing points, perhaps the most important limitation of the specificational/prescriptive view, in the present context, is

---

[50]See §5.2.

that it fails to be an adequate semantical view in terms of which even to *define* reflection.

Not a single one of these demerits applies to the ingrediential/representational view. On the contrary, the ingrediential/representational view:

1. Is compatible with both constructive and non-constructive approaches to mathematics, or indeed mathematical philosophies of any other variety;

2. Makes eminent sense of why programmers use natural language terms for variables—and even suggests that programs should be held normatively accountable to the references and denotations that programmers have in mind for those terms;

3. Engenders an approach to semantics that is perfectly appropriate for treating the semantics of individual programs, not just languages as a whole;[51]

4. Is by no means restricted to a "blanket mechanist" approach to computation (chapter 7);

5. Not only makes room for, but in fact *provides an answer to*, the question of what computation is: computation is symbol manipulation, with programs being a paradigmatic example of such symbolic structures; and

6. *Does* provide a basis on which to define a conceptually justified notion of reflection, as evidenced by 3Lisp.

Let it immediately be said, however, that none of this is to suggest that the ingrediential/representational view is without its own problems. On the contrary, the difficulties it faces are considerable—or at least the difficulties faced by any version of it

---

[51]The mandated theoretical changes are not straightforward, however, and would require that standards of "correctness" are replaced with those of "soundness," since interpretation functions would, as in logic, have to be parameterized on a program-specific interpretation of atoms, identifiers, and data structures in general.

on which we yet have a decent theoretical grasp. They have to do with fundamental issues of ontology, with still-unanswered questions about the nature of semantics, with limitations as regards how we understand the semantical or intentional nature of *activity* or *process*, on issues of deixis, indexicality, and perspectival conceptions of semantic value, and myriad other issues. In a way, in fact, the employment of a ingrediential/representational view in 2/3Lisp, and the practical as well as theoretical failure of those dialects, illustrates not only the advantages, but also the disadvantages, of any ingrediential/representational view that we are currently able to formulate.

Even more strongly: I would be the first to admit that the particular form of representational framework used to design and analyse 2/3Lisp is ultimately no more tenable than the specificational/prescriptive view. Crucially, though, that fact does not supply any warrant for decrying representation in a broader sense. There is more to representation than the version used in 2/3Lisp.[52]

In chapter 8 I will sketch some ways in which the merits of the specificational/prescriptive view might be incorporated into a revision or opening-up of the ingrediential/representational approach, so as to combine their respective merits. But that proposal will be merely a suggestion, which would take work to develop. We are still a long ways away from having forged a theoretical understanding adequate even to present-day programming practice, let alone to the future of computing.

---

[52]See "Rehabilitating Representation" (Smith, 1998).

4 Semantics
Diagnosis · Second Pass

We need to dig deeper.

The first diagnosis distinguished two conceptions of the notion of a program. Analysis suggested that that issue is merely the tip of an iceberg. En route, we uncovered a range of additional issues in need of exploration: about the relation between the semantics of individual programs and the semantics of the programming languages they are written in, about the nature of computation itself (e.g., whether computers can add), about the relation of computation to mathematics, and so on. This suggests a second diagnosis, which instead of focusing in on programs, addresses the problematic from a more general semantical perspective, with the aim of clarifying the structural configuration in which symbolic systems are deemed to play a role.

The problematic has two major dimensions. First, a spate of entangled issues are involved, including at least *program*, *process*, *semantics*, *meaning*, *interpretation*, *mechanism*, and *computing* itself. Second, this range of concepts needs to be assessed from several viewpoints, of which four are already on the table:
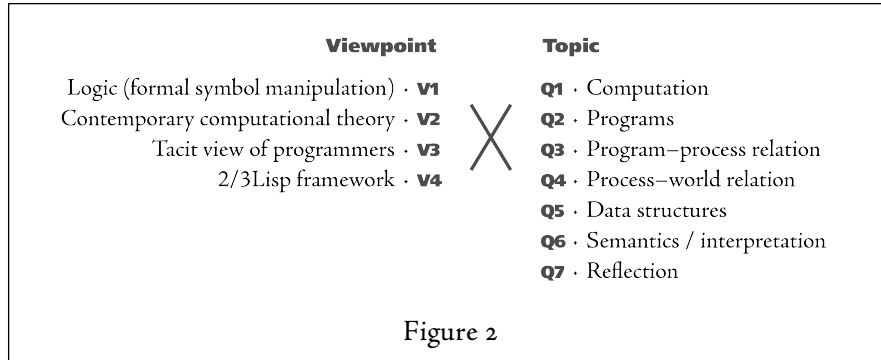
1. That on which philosophical logic and KR are based, evolved from considerable Cartesian and post-Cartesian philosophy, having not only to do with rationality, reasoning, proof, etc., but also, among other things, with a proposed solution to the problem of "mental causation";

2. That embraced by current computational theory, as embodied not only in computability and complexity theory, but also in existing approaches to programming language semantics (including both operational and denotational varieties);

3. That on which programmers tacitly rely, in performing their work—assuming that, even if not explicitly articulated, their collective work is based on at least a roughly shared understanding; and

4. That in terms of which 2/3Lisp was designed and theoretically framed.

Each viewpoint makes its own assumptions, embodies particular insights, has significant merits in some respects, and fails in others. The task of this chapter is roughly archaeological: to give voice, from a workably comprehensive vantage point, to the assumptions, commitments, and perspectives of each of the four viewpoints listed above, explaining their stance on the enumerated suite of issues (*program*, *process*, *semantics*, *interpretation*, etc.), and in so far as is possible at this stage, exposing their relative merits and demerits. The hope is to use this catholic understanding as a basis on which to develop a map of the whole territory. With luck, we will be able to discern, at least in outline, some of what will be required in a "successor" approach— a candidate alternative that integrates the merits of all those assessed. I will not present such a successor view here, but some initial framing comments are given in chapter 8.

The investigation is bedeviled by the issue of conflicting vocabularies mentioned in the Introduction. The various disciplines being excavated—computer science, logic, knowledge representation, philosophy of science, etc.—use many of the same words in different ways. Because I expect readers of these pages to come from all these communities, the presentation cannot assume any single common

Viewpoint		Topic
Logic (formal symbol manipulation) · **V1**		**Q1** · Computation
Contemporary computational theory · **V2**		**Q2** · Programs
Tacit view of programmers · **V3**		**Q3** · Program–process relation
2/3Lisp framework · **V4**		**Q4** · Process–world relation
		Q5 · Data structures
		Q6 · Semantics / interpretation
		Q7 · Reflection

Figure 2

terminology (as would be the case for example, if this were merely an attempt to explicate, from the viewpoint of analytic philosophy, computer science's use of the technical vocabulary it inherited from logic, or if it were addressed to programmers with primarily pragmatic concerns). One approach would be to argue for the superiority of one set of terminological uses, and then to use that one, but as it happens I endorse none. Another would be to introduce new vocabulary and/or neologisms, but the exploration hardly warrants that (this is a *diagnostic* endeavour, after all, not yet a synthetic one).

To mitigate these odds, I will adopt a rather formal strategy, depicted in figure 2. First, the viewpoints I want to interrogate are four, suggested above but more carefully identified as follows:

V1 The classical "formal symbol manipulation" (FSM) view of computation, based on a conception of symbols and semantics inherited from classical logic and philosophy of language. Not only is this view widely associated with classical artificial intelligence (GOFAI), but many philosophers of mind and cognitive scientists[1] assume that it is constitutive of what computation is. I will not address the broad claim

―――――――――

[1]E.g., Van Gelder (1995).

here (about its adequacy as an overall theory of computation, let alone as a theory of mind); I will restrict myself to assessing it as one of several competing perspectives on the nature of computing.

V2 What in chapter 3 I called the "official computer science" view, associated with Turing machines, mathematical theories of computability, and most approaches to programming language semantics, including denotational, operational, algebraic, and axiomatic. As I have noted elsewhere, outsiders to computing are likely to think that V2 is the same as V1; one goal of this chapter is to demonstrate how far that is from being the case.

V3 Something that I will presumptuously call "the tacit view of programmers"—not a perspective that has received theoretical articulation, but my own intuitive sense of how programmers in fact understand the programs and systems that they build. I have no proof that this view is representative or widespread. I include it not only because of how far it diverges from V2, with consequences for reflection and for our understanding of both computation and semantics more generally, but also because I believe its examination will foreground important requirements that any successor account must meet.

V4 The framework in terms of which 2/3Lisp was defined, which started out based on V1, but was modified and extended in a number of ways to deal with some of the issues that V2 treats directly, especially having to do with the consequences of dynamic processing.

The strategy will be to ask the same baker's half-dozen questions of each view—regarding its assumptions about, and theoretical orientation towards:

Q1 The nature of *computation*;

Q2 The nature of *programs*;

Q3 The character of the *program–process* relation;

Q4 The character of the *process–world* relation, where by 'world' is meant the program's (and process') task domain;

Q5 The nature of *data structures*;

Q6 The nature of *semantics* and *interpretation*; and

Q7 The nature of (and prospects for) *reflection*.

By the program–process relation (Q3), I mean the relation between: (i) the static or at least relatively passive program—the sort of entity that might be printed out, edited before the program is run, etc.; and (ii) the behaviour or activity that results from "running" or executing that program, whether that is considered merely in terms of an initial input and final output, more richly in terms of ongoing activity, or whatever.[2] What is critical, for purposes of this question, is that the 'process' or 'behaviour' *not* be understood "under interpretation" (interpretation-L)—as, for example, in "adding the numbers two and three" (at least on the formal symbol manipulation view of computing), or "ordering students by age." Though otherwise perfectly unexceptional, these descriptions involve at least one level of (semantic-L) interpretation of the mechanical or behavioural results, and thus combine issues of both Q3 and Q6. An appropriate answer to Q3 alone, in the first of these two cases, would be something along the lines of "two numerals are taken as input, and a numeral produced as output, normatively mandated to denote (designate, refer to, etc.) the sum of the numbers denoted by the inputs."

Questions Q1–Q7 are interrelated, but in different ways

---

[2]If it is helpful, one might substitute the word 'behaviour,' taking Q3 to be about the *program–behaviour* relation, though I believe 'process' is more general.

for different views. Thus the program-process relation (Q3) is pretty much what the official CS view (V2) calls "semantics" (Q6), whereas in logic and FSM (V1) semantics lies closer to the relation between a process and the world (Q4). Because of these different associations, in considering each view (V1–V4) I will not always address the questions (Q1–Q7) in the same order. Moreover, in some cases it may be that Q6, concerning semantics, has effectively been answered in the course of addressing Q1–Q5. The point is merely that it will be taken as a criterion on our assessment of each of the four views that by the time we are done we have developed a clear understanding of how that view treats each of the seven questions.

The aim will not only be to identify "what reflection would look like" from each point of view, but also to identify the minimal conditions on a tenable answer, so that, as we examine each view of computing, programs, etc., we can consider whether it provides sufficient resources for designing an architecture that can lay legitimate claim to being reflective at all.

As already intimated, I do not think that V2—the official CS view—is strong enough to meet that challenge. I would also be the first to admit that the 2/3Lisp approach (V4), while more promising than either V1 or V2, remains inchoate and limited.

One final preparatory comment. Not surprisingly, theoretical discussions about computing, especially discipline-internal ones, are often conducted in mathematical terms. Throughout both science and philosophy, it is controversial as to whether the mathematical entities adverted to in scientific explanation, such as numbers, functions, sets, vectors, and so on, should be understood as ontologically intrinsic features of the subject matter, or whether they are better considered as *epistemic aids*—theoretical equipment used to model or understand the subject matter or phenomena in question.

The distinction is of philosophical importance in its own right, relevant in the foundations of science and fundamental metaphysics. In practice, though, it is liable to be of little concern to working scientists, who are unlikely to confuse the mass of a proton, or the velocity required to escape from solar orbit, with the numbers used to classify them. In most sciences, that is, the differences between measures and what is measured are usually made manifest in such techniques as the ubiquitous use of units and coordinate systems (1.67×10^{-27} kg, 42.1 km/sec, etc.).

The situation in computer science is considerably more vexed. It is absolutely unclear—and in fact I believe there is ample evidence that there are differing views among both working programmers and theoretical computer scientists—as to whether computing is *itself* a mathematical domain, or whether, as in other parts of science, mathematics is used, for theoretical purposes, to *model* or *classify* computational phenomena. We talk about computing *numbers* (e.g., "compute the positive square root of 961"), but the question of whether that phrasing is short-hand for "produce, via some effective means, a *numeral that denotes* the positive square root of 961" is usually left implicit—and I suspect any particular answer would be disputed, if posed. Certainly on a conception of computation as formal symbol manipulation, as I have indicated several times, numbers *per se* cannot be the output or result of a computation—for the simple reason that, at least on anything like a traditional conception, numbers are abstract, whereas symbols (or anyway their instances) are concrete, at least symbols of the sort that formal symbol manipulation machines could produce.[3] So if one genuinely believes that

---

[3]Symbol *tokens*, at least, are concrete; symbol *types*, understood in such a way as to claim that the expression '$a=(a+b)$' contains two different symbols, are of course abstract. No types of any sort can be returned by a computational procedure, and so it would not help, if one wanted to defend the idea that computation involves numbers, to identify the number

computations can issue in numbers, then one must reject the idea that formal symbol manipulation is what computation is. (Note, too, if one's predilections run concrete, that "compute the positive square root of 961" cannot simply be an instruction to produce a symbol structure that denotes thirty-one, since the term "the positive square root of 961" already does that.)

These issues are fundamental to any philosophically rigorous theory of computation. They also have directly to do with the meaning/mechanism dialectic discussed in the Introduction. And they not only permeate, but diabolically complicate, all the issues with which we are concerned here: about programs, semantics, and the design of reflective architectures. The semantical situation is particularly problematic. At a minimum, semantics rests on (among other things) a distinction between the realm of the sign and the realm of the signified—between name and named, representation and represented, term and extension, map and territory. This sign/signified duo is frequently

---

with a *type* of number.

That is not to say that abstract entities, such as genuine numbers (however conceived) may not themselves be used, in some circumstances, as symbols—as they manifestly are, for example, in diagonalization proofs of the Gödel incompleteness theorem. The point here, though, is merely that the *formal symbol manipulation* construal of computation rests on a notion of *concrete* symbols, not abstract entities of any sort.

(The suggestion of taking numbers to be types of numeral, one I frequently encounter among working programmers, is problematic for other reasons as well. It raises the issue of whether the number of which the binary numeral '110' is a token of is a *different number* from that of which the decimal numeral '6' is a token, for example. One might attempt to address that issue by taking numbers to be *more abstract* symbol types, so that tokens of the binary '110' and of the decimal '6' could be taken to be [tokens of] the "same number," of which the number [numeral type] of '110' [binary] and [decimal] '6' were subtypes. But that way lies madness.)

amplified with at least one other intermediating realm of sense, meaning, or intension. Cross-cutting that issue, it is also a widespread practice in logic, philosophy, linguistics, and computer science to construct mathematical models of these realms, including of signs (symbols or expressions—often called 'syntactic models'), and frequently of signified ("semantic models"), and of any intermediating realms of sense or meaning or intension.[4]

Moreover, if this were not enough, the prospects for confusion are exacerbated by yet another confounding complexity, largely orthogonal to either of the former two.[5] Like everyone else, programmers and computer scientists subscribe to a wide variety of metaphysical views and concomitant philosophies of mathematics: Platonism, realism, idealism, formalism, intuitionism, constructivism, species thereof, and likely other variants without common names. It is hardly surprising that, even in the human case (leave computers aside for a moment), answers to the question of whether one computes a number or a numeral would garner different responses from a realist, intuitionist, formalist, or constructivist mathematician—perhaps even from a phenomenologist.

And to put a nail in the coffin: the realm of computing, and of computational theory, is often looked to for resources to illuminate, clarify, and sort out many such confusions. If our understanding of computing itself is plagued with unclarity, ambiguity, and divergence of opinion, far from clarifying relations between meaning and mechanism, it will merely contribute to their mystification.

For present purposes one moral at least is straightforward. In my judgment, any attempt to clarify the

---

[4]Where, perversely, they are also called "models of the sentences," which to the uninitiated is probably best understood as shorthand for "models of the semantic interpretation of the sentences."
[5]Only partially orthogonal, needless to say!

fundamental nature of computing, semantics, programs, and the like must, to the maximum extent possible, avoid mathematical entities, examples, models, and techniques entirely. "Forget numbers; think about potatoes," as we might say, with Steinian apology.[6] As a corollary, I believe it is also salutary, at least in such investigations' initial stages, to avoid *documents*, *images*, *files*, *characters*, and other semi-abstract entities, where the intertwined issues of identity and concreteness bedevil analysis (How many letters are there in the word 'Boston'?)[7] For present purposes, therefore, I will focus on examples of programs such as those that control elevators, identify people, and perform side effects on (concrete) data structures.[8]

V1 Formal Symbol Manipulation

Start, then, with the formal symbol manipulation (FSM) view, as espoused in philosophy of mind and classical artificial intelligence. It has the non-trivial merit of providing a relatively clear answer to the first question, Q1, regarding the nature of computation. As I have said several times, computation, on the FSM account, is a process or behaviour that results from "formally manipulating" an array or assemblage of symbols, where symbols are semantic entities (i.e., entities that signify, denote, can be semantically "interpreted-L," and so on).

It is sometimes imagined that the symbols must be *formal* symbols, as if the correct parse of the identifying phrase were

---

[6]Stein (1931).

[7]It is not that I think cases of documents, images, communication, etc. (or, for that matter, genuinely mathematical task domains) are unimportant. On the contrary, they are *so* important, particularly in the computational realm, that they are the primary motivating examples for the development of the fan calculus mentioned briefly in §3.3.

[8]Side-effects because they establish relatively clear identity conditions.

((formal symbol) manipulation)

but it is obscure to know what a *formal symbol* might be. Much better, in my view, to understand the position as

(formal (symbol manipulation))

understood in turn as consisting of two essential parts. First, as is classically assumed in philosophy,[9] and as further explicated in chapter 6, it is presumed that the symbols exemplify two sets of properties:

1. So-called *formal* properties, either themselves concrete, in the sense of being properties of concrete entities, or else realised in (perhaps higher-order or functional classifications of) physical properties of such concrete entities, such as being of a given grammatical type, or being constituted of certain concrete mereological elements;[10] and

2. *Semantic* properties, such as denoting Mars, being true, or signifying the demise of the Egyptian Pharaohs.

Second, what makes the whole situation *computational*, on the FSM view, is that the manipulation process is constrained to respond to, and affect, properties only of the first, formal sort.[11] Formal properties, that is, or anyway all of those relevant to computational operation, are all assumed to be *effective*. Other assumptions about computation are paradigmatically made by FSM adherents, but in my judgment are not foundational to the view, at least in respect of the issues I am concerned with here.

Given this VI answer to Q1, turn next to Q6, about semantics. At least in outline, the situation is again relatively

---

[9]E.g., Fodor (1980).

[10]Leading the whole question to recurse, needless to say, as to what constitutes mereology—what sorts of *parts* are licensed?

[11]Fodor (1975, 1980).

straightforward, or anyway theoretically familiar. I have already indicated that the symbols must have a semantic interpretation, either intrinsically or (more likely) by external assignment, in order for the view to have any substance at all.[12] Some seem to view such interpretation as optional (since the processing regimen cannot depend on any such thing—the whole point of its being "formal"). But especially if the characteristics deflected in the previous paragraph are admitted to be contingent, making semantics or even semantic evaluability optional evacuates the entire proposal of content—reducing it to something honesty would compel us to call *stuff manipulation*.

Finally, if somewhat more contentiously, and as reiterated throughout these pages, I believe it to be a deep fact about the FSM view that the semantical interpretation function be *deferential* in the sense identified in the Introduction, and again more fully explained in chapter 6. That is: the symbols must not only denote or be about, but also *be normatively accountable to*, entities or states of affairs that transcend the symbol's own local physical properties and the immediate consequences of those symbols' treatment by the manipulation routine. This is necessary in order for the governing normative conditions on the manipulation regimen to get any substantive grip. It is absolutely constitutive of the FSM model, be it of logic or language or computing, that "not anything goes" in terms of how the symbols are processed and what results are produced. This is why soundness and completeness are such profoundly important properties, so aptly distilled in Etchemendy's memorable framing:[13]

---

[12]In logic, the interpretation function is partially specified in the design of the logic—that constants denote objects, that property and relation symbols denote properties and relations, respectively, and so on— though the precise specification of what objects, properties, and relations they are taken to denote is left, as it were, to the "user."

[13]Personal communication. Whether Etchemendy said this with an

Soundness · *Wanting what you get*
Completeness · *Getting what you want*

"What you want" is cryptic for *what is semantically warranted or appropriate*; "what you get," for *what is produced by the process of formal symbol manipulation*. The properties are substantial only when the former (semantic worth or appropriateness) is not intensionally subsumed by the latter.

Suppose, in contrast, that, starting with a Meccano or Erector set, we were to call each individual structural element in the kit an "atomic symbol," and define its "interpretation" to be the space of configurations that it can occupy when attached to another part. The "interpretation" of complexes would thus be the spatial configuration of possible assemblies made out of them (constrained by whatever geometric constraints curtail the ways in which they could be put together). And then suppose we define a "formal symbol manipulation" regimen to put parts together subject only to those geometric constraints (i.e., to assemble pieces randomly, so long as they fit, can be positively connected, etc.). By stipulation, the resulting system would be sound and complete. And the analysis: *vacuous.* There being nothing deferential about the (so-called) interpretation, the normative conditions on the symbol's processing would be empty; the result, a mockery of the FSM idea.

Note, since issues of reflection will come up presently, that I have not stipulated that the semantic interpretation must reach out to a semantic realm "external" to the domain of symbols—though in practice that is almost invariably the case (with the exception of term models, which almost never count as deferential on my view). What matters is that there be "more" to the semantics than what happens, even in internal (introspective) cases. For example, one can

---

explicit nod to Ingrid Bergman or Dale Carnegie I do not know.

imagine a system in which symbols are taken to denote various non-computable properties of their collective arrangement, configured in such a way that completeness was impossible to achieve, and perhaps soundness substantially challenging as well, even though all references would at least in some sense remain "internal." But once again the awkwardness of the example illustrates the point: it is a deep and inalienable fact about the FSM conception that the semantics of the constituent symbols be capable of outstripping, at least in ontological character, and often in extension as well, what can be "locally and mechanically" accomplished by a physically realizable process defined over their formal properties.

What then is V1's view of programs (Q3)? Curiously, but as already mentioned, *programs per se* are not part of the FSM conception at all. Two different things are often informally assumed, I believe.

First, it is typically presumed that programs play a role in *specifying the formal operations* that the interior process makes in manipulating the symbols. Tellingly, however, what constitutes a "formal operation" in the case of programs is unclear. There are two ways of interpreting what the predicate 'formal' *applies to*, in regards to programs; and additionally, two ways of understanding what 'formal' *means*, leading to a space of four possible readings.

Re the latter issue, regarding the meaning of formality, most discussions analyse the notion in one of two ways, usually assumed to be extensionally equivalent: positively, as having to do (solely) with the syntactic or grammatical type or category or overall "shape" of the symbol structure; and *negatively*, as "not being semantic."[14] In other work I argue that what is right about the positive reading

---

[14]Cf. Fodor's "Methodological Solipsism" (1980), the *locus classicus* of where this issue is discussed.

ultimately reduces to *effective* or *mechanical*—a formal property, in this positive sense, is one that a mechanism (or physical device) can respond to, independent of whether that property is semantical or not. I.e., the notion of 'formality' in discussions of logic and allied fields is either equivalent to 'effective,' or restricted to that subset of the effective properties (of terms, expressions, constructions, whatever) that are relevant—e.g., compositionally—to its semantic interpretation (to its "semantic-L interpretation-L").

The negative reading, in contrast, I take to be a limit case of a fact underlying deference: that semantical properties are not constrained to be, and indeed are often not (and in paradigmatic cases of formal symbol manipulation *are* not), mechanical or effective in this sense.

As regards the former issue, of formality's application, the ambiguity arises from the fact that to say that a program is 'formal' is ambiguous as to:

1. Whether its execution engenders *formal operations on the symbol structures being manipulated*—paradigmatically, formal operations on the program's *data structures* (i.e., if one were to adopt the specificational view, where the program is taken to *designate* formal operations over data structures); or

2. Whether the running or execution of the program *itself* proceeds formally, in the sense that whatever behaviour the program specifies or engenders arises based on formal properties *of the program itself* (as a symbolic structure), perhaps independent of *its* semantics.

One might imagine it to be overwhelmingly likely that programs must be (or are anyway understood to be) formal in both respects, if the process is to be automatic. From the point of view of the FSM model of computation, however, what matters about programs, to the extent that they enter the picture at all (or rather: if we forge a conception of

them consonant with the FSM model), is the first issue: that they *specify or engender formal operations*. How those operations come into being is not something on which the FSM conception itself takes a stand.

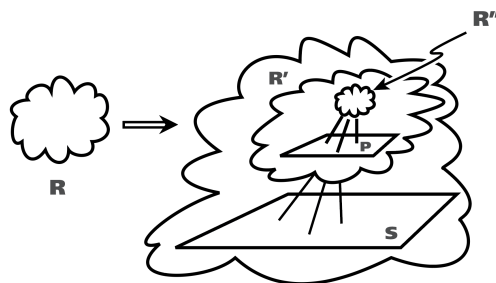What the discussion also makes evident is that the very idea of a *program* implies that the overall situation is perhaps more complex than is usually realized: that the classical pictures implies the existence of three distinct processes, as depicted in figure 4:[15] (i) the overall resulting process, labeled R, assumed on the FSM view to be constituted from the formal symbol manipulation of an assemblage of ingredient symbols; (ii) an interior process or locus of agency, R′, that does that manipulation—i.e., that operates "over" or "on" those symbols, which in contrast to it are viewed as relatively inert or passive; and (iii) an additional process, R″, not normally explicitly theorized in philosophical discussions of formal symbol manipulation, including in discussions of logic, but of central relevance in computer science, which takes the program P and generates process R′ from it.

Figure 4

In the case of an "interpreted" programming language ('interpreted,' in this case, not in the logico-semantic sense in which the term is used in logic—not interpretation-L, as I am calling it—but in the sense I am calling interpretation-C: the sense used in the specificational reading of programming-languages, familiar to computer scientists in such

---

[15]Adapted from figure 5 in the Introduction of Smith (1982).

phrases as "Java compilers produce bytecodes for an *interpreted* language"):

1. R″ would be the language "interpreter-C";

2. R′ would be the running program, reified in a way that (rather unusually) maintains a distinction between it and the data structures or symbols that it manipulates; and

3. R would be the overall process or behaviour that results from their interaction.

Whereas R″ (e.g., the Lisp "interpreter$_c$") and R (the overall running program) are the two processes most likely to be individuated as such, especially by programmers, what is striking is that, according to the FSM model, it is R′, *and only R′*, that "operates over the symbols," and that is constrained to do so formally. On the other hand—an important caveat—if, which almost undoubtedly matches programmer intuition more closely, one erodes the distinction between the program itself (P) and the data structures on which it operates (i.e., the "symbols" S, but here calling them data structures to align more directly with computational parlance), then R″ becomes the process required by the FSM model, with R″ formally manipulating the program-cum-data-structure amalgam so as to engender R.

If program P is *compiled*, there is a change to the identities of the program being executed and the "doubly interior" process R″ that does the execution, but the tri-partite process structure remains, as do the identities of R′ and R. In particular, the compilation process will take P as input, and output a different program P̄ (most likely in binary machine language, or in byte codes for an underlying virtual machine), able to be "interpreted-C" by a different runtime process R̄″ (the underlying CPU, if P̄ is a machine binary, or a "byte code processor," if P̄ consists of a sequence of byte codes), in such a way as to honour this condition: the process or behaviour R̄′ that results from R̄″ (the CPU or byte code processor) interpreting-C P̄ (the "compiled code")

must be "the same" as the process R′ that would have re-
sulted from the language interpreter R″ interpreting-C P di-
rectly. "The same," here, is defined in terms of an *outer*
equivalence: the overall process or behaviour R̄ that results
from R̄′ interacting with the data structures S must be
equivalent to the overall process or behaviour R that would
have resulted from R′ interacting with those same data
structures—on an appropriate notion of equivalence, which
commonly does not include timing, to allow R̄′ to be faster
than R′.

Spelling it out in this way may be more tedious than
helpful; certainly the points being made are second-nature
to any working programmer.[16] The distinctions start to
matter, though, as soon as we turn to issues of semantics.

Start with the process-world relation (Q4). This is the clas-
sical realm of semantics for logic, and by extension for the
FSM view as a whole. Moreover, as already indicated, in or-
der for the FSM view to have substance (and not to reduce
to stuff manipulation), the semantic relation must be def-
erential, in order for the symbol-manipulation process R′
to be normatively constrained in an appropriate way. Pro-
cess R″, we can assume, is given in advance (a Lisp inter-
preter$_c$, a CPU, whatever). The relevant normative con-
straints are focused on program P: it must be such that, in
manipulating symbols S, it does so in a way that is seman-
tically appropriate overall—"sound," in the logical tradi-
tion, though as we will see, the form that the constraints
take in the computational case may be more complex. This
is all commonsense: if one develops a knowledge represen-
tation (KRL or Mantiq or any other), one must write

---

[16]The degree to which they may be unfamiliar, if not opaque, to philo-
sophical readers may be indicative of the extent to which the actual na-
ture of computation remains to be adequately theorized in contemporary
philosophical analysis.

programs P to manipulate the knowledge representation structures in such a way as to "make sense" (i.e., honour the semantic-L content, in an appropriate way) of those structures, in terms of their (perhaps attributed) semantic interpretation-L.

At the most fundamental level, the process-world relation at issue in Q4 is that between R and its task domain. According to the FSM view, normative responsibility for this relation is placed on both S and R′.[17]. Suppose structure S_I (∈S) represents the fact that water flows downhill if possible and otherwise accumulates, and S_2 (also ∈S) represents the fact that some particular land region α has no downhill outlet. And suppose in addition that, based on these S structures, process R′ yields the conclusion S_3 (again ∈S) representing the fact that water accumulates at α—i.e., that α is a lake. Then it is a normative condition on S that S_I and S_2 should be *true*; if S_2 is wrong, and α is in fact the side of a mountain, then R, in yielding S_3, has reached a wrong conclusion on account of a failure in S. If, on the other hand, S_2 is correct (R indeed has no downhill outlet), but the program P licenses the conclusion that α is a river (rather than a lake), then R has again reached a wrong conclusion, but this time on account of a failure in P (P is unsound). In the vernacular of folk psychology, or anyway philosophy's reconstruction of folk psychology, the first is a failure of *belief*, the second a failure of *reasoning*; it is undoubtedly no accident that the S-R′ distinction mirrors this intuitive distinction so closely.

---

[17]These are the two normative conditions on logic discussed in chapter 6: one on sentences (S), that the sentences be *true*; the other on inference (⊢ in logic, R′ here), that it be *truth-preserving*.

Normative responsibility would be shouldered by S and R′ only if the computational parts of this assemblages were capable of being ethically responsible agents. The normative responsibility, therefore, is shouldered in the first instance by the programmers, and behind them by the institutions and social practices within which they are embedded.

What else, then, since we are now well into question Q3, can be said about program-process relation P→R, on this FSM (VI) construal, besides the fact just noted: that P is indirectly subject to the normative constraints placed on R by the deferential semantics of S? In particular, can we say anything about the *semantics* of P, viewed as a symbolic or signifying structure? This is the question we must be able to answer, if we are ever going to understand reflection. And the curious fact is this:

> *On the issue of the nature and (what computer science would call) the "semantics" of* programs, *the FSM construal of computation is completely silent.*

All we have been able to determine as constitutive of the FSM view is: (i) that the symbolic structures in S must be (formally) manipulated by process R′ in a semantically appropriate way; and (ii) that if the behaviour of R′ is determined by something called a 'program,' in conjunction with another interior process R″, then that program must engender such semantically appropriate behaviour *somehow*. But for all the FSM view of computation cares, the program could do it by might alone, or by divine intervention, or on a whim. Nothing in the FSM view requires the program to be a *semantically interpretable* structure or expression at all.

One natural extension of FSM to deal with this situation has already been suggested: integrate the program, viewed as a symbolic structure in its own right, into the field of symbolic structures S, and interpret it in accord with the same semantical framework that is used to interpret the rest of the expressions in S. It would be natural, if the program was framed in such a way as to describe or represent how other symbolic structures in S were to be treated, then terms in the program would refer to or "mention" other elements in S—making the expressions in the program meta-level structures. What this suggestion amounts to, however, is *no more than the representational/ingrediential view of programs explored in* chapter 3; *it is also the approach* (V4)

under which 2Lisp and 3Lisp were designed.

As I will argue in more detail below, this suggestion—of treating the program as an ingredient within the field of data structures, semantically interpreted (interpreted-L) in the regular way, but including (meta-level) references to other data structures—has numerous benefits, including: (i) obviating the need, which we have already recognized as awkward, of reifying process R′ as a distinct intermediary process between R″ and R; (ii) allowing program P to *use* (not just mention) "object-level" or "base-level" terms to refer to external entities, such as numbers and sequences and such, in the same deferential way as any other symbolic structures in S;[18] and (iii) dissolving the issue discussed above, about whether the program itself is dealt with formally, or whether the program leads to formal operations on data structures (both views become true, essentially automatically, when the program is taken to be a process ingredient).

For example, consider a program that contains a statement such as «(IF (AGE-OF(COMMITTEE-CHAIR)==47) THEN …)» which *uses* data structures to *mention* elements of the task domain (the chair of the committee, the number forty-seven, etc.). That is not to say that symbolic elements of P may not refer to other elements of S (rather than D), using quotation, mentioning them by name, etc. As the 3Lisp architecture makes evident, making this approach work mandates the use of a certain amount of straightforward meta-structural machinery. But the bottom line is simple:

> The FSM view of computing (V1) leads naturally to an ingrediential view of programs (Q2), in the process answering the question about the program-process relation (Q3): P derives its semantics (not just its normative constraint) from the semantics defined over S

---

[18]I say 'other' in this context because the idea is that, on this approach, the program P has been included as *part* of symbolic field S.

(Q6), in virtue of being considered to be *part of* S.

V2 Theoretical computer science

What then of the view of theoretical computer science?

The stunning fact—and the reason this entire exercise is important—is that this second construal's answers to the first six questions differ profoundly from those of the first (V1, or FSM). To see this, keep in mind that this second perspective (V2) is about programs, computing and the like *as theorized in computer science.* When we turn to V3, I will argue that this V2 theoretical understanding is at least augmented, in the minds of virtually all computer scientists, or anyway all programmers, by an additional layer of tacit understanding, essential to the task of programming, which pulls it closer to that of the FSM (V1) view. I will even argue that the tacit view changes the theoretical nature of the predicate 'computational,' in ways that are conceptually important. But if we restrict our focus to the currently accepted theoretical approach, the differences between V1 and V2 are stark.

Consider question Q1, first, about the nature of computation. One might expect computer science to be committed to a conception of computation as something like symbol manipulation or information processing[19]—i.e., to take computing as having something to do with meaning, semantics, or intentionality. But I do not believe that is correct. Although, as we have seen, computer science's theoretical language is drenched in semantical vocabulary, betraying its logical origins, I have come to believe that the pressures of a reigning mechanistic philosophy, coupled with the idea that science should traffic in causal explanations, has eviscerated the core of computer science of all commitments to anything I would countenance as

---

[19]'Information' is as multiplicitous and overloaded a term as any other in computer science.

genuinely semantic (semantic-L), pulling the entire field of computing inside of what I am calling *blanket mechanism* (by "the core of computer sciences" I especially mean to exclude AI, knowledge representation, and some approaches to database theory).[20] This is manifest in recent theorizations of information, stemming originally from computational quarters, which not only take information to be a measurable quantity, which already starts to lean away from intentional concerns, but defines it entirely mechanistically, in terms of something like the "amount of effective complexity"—e.g., in the Kolmogorov and Chaitin conceptions of the "amount of information" in a signal or message being equated with the length of a maximally compressed version of that structure *syntactically* construed.[21]

As regards the underlying "theory of computing," more generally, and the entire enterprise of theorizing programs and programming languages, including the entire enterprise that goes under the label "programming language semantics," all senses of *deference*, as I am using that term, have at least been relegated to the wings, if not eliminated entirely. Falsehood, incompleteness, unsound reasoning, etc., are similarly off-topic (not that entities exemplifying such properties cannot be treated as computational structures of some sort, but the perspective we are considering here renders invisible their falsehood, incompleteness, unsoundness, and so on).

The easiest way to see this is by considering the hierarchy of machine types in terms of which general computation is often introduced: finite state machines, push-down automata, and Turing machines—the latter two consisting of finite state machines plus internal 'memory': a pushdown stack in the former case, an unbounded tape in the

---

[20]Cf. chapter 7, and also Piccinini (2015).
[21]Kolmogorov (1963,1998); Chaitin (1969); Levin (1980,2010); Burgin (1982).

latter. Note, for starters, that the *states* of the constitutive finite state machines are in general *not* semantically interpreted. They are atomic "total conditions" that the machine can be in, whose only defining (and usually individuating[22]) characteristics are that, when presented with effectively different inputs, they respond differentially, transitioning to a new state, and possibly generating an output while doing so. From the point of view of the formal theory,[23] no assumption is made, nor any implication even mooted, that these states *mean*, *represent*, or *stand for* anything. They are not considered to be of an appropriate type for having any visible content or semantic value. (As explored in more detail in chapter 7, one does need to interpret them in order to say that they are capable of arithmetic operations, such as "adding binary numbers," itself a semantic mongrel of an idea.)

More relevant, for our purposes, are the inputs and outputs such machines respond to and produce, and the structure of the internal memory or tape employed in machines of the latter two types. Here there are two options, both of which can be found in explications of underlying theory. From the point of view of the ontology of computing, they seem quite different, although I will argue in a moment that the apparent differences are relatively superficial.

Perhaps most commonly, inputs and outputs are characterized as "formal symbols"—but not, in this case, in the sense that they can be written, responded to, and manipulated *independent of semantics that they nevertheless necessarily possess*, in the way that I argued to be essential to the

---

[22]There is some debate about whether functional/behaviour indistinguishability or topological similarity should be used to individuate finite state machine types.

[23]'Formal' as a predicate on theories is usually associated with the theory being mathematically expressed, which is a distinct notion from the non-semantic, "in virtue of shape" idea associated with formal symbol manipulation. See "Foundations of Computing" (Smith, 2002).

formal symbol manipulation view. Rather, in these automata-theoretic cases, for all intents and purposes the symbols are *not considered to have any semantic interpretation at all*. That is, to use the "stuff manipulation" epithet suggested above, on the automata-theoretic view, inputs and outputs are merely "(discrete) mechanical stuff." The "mechanical stuff" may be theorized as abstract, with the constraints of mechanism shouldered by a mathematized version of the property of being "effective," but as I argue in chapter 7, I believe that abstract considerations of computational effectiveness are merely relatively abstract physical constraints theorized in a mathematical model.

The other option makes the point even more obvious: computation in computer science is sometimes characterized directly in terms of numbers (i.e., the inputs are taken to be actual numbers, not numerals or other symbols representing numbers; the outputs are similarly numbers; numbers are placed into appropriate position in "memory," considered as an abstract mathematical structure; etc.). Under this abstract mathematical conception, issues of semantics and interpretation do not even arise.

In FSM and logic (i.e., in VI), to use an analogy I have employed elsewhere, semantics plays a role not unlike that of alcohol for the temperance union. There is no question about its existence; on the contrary, that existence is the field's defining *raison d'être*. Far from being "out of mind," semantics is very definitely *in* mind, but for behavioural purposes *decisively set aside* (as a mechanical cause of operation, in the FSM case; as something to imbibe, for teetotalers). As mentioned several times already, even if behaviourally eschewed, semantics nevertheless continues to play a decisive role, in the FSM view, by *normatively governing* that behaviour (in licensing what formal transitions are "good," in the machine case).

In the automata-theoretic case, in contrast, semantics is not just set aside, but swept off the table completely, dismissed from analytic imagination. The situation is not one

of its existing but playing no mechanical (causal) role, as in the FSM construal, but rather of its not playing any theoretical or normative role at all. In this way, far from being like the teetotaler's alcohol, it is more like how physicists view the luminiferous ether, or might view the political leanings of the mythologically objective experimenter—something that either does not exist, or if it does, is for theoretical purposes irrelevant. As far as the mathematical theory of computation is concerned, computational machines simply "do what they do," based on how they are constructed or defined—unjudged and uninterpreted by any external constraint.

Two facts can mislead an outsider into thinking that, in the received theory of computing, semantics or interpretation is still involved:

1. The range of inputs acceptable or "recognizable" by an automata (especially in the case of finite-state and pushdown automata) are often characterized in terms of *grammars* of distinctive types. Grammars, in turn, are invariably taken to be grammars of *languages*. But for purposes of the associated mathematical theory, these are "languages" in form alone; no reason is given to suppose that expressions in them *mean* anything. In terms of the Meccano analogy mentioned above, one could as well have a "grammar" specifying the ways in which Meccano parts can be assembled. Calling these finite, recursive characterizations of unbounded combinatoric spaces "grammars" is at best misleading, in my view; at worst dishonest. Meaning is not just kept "out of bounds," but disregarded to the point that nothing hinges on whether it exists at all.

2. As we will see when we examine V3 (programmers' tacit view), in real-world practice—in what I call *computation in the wild*—computers' inputs, outputs, and memory states generally *are* interpreted

(in the philosophical sense of interpreted-L): taken as being representations or encodings of information about, entities, phenomena, and states of affairs in the task domain for which the system has been constructed, as well as representations of and information about other internal states (which will be relevant when we discuss reflection). In fact I will argue that *no programmer could write or comprehend a program of substantial complexity if they were not so interpreted*$_L$.[24] What matters for V2, though, is that the mathematical theory of computation does not even recognize, let alone treat, these "into the world" forms of intentional directedness.

In sum, theoretical computer science theorizes computation as moves between and among abstractly specified states of equally abstractly specified machines. In my view, it is a mathematical theory of abstract mechanism—hook, line, and sinker.

But what then about *programs* (Q2)? Are they not intentional symbolic entities, subject to semantic interpretation? After all, is that not what programming language semantics is about?

Finally we are getting somewhere.

Note, first, that the so-called mathematical theory of computation—perhaps better (if still not yet adequately) labeled a "mathematical theory of *computability*," and extended to include associated theories of computational

---

[24]The following more general statement used to be largely true: "no programmer can build or comprehend a system of substantial complexity if it is not so interpreted." But, as is most obvious in the case of deep learning systems, programmers now write programs that effectively "build systems," often based on massive amounts of data, where the resulting system (as opposed to the program that generates it) may or may not be interpretable.

complexity—by and large deals with *algorithms*, rather than with *programs per se*. While there is no generally accepted theory of exactly what constitutes an algorithm—in particular what criteria play the role of algorithmic individuation, in order to say of one algorithm whether it is the same or different from another—an algorithm is generally described as a "way of doing something." The "things," for the doing of which algorithms provide a way, can in general include just about anything: sorting socks, deciding who should be admitted to university, predicting the outcome of an election. But in line with its mathematical character, algorithms in theoretical computer science are generally framed (perhaps modeled) either in terms of pure mathematical entities (numbers, functions, etc.), or straightforward encodings of such mathematical entities in "formal" symbols.

The most famous open problem in computer science, applicable to the question of whether the prime factors of a number can be "computed" in nondeterministic polynomial time, looks, superficially, as if it is an abstract problem about numbers. In point of fact, however, both the problem itself, and what would count as an acceptable answer, are relative to the way in which the numbers are encoded or symbolically represented. (If numbers were represented by concatenations of those numerals that designate their prime factors, then "factoring" them would be instantaneous—and trivial.) Somewhat surprisingly, the question of what constitutes a legitimate encoding of numbers, functions, etc., for purposes of this theory, remains largely untheorized.[25]

For some purposes, such as calculating complexity, algorithms are the natural subject matter—and in such endeavors, issues of symbolism and semantics do not arise.

---

[25]See "Solving the Halting Problem, and Other Skullduggery in the Foundations of Computing" (Smith, 2014).

Programming language semantics, however, *does* deal with programs, not just algorithms, and *appears* to treat them semantically: assigning them meanings, denotations, etc., in the familiar compositional way.

This has been our target since the beginning; it has been a long road to get to this point.

What, then, can we say about the V2 perspective, with respect to our suite of questions? (Keep in mind, as always, that V2 is currently accepted *theoretical* practice in computer science, not the tacit understanding of programmers, which will be dealt with separately, under V3, below.)

First, as indicated earlier, data structure identifiers are treated (in this theory) as names of memory locations, or as names of entries in a mathematically-modeled "store," not as designators of what a given program uses those memory locations to represent. Imagine, for example, that a program for an elevator controller contains a constant called «NUMBER-OF-FLOORS», and two variables called «CURRENT-FLOOR» and «FLOORS-TO-STOP-ON-WHILE-DESCENDING» (bound to a list or sequence). If the program were to encounter an instruction such as

```
current-floor ← current-floor − 1
```

one might reasonably assume that the variable 'CURRENT-FLOOR' was being *used* (in the logically technical sense) to denote or refer to the floor at which the elevator was currently stopped (was passing by, whatever). This "interpretation-L" is of exactly the sort that was addressed in V1, and will be addressed again when we get to V3. It is not, however, what would be theorized by programming language semantics as currently conceived (V2). Rather, 'CURRENT-FLOOR' would be mapped, by the official computer science semantical account, onto either: (i) a memory location used to store information about the current floor; or possibly (ii) an integer, if for example the variable 'CURRENT-FLOOR' was (had been declared to be) of type "fixed integer."

Two issues need to be explored: (i) relations among the "storage location," any "mathematical" entities onto which these program entities are mapped or with which they are associated, the "number" or "numeral" that might be stored in that location, and the floor on which the elevator is destined to stop; and (ii) the consequences of those choices for our general understanding of computation, particularly as relevant to an account of reflection.

As regard the first issue, the point is elementary. As we have already seen, a dialectic exists between viewing computation as concrete or as abstract. Theoreticians often treat it as abstract; my own view is that the mathematical structures employed for theoretical purposes are (abstract) *models* of (concrete) computation, not the "real thing"— primarily because I do not believe that the computability and complexity results, the *sine qua non* of any substantial theory of computability, can be defended in other than concrete terms. In order to remain as neutral as possible about this issue here, however, in what follows I will refer simply to "the computation," using the term to cover both the abstract and concrete conceptions.

Given that usage, the most important fact about programming language semantics can then be stated very simply: programs are taken to be compositionally-constructed "symbolic" structures, the execution of which leads to *computational behaviour*. An assignment statement, for example, of the form

$$VAR \leftarrow EXP$$

engenders a change in the state of the computation, such that the "binding" of the variable *VAR*, in the resultant state, is to the "value" of the expression *EXP* in the prior state. All these entities—the variables themselves, their "values," etc.—are *elements of the computation* (abstract or concrete). No attempt is made to assign, as the value of a variable, anything external to the computation—employees, salaries, elevator states, floors, or anything of the sort. In fact

to a programmer that would seem like a conceptually mistaken idea: the values of variables must be computational entities, on the received view.

More generally, the "denotations" or "semantic values" of all expression types—including not just this assignment statement, but also all other composite forms, not just variables or the two ingredient expressions in this example, *VAR* and *EXP*—are understood to be the elements within the computation to which the expressions are taken to "refer" or "denote," or the computational element that they return, upon being executed, or activity of change that their execution entails, or some other internal phenomena of that sort. That is: the focus of the analysis is on *internal behaviour*. Informally, if a programmer asks "What is the semantics of that construct?," what they are asking, in effect—or at least what theoretical practice in computer science takes them to be asking—is *What will happen, expressed in terms of this mechanism in front of us, when this program fragment is executed or run?* Or to put it in another way: as described in programming language semantics, the "semantics" of a program expression or fragment has to do with *how it will be processed, mechanically or effectively* ("how it will be processed, syntactically," an adherent of VI might say).

Complicating the issue for readers from other disciplines, analyses of programming language semantics can be conducted in a variety of styles, known as *denotational*, *operational*, *algebraic*, *axiomatic*, etc. As mentioned at the end of the last section, a philosopher or logician might imagine that "operational" semantics would analyse the mechanical consequences or implications of an expression or structure (i.e., something like the proof-theoretic implications, or the mechanistic consequences of its syntactic profile), and that "denotational" semantics would analyse what it refers to or denotes, in the sense that those terms are used in philosophy and the FSM tradition (i.e., would analyse its semantics-L). That is, one might expect that the normative constraints on the operational semantics would be that it

should *honour* or *defer* to the denotational semantics, even if, perhaps in the presence of prevailing incompleteness, it was unable to encompass it entirely. That is, the philosopher or logician might expect a study of the *operational semantics* of a program or program fragment to be a study of those syntactic or mechanical operations that are normatively bound to honour the semantical (semantical-L) facts that would be revealed by an analysis of that program or fragment's *denotational semantics*. These things would be expected on an understanding of operational semantics as being analogous to proof theory; denotational semantics, to model theory.

It is not so. Rather, within computer science, all these forms of analysis (denotational, operational, algebraic, axiomatic) account for the same thing, but in different ways: the mechanically-produced behaviour that results from "executing" or "running" the program. Though differing in modeling techniques, they coincide on (have identical) subject matters—a subject matter, in fact, that is closer to what is studied in proof theory than anything that in the logical tradition would ever be called semantic. That is why their results are—or anyway should always be able to be—proved equivalent: not because the system is sound or complete, or is assumed to be sound or complete (notions of soundness and completeness do not even make sense in the accepted theoretical context of computational semantics), but because they are *different ways of analysing the same thing*.

I take the problems with this approach, with respect to an overall view of computation as an intentional process (necessary in order to define reflection) to be self-evident: there is nothing deferential about such an account; there is nothing about it which explains how the computation is intentionally and normatively directed to a task domain or subject matter. What is called the "semantics" is merely one dimension of the phenomenon—the mechanical, operational or effective one, contrary to what a logician or

philosopher would expect—whereas in the general semantical model rehearsed in the examination of the FSM view, and described in more detail in chapter 5, there were invariably *two*. Because the official computer science (V2) version is unidimensional in this sense, there are no normative constraints—nothing for notions such as soundness and completeness to get a grip on. We are back to the realm of abstract Meccano: compositional mechanisms whose behaviour arises in a systematic, causal way from the nature of their ingredients. *C'est tout.*

As an example to illustrate the difference between the theoretical computer science (V2) approach and the FSM (V1) view, and to foreshadow our discussion of the tacit view of programmers (V3), below, consider the following flawed code fragment:[26]

```
L1)   (define factorial
L2)     (lambda (x)
L3)       (if (= x 1)
L4)         1
L5)           (factorial (- x 1)))))
```
$\left.\right\}$ ⇐ incorrect

Obviously enough, the code contains a bug; the fifth line should instead be:

```
L5')              (* x (factorial (- x 1)))      ⇐ correct
```

What matters for present purposes is the discrepancy between the intuitive understanding of the example (subject of V3) and that provided by the programming language

---

[26] In Python:

```
L1)  def factorial(x):
L2)       if x == 1:
L3)           return 1
L4)      else:
L5)           return factorial(x – 1)
```
and:
```
L5')           return x * factorial(x – 1)
```

semantics account (V2, under analysis here). We humans know that the code is faulty, because, for us, 'factorial' is a term in our language, established by long-standing social convention, mathematical practice etc., that *denotes the factorial function*. When, as programmers, we write code such as that above, we are attempting to write a program that will calculate values of the function, called factorial, that we antecedently understand (or at least produce numerals that denote the numbers constitutive of that mathematical function). That is why we can say that the code fragment is *wrong*. According to the programming language semantics tradition, however, lines L1–L5 would simply be characterized as an (unproblematic, if inefficient) definition of the constant function 1 over all positive integers (technically: a function over the those numerals that designate the positive integers. All the programming language account can do, that is, is to provide an analysis of how the foregoing code fragment will *behave*, independent of anything that we humans think its identifiers *mean*. As such, it is not its semantical task to label the fragment wrong; that is left as an "extra-theoretical" exercise.

How *could* theoretical analysis accommodate our understanding of the denotation of the term 'factorial', a computationalist might ask? Sure enough, theoretical analysis cannot know, especially in advance, what we have in mind; that much is uncontestable. But that does not mean that theory is left empty-handed. As explained in chapter 6, logic shows a strategy: theoretical analysis need not itself be a closed, formal system. The logical tradition of semantical analysis encompasses, and makes rigorous, an *externally-supplied* understanding, paradigmatically by including (a formalization of) our pre-existing or at any rate externally supplied "interpretation-L" of the terms in virtue of which the system under investigation is understood. That is, as I have said several times, the analytic semantical frameworks of logic are *parameterized* on the interpretations-L of all constants in the language; that is the task of

the "user-supplied" interpretation-L function.

This is why the appropriate constraint to place on an inference (inference-L) system (the regimen of moving from one set of symbols to another, given such external parameterization) cannot be that it be true or correct—more carefully: cannot be that it produces only true and correct results—since to know whether something was true or correct would depend on substantial ('material') semantical facts about the externally-supplied interpretation function. Rather, the formal analysis is restricted to something we have already seen, captured in the logical notion of *soundness*: a normative condition that *if* the material statements made by the user are true, then the results that are yielded by the inference regimen must be true as well. By analogy, if we had what logic (∨I) would take to be a semantic analysis of the foregoing code fragment and associated programming language, the mandate on the inference system would not be that it provides a *correct* implementation of the language, but a *sound* one.

The problem with code fragment L1–L5 could then be stated very simply: it is *false*. The factorial function is *not* as described on the right-hand side; since that complex expression denotes the constant function 1, which is not factorial (not what the term 'factorial' denotes, the term on the left-hand side).

A deferential analysis of what we call procedure "definitions," that is, would treat such constructions (of which L1–L5 is an example) as something like a *double claim*:

1. *Extensionally*, the term on the right (the λ-term in L1–L5) denotes the function named on the left; and

2. *Effectively* (or as some, but not I, would say, *intensionally*), the λ-term on the right specifies an effective way of "computing" that function, with respect to an assumed processing regimen (i.e., with respect

to process R″, in the vocabulary of §4.vɪ),[27] where "computing" is understood to be something like coming up with a normal-form or otherwise distinguished designator of the value of the function on the element designated by the symbol used when it is "called."

In the analysis of ᴠ4, the theoretical framework underlying 2/3Lisp, I will argue that calling the second reading 'intensional' is both incorrect and misleading. Nevertheless, I believe that something like this double claim version of so-called "definitions" is in fact the way in which programmers understand their code. In fact, as I will argue when we analyze ᴠ3, this double-claim view is almost necessitated by the tacit view of programmers—necessary in order for a programmer to believe that there could be a *bug* in their code—for what is a bug, other than a discrepancy between the behaviour that the code, as written, effectively engenders, and the behaviour that the programmer *intended* it to engender?

In sum: contemporary theoretical computer science, including all styles of programming language semantics (denotation, operational, axiomatic, and algebraic):

1. Takes computation to consist of *state changes of abstract mechanisms or machines*, together with their associated inputs and outputs, where the inputs, outputs, and the state of any involved "memories" are treated as uninterpreted (either empty symbolism, or empty symbolism mathematically modeled, or mathematical entities in their own right);

2. Views programs as structured "grammatical" complexes that behaviourally cause or engender or in

---

[27]And also with respect to an assumed way of *representing* values of the function—as numerals, we can presume.

some other effective way impinge upon the resulting computational process, where the net behavioural consequence, though analysed compositionally, is specified without regard to what I am calling semantical-L interpretation (i.e., without deference to a larger world or task domain, based on external assumptions from a wider use of language, etc.);

3. Does not address the process-world relation R⇒W at all, except in as much as it accommodates inputs and outputs, which are taken as uninterpreted tokens that "cross the (effective) boundary" of the machine; and

4. In spite of its pervasive use of semantical vocabulary, neither involves, nor admits, nor theorizes, nor in any other way substantially deals with, the sorts of non-effective, deferential relations on which, I am claiming, the notions of intentionality and semantics ultimately rest.

Because I take reflection to be a substantial intentional phenomenon, *crucially involving deference*, it follows, at least in my view, that mathematical computer science, as currently conceived, is not a strong enough theoretical framework in terms of which to define, analyse, or understand reflection—and therefore cannot serve as an intellectual framework in terms of which to design a reflective language. The following, that is, is the V2 answer to question Q7:

> *Genuine reflection will remain forever invisible, from the point of view of computational theory as currently formulated.*

Does this mean that V2 is without merit? Far from it. Much more than habit and familiarity recommend the view. For one thing, it is purpose-designed to facilitate construction and implementation, against a background of well-understood programmer intent. Moreover, as will be

argued in §4, V2's internally behaviourist perspective also allows powerful (and disruptive) ontological insights and intuitions to be tacitly accommodated within it—in ways that enable programmers, and computational systems, at least partially to escape the confines of classical ("formalist") ontology, and perhaps also to escape equally restrictive (if they are taken substantially) constraints on compositionality. But to understand how that goes, we need to know more about how programmers actually understand the programs that they write.

V3 Programmers' intuitive understanding

The discrepancy between programmers' intuitive understanding of programs and the analysis provided in theoretical accounts of programming language semantics has surfaced several times—in the example of the elevator program, in the intuitive discussion of "computing" highest paid employees, and in the recognition that the initial (L1–L5) definition of FACTORIAL is incorrect. I have also suggested that this discrepancy is far from theoretically innocent. It is impossible to develop or understand a program of any complexity, I believe, unless one uses, as the names of the constants, variables, classes, procedures, etc., words that *signify*, in a semantically-L substantial way, what those entities represent, stand for, carry information about, concern, address, etc. No theory of the semantics of programs, I claim—indeed, no theory of programs at all—can pretend to adequacy unless it includes a theoretical analysis of that substantial semantic interpretation (interpretation-L).

Every programmer learns something that might be taken as countervailing: that English names (or names from any other natural language) *mean nothing to computers*; programs would compile, run, and "do what they do" just as well—identically, in fact—if all their labels were replaced with random symbols: «G001», «G002», «G003», etc., or any other distinct nomenclature. In the same vein, it

might be noted that the difference is equally invisible to the programming language semantics tradition, since it focuses exclusively on behavioural consequence, mechanically construed. That is: the attributed, referential semantics (semantics-L) of names that is crucial to programmers is "invisible" both to what I have called the language *processor*— the active locus of agency, called ʀ′ in §4.ᴠɪ, that manipulates them—*and to the standard theoretical semantical account* (ᴠ2).

Someone compelled by these observations might argue that using identifiers based on natural language is "purely a matter of convenience," of neither behavioural nor theoretical consequence.

That it is of no behavioral consequence might be taken to be what was just discussed: that is one way of understanding what it is for programs and computation to be "formal." Note, however, that any claim that the behaviour of the program is immune to semantics holds only on the presupposition that behaviour is identified (individuated) purely mechanically or mechanistically. That is, it is true only on a semantically *uninterpreted* notion of being "equivalent." If understood as semantically interpreted, the behaviour might be quite different. Suppose x and ʏ, a guilty and innocent party, are asked whether they committed a crime, and, each pointing to the other, say "No! I am innocent! They did it!" Have x and ʏ "behaved equivalently"? In one (uninterpreted) sense, *yes*, since they uttered exactly the same sequence of words. If their two responses were printed out, for example, the two typescripts would be indistinguishable. But in another (interpreted) sense, *no*, their responses were different; they disagree. If "behaviour" is understood "under interpretation-L," that is, their responses differ, since they refer to different people (one claims ʏ is the culprit; the other, that x is the guilty party); one has lied and one has told the truth; and so on.

Contrapositively, suppose one asks two physically separated witnesses where the crime was committed, and one

says "right here," and the other, pointing to the first, says "over there." It might well be natural, in such a case, to say that the two gave the "same answer," though in this case their words differed (and, similarly, to say that they gave *different* answers, if their words were the same). Similarly, when conducting a survey, it might make sense to group together, *as behaving in the same way*, those who plan to vote for an incumbent, no matter what words they use or how else they indicate that person; and similarly group together those planning to vote for a challenger, again independently of what words *they* use, even if they used the same words as someone else who is voting for the incumbent (e.g., if they both said: "the person whose rally I attended last night"). The point is that, when intentionality pertains, it is common for our ordinary understanding of 'behaviour' to reach out to encompass the semantic and referential field of the raw physical events, and to individuate behaviour *under interpretation*.[28] It is not clear what, other than prejudice, would prevent us from doing so in a computational case. To say that we should individuate behaviour mechanistically *because computation is a mechanical process* is simply to argue in a circle. Whether computing is constitutively purely mechanical, or whether it is a concretely embodied intentional activity, is exactly the issue.

That the use of meaningful identifiers in programs (and in their input and output) is of no theoretical consequence is arguably true given the current state of theory. *But so much the worse for the current state of theory*. What should matter is not what theory is *currently* like, but what theory *should* be like, based in turn on what is the case. And on that question my position should be clear: theory should be held accountable for (i) articulating the semantics of the computational ingredients, however those semantics arise; and (ii) accounting for the extent to which behaviour

---

[28]'Interpretation,' again, in the philosophical/representational sense.

honours the norms that the semantic interpretation establishes.

To open up the possibility that we might be able to develop a theoretical framework that does such justice, remember that it is a mistake to argue that world-directed semantic interpretation must be theoretically irrelevant because that kind of semantics cannot play an effective ("formal," mechanical) role in engendering (mechanically individuated) behaviour. Formal inference systems for logic (VI) are blind to the semantic interpretation of their constituent symbols, too, but it does not follow—in fact it is patently untrue—that the *subject matter* of logic and formal symbol manipulation is restricted to the arbitrary rearrangement of uninterpreted grammatical expressions. The study of logic is not the study of abstract Meccano. Rather, as I keep emphasizing, logic is—and logical theory a theory of—patterns of inference *normatively governed by the constituent expressions' semantic-L content*. This is why, as said above, the entire subject matter of (what we call) "formal logic" is not formal; rather, what is formal is its behavioral dimension—its projection onto the mechanical wall of the cave. One could claim that logic as a whole is formal only by pledging advance allegiance to blanket mechanism, banishing deferential semantics, and reducing logic to the blind manipulation of arbitrary inscriptions—exactly the approach I am disparaging as "stuff manipulation."

One way to see what is going on is to approach the point combinatorically. The number of possible machine states is vast (exponential in the size of the memory), of which we construct a vanishingly small subset. If, as I approximately believe, the work of programming is to construct mechanically effective systems that are normatively governed by their relations to their task domains, where "relations to their task domains" include not only behavioural interaction but also non-effective semantic interpretation—then it should be the job of an adequate theory (i) to take those interpretations into account, (ii) to theorize the systems'

mechanically-induced behaviour, and (iii) to theorize the ensuing normative accountability—all just as is done in theories of soundness and completeness in logic.

What have we established? At least this:

1. Programmers routinely, and perhaps necessarily, semantically interpret (in a representational sense—i.e., "semantically-L interpret-L") the terms and other structures in their programs, and the data structures kept in computational memory—taking them to signify, among other things, entities in the program's task domain, as well as mathematical, abstract, and other non-effective entities of diverse sorts.

2. The norms to which the program is accountable almost certainly involve the entities in that task domain and pertinent mathematical entities that those terms and other structures in the programs are taken to signify.

3. Current theories of programming language semantics theorize neither this ubiquitous programmer interpretation-L, nor the norms (implicitly) framed in their terms, but instead use semantical vocabulary for a different purpose—for the program-process (P⟹R) relation described under V2.

4. It remains an open question whether and how we might incorporate this programmer interpretation-L in appropriate theoretical frameworks.

As already indicated, the theoretical framework for 2/3Lisp *was* set up to theorize the semantical process–world (R–W) relationship, as well as the behavioural consequence of its ingredient structures, which in turn were arranged so as to honour such an interpretation. It was also claimed—and I still believe—that doing so is a prerequisite to defining an architecture that can lay any genuine claim to being

reflective, because the form of semantic relationship that a reflective program (and reflective reasoning more generally) must bear to the computation *on which it is reflecting* must be of this representational sort, modeled on its non-reflective process–world (R⇒W) relation, rather than of the program-process (P⇒R) sort discussed under V2.

Needless to say, the 2/3Lisp framework does not address the task of doing justice to the semantic-L interpretation of data structures *in general*. This was the task of 4Lisp, which never materialized—for reasons to be investigated in more detail in chapter 7.

V4 The 2Lisp/3Lisp Framework

As befits its knowledge representation (KR) heritage, the model of computation underlying 2Lisp and 3Lisp was initially based on the formal symbol manipulation (FSM) construal—i.e., on V1. The V4 answer to Q1 is thus straightforward. Computational ingredients were taken to be symbols, with "declarative" representational content taken to consist of deferential intentional directedness towards the task domain, formally manipulated by a language processor. These KR origins are manifest in the Knowledge Representation Hypothesis (KRH) presented in §1.5. As noted, however, not only would I no longer endorse the KRH as stated, but even if I were today writing about 2/3Lisp as those dialects were defined four decades ago, I would not frame the conception on which they were based in exactly such terms. Among other things, the causal involvement of a computational system in its task domain (a constitutive condition of reflection) undermines the two parts of this claim classically taken to be constitutive of formality: (i) that the semantics must be externally attributed; and (ii) that the effective manipulation of the symbols must operate independently of those semantics.[29] Nevertheless, the

---

[29]It is not that I believe that the semantics can itself ever be an effective

use of the KRH as a starting point for the development of 2/3Lisp makes it perfectly evident that the approach to computation being employed (Q1) was indeed, at least as a point of departure, that of the logical/FSM (VI) tradition.

It is because of this overall approach to semantics that the issue of data structures in 2/3Lisp is so fraught, and why the Lisp design study, en route to Mantiq, would only have been completed had 4Lisp been developed as well. Purely for purposes of reflection, at least in what in "Varieties of Self-Reference" (1986) I called its introspective aspects, and in the modest form exemplified by 3Lisp, it sufficed to provide the dialect with the ability to mention data structures (using an analogue of quotation), to designate functions (with normal-form closures), and to represent processing by modeling environments and continuations with functions. And even within that restricted domain, there were limitations. As noted in §2.2, in not even attempting to address such issues, 2/3Lisp fell down on a plethora of important topics relevant even in the introspective realm: general types (beyond those primitively provided), reference to procedural consequence and intension, representation of process and change, among others. Though untenably restrictive, however, with respect to Q1, involving the presumed nature of computation, 2/3Lisp's position was clear: computation consists of activities defined over representational symbols ("vehicles," as philosophers would put it) deferentially directed towards their task domain.

2/3Lisp's approaches to programs (Q2) and behaviour (Q3) were more complex, more controversial, and while I believe they made some stabs in the right direction, ultimately

---

force in causing or leading to behaviour. Rather, the notion of "independence" is in my view too strong a statement of how this non-causal element figures in a computational setting overall.

unsatisfying. These are the issues, I believe, that underlay the gap that opened up between my understanding (V4) and that of Goguen and Meseguer (V2), discussed in §3.3. To understand it, though, we have to consider those two views in conjunction with the third, just sketched: the intuitive understanding of programmers (V3).

Most fundamental is the fact that, in 2/3Lisp (V4) and at least sometimes in programmers' tacit view (V3), *terms* (names, identifiers, "references," etc.) can be used in a way that logicians would call *transparent*. That is: they can be understood as linguistic items that *refer* to, *designate*, or *denote* something, such that the way they are treated (in the processing of the complex expressions of which they are a part) *respects* that reference, but is in no way constrained or expected to *interact causally or effectively* with that to which they refer. Most especially, the appropriate treatment of them in no way involves *producing* that to which they refer—any more than talking about Plotinus, Alpha Centauri, or Blitzen requires us to interact causally with or produce any of those entities. As usual, the simplest computational example is mundane arithmetic: the processing of the term «(+ 2 3)» can lead to the creation or returning of the term «5», for example, where «2», «3», and «5» are understood to be *numerals*—symbols representing or denoting (abstract) *numbers*. The reason that the specific numeral «5» is returned is the obvious one: it is the numeral that denotes the sum of the numbers denoted by «2» and «3». As already indicated, this behaviour runs counter to the specificational (V2) account. It is perfectly explicable as "term rewriting" not only under the 2/3Lisp framework (V4) we are currently examining, but also under the FSM (V1) and tacit programmer (V3) views.

In terms of simple, transparent reference to abstract entities, to generalize, it seems evident that those three views (V1, V3, and V4) that treat reference/denotation transparently, and operation as *orthogonal* but *deferentially accountable* to it, have the edge—in contrast to the official

computer science account (V2).

If reference were all that was at stake, the case I have by now assembled against the computer science view (V2) might be decisive. Matters quickly grow more complex, however—and the 2/3Lisp framework begins to fall down.[30]

The ultimate mandate on programs—to state the obvious, and to go back to the discussion with which chapter 3 opened—is not merely to *refer*, but to *lead to action*, to *engender* (if not actually to specify) *the doing of things*. And not just to engender activity, but to *make changes*—to engender activity that causes "side effects," to have consequences that in turn impinge on subsequent computations, and perhaps on the wider world as well. Needless to say, activity and change are not constitutive ingredients of the (V1) logical framework. Especially these days, logic is often "extended" by treating proof-theoretic relations temporally, which takes a small first step towards incorporating *activity*. Genuine *change*, though, remains beyond logic's compass.[31]

Change is not well treated in the 2/3Lisp framework, either; on this issue both V1 and V4 are weak. Nevertheless, regarding activity *per se*, 2/3Lisp takes an important step beyond logic and V1—and also beyond that embodied in

---

[30]It is also where the merits of the V2 approach begin to come into their own, though it will not be until chapter 7 that we will be able to understand those merits from a genuinely semantical perspective.

[31]To repeat a recurring point: logical frameworks can of course be used to *reason about* change—i.e., in situations where the change is in the semantic domain. That is an entirely different matter. So-called "dynamic logic" (the version that is a modification/extension of modal logic) is a case in point—though I myself would call it *dynamical* logic.

The term 'dynamic logic' is also used in digital electronics for something that genuinely is dynamic (not dynamical)—though in its case, in spite of the statistics of usage, I would challenge its claim to being *logic*.

the analysis of purely functional languages (even though 2/3Lisp might be said to fall within the scope of functional programming languages broadly construed). This step is not only peculiar to these dialects, and critical to 3Lisp's reflective model; it also makes the 2/3Lisp semantical framework sufficiently distinct as to warrant separate inclusion (as V4) in this list.

The fundamental approach to activity in 2/3Lisp was to distinguish **declarative import** and **procedural consequence**[32] somewhat in the way that a distinction between *reference* and *cognitive role* is sometimes advocated in artificial intelligence and cognitive science, where: (i) declarative import, at least roughly, is what terms (including program expressions) refer to or represent—i.e., to be what might be called their "content" in logic, natural language, V1, and perhaps V3; and (ii) procedural (or behavioural) consequence is the activity, or result thereof, that those terms (including programs) engender when they are processed ("run," "executed," "thought," etc.). Procedural consequence is thus approximately what computer science [V2] calls "interpretation-C"—though a careful understanding of the relation of V4 to V2 will take some work to develop.

Declarative import and procedural consequence were theorized in 2/3Lisp under the labels φ and ψ, respectively—whimsically chosen to connote 'philosophy' and 'psychology,' respectively. Thus the 2/3Lisp analysis of the corrected program for "computing factorial" (L1–L4 plus L5', above) went approximately as follows: from a declarative point of view, the expression «(FACTORIAL 5)» was taken to represent (φ) the abstract number one hundred and twenty; and from a procedural ⟦behavioural⟧ point of view, to lead (ψ) to the production of the numeral «120». The production of this procedural consequence developed

---

[32]If I were defining 2/3Lisp today, I would have called it *behavioural consequence.*

according to a standard procedural regimen: processing the original expression—*normalizing* it, in 2/3Lisp parlance, in place of the standard Lisp notion of *evaluation*—in ways that are "effectively indicated" by the structure of the FACTORIAL definition.

What does "effectively indicated" mean?

In traditional logic (and philosophy of language), the expression «(FACTORIAL 5)» would be said to have a *meaning* or *intension*, typically more fine-grained and detailed than the reference.[33] Critical to any computational processing of «(FACTORIAL 5)» would be the link between the term or atom «FACTORIAL» and the "definition"[34] encoded in L1–L5', or anyway the "closure" of that alleged definition (a structure that binds in the values of any free variables, including other function or procedure definitions). What might have seemed a natural strategy for 2/3Lisp, therefore, would have been to extend the notion of "intension" to include all of this detail necessary in order to determine the nature of the processing that would result from processing the expression (the expression «(FACTORIAL 5)», in the present case), under an assumption that *what the expression means*, in contradistinction to what it refers to, *incorporates what it implies as regards processing.*

The problem with that approach, however, is that it would have started to step away from the representational view of the program towards that of V2—that what the program is doing, semantically, is specifying the behaviour that will result—but without going the whole way, and saying that that behaviour is what the program term is *about.* But even more seriously, there is nothing about the

---

[33]This is not the place for an introduction to the philosophy of logic and language; I assume that the vocabulary of intensions (inten*s*ions as opposed to from inten*t*ions) will be familiar.

[34]Whether a procedure declaration should be taken as a definition at all, and if so whether as a definition of declarative import or of procedural consequence, is of course exactly the issue under discussion.

logical/philosophical/linguistic notion of "intension" that involves the kind of semantic ascent that talking about the behavioural consequence of a (formal) term requires. To use Frege's iconic example: "the first star to be seen in the evening" and "the last star to be seen in the morning" may both refer to Venus (i.e., they share Venus—that is, the planet second closest to the sun—as their denotation, reference, or extension), but have different senses or intensions—but there is nothing *formal* or *syntactic* or *linguistic* about those intensions, at least on traditional analyses. Rather, the intensions have to do with relational properties of genuine celestial entities, actual mornings and evenings, etc. Fregean intensions, that is, are neither formal entities, nor solipsist, nor psychological,[35] nor in any way internal— *they involve the full-blooded world as much as do the extensions.*

In contrast, to impose an (uninterpreted) notion of behavioural consequence into one's notion of meaning—especially when behavioural consequence is "wholly defined within the structure of a symbol system," in Newell's memorable phrase (i.e., is mechanically construed)—is to take a fateful step into blanket mechanism—complete with its ties to constructive mathematics, computer science's conception of intuitionism, etc.

In addition, which will matter more and more as computer languages extend from mere programming languages to incorporate knowledge representation systems, of the sort that Mantiq was envisaged to be, and of the sort that play a role in machine learning systems, the fine-grained meaning of a description, term, or other referring expression will in general differ from its procedural consequence. Consider a natural language expression (term) such as "the 3¼ minute piece Stockhausen composed in 1952," or a natural language description of a mathematical object, such as "the

---

[35]Frege was famously anti-psychologistic.

smallest perfect number." The former refers to *Concrète Étude*; the latter, to the number six. Both terms indubitably have intensional meanings—the first involving minutes, durations, and dates; the latter, sums, divisors, etc. But it would be bizarre to associate the meaning of either with *how people process them* (even, in the mathematical case, if a constructive method of "finding" the referent can be deduced[36]).

For these and analogous reasons, in the 2/3Lisp framework I rejected the idea of encapsulating procedural consequence within an intensional notion of meaning. Instead, in describing and analysing the dialects, I theorized *both* procedural consequence and declarative import—separately but in relationship, with the combination claimed to constitute the **full (semantical) significance** (Σ) of the expressions. Had 2/3Lisp been better able to deal extensionally with intensions (i.e., had either the language itself, or even its analysis, been able to *refer to* or *designate* intensions in an adequate way), procedural consequence and declarative import, as first-class entities in their own right, would *both* have been submitted to both intensional and extensional analysis.

None of this is to say that the details of 2/3Lisp's theorization of full significance was especially general or meritorious. As detailed in §8.2, the approach had severe theoretical problems for which I do not yet have a satisfactory

---

[36]What it is to *find* a number is not exactly clear. An intuitionist might say that the number can be "grasped," but on anything like a representational theory of mind, the question is complex. In computing, we assume that to "figure out" a number is to produce a standard numeric representation (binary or decimal, typically). It is obviously insufficient to say that one has found, or "knows," a number merely in virtue of having a term that designates it, since, for six, "the smallest perfect number" does that. Presumably the answer would involve something cognitively analogous to a numeral—a description in terms of powers of ten, etc., or at least from which such represented facts are immediately accessible.

solution, and its treatment of side effects was a complete hack. But it was critical to the framework, and thus critical to understand in order to understand 2/3Lisp's notion of reflection, that it separated the two distinctions, each, in 2/3Lisp's case, rather simplistically taken to be binary: (i) between declarative import and procedural consequence or effect, on the one hand; and (ii) between intension and extension, on the other. Far from being identified—this is what matters—the two distinctions were viewed as *orthogonal*.

As always, moreover—though the dissertation and subsequent papers may not have been as explicit on this point as they might have been—the procedural consequence (ψ) was taken to be deferential to the declarative import (φ), in an effort to honour the founding insights of logic, representation, and intentionality more generally. A severe limitation, however, was that *external* attribution of meaning to identifiers (and data structures in general) was not incorporated into the analytic framework in terms of which 2/3Lisp was analysed and designed—that step was reserved for 4Lisp. Embarrassingly, therefore, the incorrect "definition" of FACTORIAL illustrated in L1–L5, above, would have been taken in 2/3Lisp, as in all other programming languages, to be a *definition*, not, as it should have been, as a *false claim*.[37] In 2/3Lisp's case, therefore, the declarative content function (φ) was completely defined, with no room made for programmers'   (or   other   wider   social   context)

---

[37]To put this another way, the position occupied by the identifier in "function definitions" was assumed to be opaque, not transparent—so that the factorial definition was effectively interpreted as "Take the label FACTORIAL to refer to the function represented by the following λ-term," rather than what it should have been—what at least in this simple case programmers would take it to mean—more along the lines of "take the λ-term (on the right hand side) to denote the FACTORIAL function, but in such a way that it procedurally indicates a manner of effectively computing it."

interpretations—i.e., with no provision for contextual parameterization.[38]

Another substantial step, which took 2/3Lisp (V4) beyond logic and FSM (V1), was a denial of the inverse of the usual definition of formality: that declarative import (designation) be *independent of behaviour*. It follows that the familiar thesis about formality that Fodor championed as the essential property of computation is simply false of 2/3Lisp. Rather, in an inchoate gesture towards a "meaning is use" position, to be examined more in chapter 8, 2/3Lisp allowed for the possibility that what an expression or symbol represented or designated (i.e., its declarative import) might be affected or partially determined by how the system behaved (its behavioural consequence)—by how the system dealt with that expression, by what patterns of activity that expression played a role in. This is one reason why behavioural consequence (ψ) was not only theorized directly (rather than incorporated into something like declarative intension), but more seriously considered to be an *integral part of the overall semantics* (in the general category called "significance"), rather than being relegated to a nonsemantical category like proof or formal processing. While far from achieving anything that Vygotsky, Wittgenstein, or a card-carrying pragmatist would have embraced, 2/3Lisp took a definite step in their direction.

In particular, as described in detail in the papers in *Legacy*, in order to deal appropriately with side-effects, (the theory of) 2/3Lisp takes the full significance (Σ) of an expression to be the only complete or self-contained intentional/semantic story, with both declarative import and procedural consequence (φ and ψ) treated as *aspects* of the

---

[38]Whether the 2/3Lisp framework would have been better understood, in the wider community, had I succeeded in developing 4Lisp is an unanswerable question.

full significance, interdefined in terms of it. It was only with respect to significance (Σ), not denotation (φ) or procedural (behavioural) consequence (ψ) that 2/3Lisp was even approximately compositional.

Crucially, however, in spite of this overall approach, the framework maintained commitment to the overarching mandate that behaviour be normatively governed by (be deferential to) representational content. This overarching deference—reliant on the fact that, like logic, 2/3Lisp retains two aspects for each expression or symbol or representational structure—is demonstrated in theorems proved for 2/3Lisp, reminiscent of logic's soundness and completeness proofs. In particular, as proved in the dissertation, there were two specific mandates to which procedural consequence (i.e., the result of processing) was held in both 2Lisp and 3Lisp: first, that it be *designation preserving*:

$$\forall\alpha\ [\varphi(\psi(\alpha)) = \varphi(\alpha)] \qquad\qquad \text{E1}$$

and second, that it produce ("return") a normal form designator of that which its argument designates:

$$\forall\alpha\ [\text{NORMAL-FORM}(\psi(\alpha))]$$

To the extent that the terms apply—using Etchemendy's "want what you get" and "get what you want" epithets, cited above—it is the former theorem that is the analog of soundness; the latter, of completeness.

In addition, in order to accommodate reflection, both φ and ψ were made "available" in the language, under the labels «DOWN» (abbreviated «↓») and «NORMALIZE», respectively, leading to the following provable properties of the system:

$$\forall\alpha\ [\varphi(\ulcorner\downarrow\alpha\urcorner) = \varphi(\varphi(\alpha))]$$

and

$$\forall\alpha\ [\varphi(\ulcorner(\text{NORMALIZE}\,\alpha)\urcorner) = \varphi(\psi(\alpha))]$$

More details are provided in the *Legacy* papers. The present point is merely that because both aspects are theorized, the mutual constraint that I take to be fundamental to intentionality (the constraint that logic captures in its norms of soundness and completeness) has a grip on the 2/3Lisp framework in a way that is not available to the theoretical computer science view (V2).

None of this is to say that the 2/3Lisp framework was without problems. One particularly evident one was that everything was assumed to have a referent (i.e., all expressions were taken to be *terms*), and everything had procedural import—a generalization that is ultimately a somewhat blinkered way of treating the full range of semiotic contexts that might be computationally useful. In particular, no provision was made for *commands* in 2/3Lisp, intuitively meaning "do this" or "change that." Nevertheless, I felt that it was important to include at least such "messy" constructs as side effects, in order to demonstrate that they, too, could be included within a normalizing, term-rewriting, deferential approach (and could be incorporated without difficulty in a reflective version as well). But in retrospect one might charitably say that including such constructions was something of a force-fit.

For example, the 2/3Lisp expression

```
(REPLACN 3 '[A B C D E] 'X)
```

was intuitively understood (i.e., in something like the V3 interpretation of 2/3Lisp) to mean "replace the 3rd element of the rail «[A B C D E]» with the atom «X»" (rails in 2/3Lisp are normal form designators of sequences[39]). More literally—and pedantically—the mandate is that that, in terms of procedural consequence, the general form «(REPLACN *N ARG*

---

[39]In usual computer science parlance, which lacks the habit of distinguishing canonical representations of simple mathematical entities from the entities thereby represented, 2/3Lisp's rails would likely have been called 'sequences' or 'vectors.'

NEW)» should be understood as follows: assuming that the first argument («*N*») denotes a number (not numeral) *N*, replace the *N*th element of the rail denoted by the second argument («*ARG*») with the item designated by the third argument («*NEW*»)"—in a way exactly analogous, semantically, to the referential structure of an English sentence of the form: "please replace the third item on the shelf with the book you are holding."

Functionally, this approach worked well enough. The semantical infelicity derived from two facts: (i) like all expressions in 2/3Lisp, the expression as a whole (not just its parts) was required to designate something (i.e., was required to have declarative content as well as procedural import); and (ii) the result of its processing was mandated, by the overarching term-rewriting norm, to return a normal-form designator of that designated item. It was natural, therefore, to stipulate that the whole expression should designate the *result* of performing the action, rather than designate the action itself—i.e., should designate the rail «[A B X D E]» (the *same* rail as was its original argument, but with a new third element *changed*), so that the expression could return ("normalize to") the normal-form designator of that rail—namely, the handle «'[A B X D E]».[40]

This is problematic because this interpretation of S1 as being a term (i.e., as having a declarative import) parts company with the programmer's natural (V3) understanding of it as fundamentally imperative. In this case, I believe that the V2 interpretation of this program fragment as *designating the change* is closer to correct, or anyway more appropriate. I was aware of the problem at the time 2/3Lisp was designed: the difficulty was that, were it to have been taken to designate the *action*, the overall mandate that 2/3Lisp processing return a normal-form designator of

---

[40]Handles, notated with single prefixed quotation marks, are the 2/3Lisp normal form designators of all internal structures.

what its argument designated would have required the return of a *normal-form designator of the action itself*—something that, if not conceptually ill-formed, was at least beyond 2/3Lisp's descriptive prowess.[41,42]

Two things can be concluded.

First, 2/3Lisp took a number of steps in a direction in which I believe we still need to travel further. I deem it as progress that 2/3Lisp:

1. Dealt with declarative import (with what something refers to or describes, with what a traditional FSM adherent would refer to as its *content*) and procedural consequence (how that something is used, what effects are engendered by processing it) in a relatively flexible and interrelated way, with both aspects considered to be part of an expression's overall semantical significance;

2. Avoided the conflation of procedural import with intension or meaning; and

3. Recognized that it is possible in some cases to allow the declarative import of a particular expression or fragment to depend on procedural import, even while the overall weight of that dependence leans in the other direction, with procedural import honouring declarative import "in the large," in

---

[41]Perhaps I could have introduced something like a computational equivalent of the deictic "It's done!"—not unlike network "ACKs"—but I was not quite up to such inventions at the time.

[42]There was a way of requiring that the designation of a 2/3Lisp term be *produced*. The expression ↓*exp* designates *the designation of what *exp* designates*; procedurally, processing «↓*exp*» requires producing a normal-form designator of *exp*'s referent; if *exp* designates a structure already in normal-form—call it α—then ↓*exp* will produce α, returning it as a result. But this, too, is indirect, and while locally consistent and possessed of its own local elegance, it does not deal with the main issue discussed in the subsequent paragraph of the text.

recognition of the deferential attitude that I take to be fundamental to intentionality.

On the other hand, not only was the 2/3Lisp framework inadequate for dealing with any of these issues in full generality, but the dialects were insufficiently subtle to allow even those aspects that were recognized in their theoretical analysis to be fully treated *within the language*. In this way, 3Lisp fails to be adequately reflective even in its own terms.

In particular, and as already suggested, while a simple (i.e., transparent) use of a term α designated—tautologically, as it were—α's declarative content, there is no way, in 2/3Lisp, to construct an expression β that would designate the *activity* engendered by processing another term α. The procedural *extension* of α could be designated, so long as processing α was side-effect free, through the use of a function (called «UP», abbreviated «↑»)—a processing-aware version of semantic ascent—that mapped entities onto their normal-form designators. That is: «↑α» designated the normal-form designator of (the designation of) «α», implying that «↑α» designated what «α» normalises to, so that «↑(+ 2 3)», for example, denotes the *numeral* «5»—and thus returns a handle (a normal-form internal structure designator, written with a preceding single quote mark):[43]

$$↑ (+ 2 3) \Longrightarrow \text{'}5$$

But any wider procedural consequences, including side-effects, were outside the realm to which 2/3Lisp could provide representational access.

In part, this limitation was a consequence of the fact that, as in computer science more generally, *actions* were

---

[43]↑ was declaratively transparent but procedurally opaque, since normal-form designators are not necessarily unique. ↑α designated the actual normal-form structure that α itself (not the referent of α) normalised into.

not taken to be elements of the semantic domain over which 2/3Lisp structures designations ranged. It would have been a reasonable requirement on a better version of the language—perhaps on 4Lisp, certainly on Mantiq—to have remedied this lack. In a sense, it would not have been a radical departure; the declarative content and procedural import were theorized in the semantical analysis of 2/3Lisp, which analysis was formulated in turn in a language based on the λ-calculus, and so terms for both aspects of an expression's significance were close at hand.

To designate the procedural import of an expression, however, is not yet to be a *command*. I claim no special insight on what it is to be a command, but perhaps I could say just this: whereas, in the referential model, the procedural consequence is, in a sense, derivative on the declarative import (content), for commands the ontological priority is reversed: commands, in the first instance, are *about* what they *do*, cause, or effectively lead to (or in the more general case, about what they enjoin their receivers to do). At a minimum, that asymmetry would want to be rendered manifest in a language with a wide range of semantical types.

2/3Lisp had other limitations. As I have already said, no facility was provided for constructing a term that designated either the procedural or declarative *intension* of that or any other term. Types, though fundamental to the languages' coherence, were stipulatively denoted with specially-designated formal constants (basically a cheat). And of course, the dialects' being procedural overall, there was no ability to make simple statements—in the sense of *affirmations*—of any sort. These and a myriad other limitations were to have been addressed, as appropriate, in 4Lisp and Mantiq.

The second conclusion we can draw about V4 has to do with a more serious looming challenge, connected to the issue of action and change—related, too, to some issues

considered in chapter 7, and tied as well to the difficulties facing 4Lisp. It has to do with the most important ways in which programmers' tacit (V3) understanding of programs outstrips all of V1, V2, and V4.

The issue can be illustrated in a simple example. Suppose a program contains an instruction that means, in a rough English translation, *increment the rank of employee$_{27}$*. In 2/3Lisp, one might imagine that this would be written:

(INCREMENT (RANK EMPLOYEE-27))

What is dicey about this statement is that, according to 2/3Lisp semantics—or anyway in the style of 2/3Lisp semantics, extended in ways imagined for 4Lisp—EMPLOYEE-27 might be taken to refer to an actual employee (a living breathing human), and RANK to their actual rank, etc., which would imply that the effect of processing the instruction would be, or at any rate might be, to *promote* them. What a programmer understands full well, of course, is that, the act of promoting the employee is distinct from the engendered computational event of *updating the computational record of that employee's rank* (if related to it, with the latter perhaps honouring or recording or deferring to the former).[44] In this ordinary understanding, that is, the expression (RANK EMPLOYEE-27) is somehow—perhaps flexibly or contextually or simultaneously—taken, by the programmer, using 'signify' for the moment as a general semantical term, (i) to signify the actual rank of the person, *and* (ii) to signify (or perhaps even to *be*) their employee record. As regards the latter interpretation, it seems as if the V2 semantical view is somehow more appropriate: terms such as EMPLOYEE-27 are in some cases taken to denote or represent records in computer memory, not real-world entities in task domains.

---

[44] Programmers of course also understand when the action *is* meant to happen, for example, when debiting an online bank account balance.

On the other hand—and this is why the whole subject matter begins to outstrip the capacities of any currently known semantical framework—the intelligibility of the program, and the normative constraints to which the program should be held, depend on the fact that that record, *in turn*, represents a real-world employee and their rank. That is—or so anyway I believe, though it would take ethnographic work to demonstrate that this is in fact the case—*the programmer must be simultaneously aware of all three types of entity: the program identifiers, the computational records, and the people and ranks represented by those records.* The problem with present-day theory is that existing semantical accounts (including that of 2/3Lisp) deal only with a single level of designation or representation: from sign to signified—with possible inclusions of intermediate intensional levels of *meaning, character,* etc. But all of such familiar multiple-stage semantical relations of expression, meaning, character, intension, and reference are inappropriate approaches for dealing with this triple form of signification; nor would so-called "two-dimensional semantic frameworks" be of use in this context. The computational record—the intermediate entity in the present three-stage layering of signification—is neither the *meaning* nor the *character* of the identifier.

In the next chapter, I will argue that the situation is in fact even more complex than three or even four stages of signification. Even recognizing just 3 stages, however, allows for a useful comparison between the computer science (V2) approach and an amalgam of the 2/3Lisp-cum-FSM (V1 and V4) approaches. The computer science approach (V2) deals with the first stage of designation—from identifier to (abstract or concrete) memory record—but ignores the second stage, from record to person. Programmers, by using natural language names, craft the program so as to enable them to understand both stages—second as well as first—even if the significance of the second stage memory record is invisible in accepted theoretical

accounts. The FSM (V1) model has no intermediate level of (changeable) records, and so is not up to dealing with the tri-stage situation at all.

The 2/3Lisp approach (V4) is reminiscent of the FSM one, but makes extensive use of quotation ("handles," in 2/3Lisp parlance) enabling the program to refer to data structures explicitly. Moreover, using the «↑» operator noted above, something like the following expression might be used in 2/3Lisp,[45] in order to obtain referential access to the data structure in question:

$$\text{(increment } \uparrow\text{(rank employee-27))} \qquad \text{[E4]}$$

Marking every level of semantic ascent in this way has the advantage of permitting unambiguous reference (and access) to each of any number of stages on a multi-stage stack of representational relations. Also in its favor is the fact that it makes it evident that what is being incremented is at least at some level of semantic remove from the employee's actual rank, and from the employee themself. And of course it makes evident the fact that 2/3Lisp programs are not prohibited from being about (i.e., being at one level of semantic remove from) data structures, as is commonplace in reflection.

However—and not incidentally—the 2/3Lisp strategy also has the ultimately fatal disadvantage of forcing the programmer to be relentlessly explicit, always and everywhere, about what level of a multi-stage semantical stack of this sort they have in mind. A decisive advantage of programmers' tacit understanding (V3) is that programmers can use context and commonsense to determine which level in the referential stack is being dealt with—a plastic and

---

[45]Needless to say, whether this would work would depend on exactly how RANK behaves; (RANK EMPLOYEE-27) would have to normalize to something other than a simple numeral, in particular, since numerals can presumably not be incremented in such a way that a subsequent normalization of (RANK EMPLOYEE-27) would normalize to the incremented version.

fluid contextual skill, as already mentioned, far beyond the capabilities of any extant semantical framework.

This is a moral we are going to encounter with increasing frequency. Current semantical frameworks in computer science (V2) are blind to the genuine semantical facts that govern contemporary programming systems. This is unfortunate, since these are facts that programmers must tacitly understand, in order to construct systems that are useful and correct. In that sense, theory has failed us. On the other hand, the semantical situation regarding complex computer systems dwarfs in complexity anything that can be addressed in any current theoretical frameworks, in computer science or philosophy or linguistics or cognitive science more generally—issues involving perspectival notions of identity, contextual (and multiple) levels of cascaded significance, and a variety of other concerns.

5 Comparison

Which of V1-V4 would I choose today? Not one. I would not even select a theoretical reconstruction of contemporary programmers' tacit understanding (V3), in spite of its considerable merits, and notwithstanding its theoretical challenges. Nor do I believe—though this remains to be argued—that an adequate approach can be constructed, piecemeal, from V1–V4's constituents. Doing justice even to computation, let alone to reflection, will require a more complete overhaul of our semantical frameworks than that.

To make progress in that direction—not even starting to frame a full-fledged successor, but starting to appreciate what that would require—we need to develop a workable perspective from which to survey all the various existing suggestions.

1 Procedural vs. Declarative

A first suggestion is to divide the four views' approaches to semantics along an informal but familiar dimension, often characterized as follows:[1]

1. PROCEDURAL: Approaches that put primary emphasis on *dynamic results*, *behaviour*, the *procedural consequences* of a symbol, expression, or program fragment being processed; and

2. DECLARATIVE: Approaches that place primary focus on

---

[1]Note that I do not agree with these characterizations, nor with the implicit assumption that they are either exclusive or exhaustive. The attempt here is simply to articulate the sense that underlies informal usage.

reference, *designation* and *truth*—i.e., on what, in a given context, a symbol or expression is *about*, what it *names* or *points to* in the world, what it *represents*.[2]

Under this distinction, we could characterize the views as follows:

α)   VI (FSM and logic) would fall into the DECLARATIVE camp. At least as classically analysed, those traditions take semantics[3] to be first and foremost representational or declarative: focusing on the reference, designation, and/or truth. At least paradigmatically, which justifies the 'declarative' label, expressions in such declarative formalisms are taken to represent *what is the case*; issues of how to reason with or use such expressions—what to do with them, procedurally—are held at bay (specified elsewhere, figured out by a reasoning system, etc.).

Four clarifying points. First, as noted above, the (semantic) representation relation in such systems is something that *obtains*—it is not a process, not something that takes time. If Alex is Pat's parent, then «(PARENT ALEX PAT)»[4] is simply *true*; it is the case; it does not require the passage of any time for it to become true (after an utterance or use of that expression, for example). The 3Lisp atom «MT-SAINT-ELIAS» can represent the

---

[2]Two clarificatory points. (i) It is contentious whether being true or false (the primary semantic property of sentences or claims, on this view) should be viewed as analogous to, or even (as Frege suggested) special cases of, naming; and (ii) truth is not dealt with in a satisfying way in 2/3Lisp, in spite of the dialects containing primitive normal-form designators for truth and falsity ($T and $F).

[3]When I say "they take semantics to be α" I mean that most classical analyses of these approach tend to view semantics as α—not that it is impossible or unknown to treat semantics differently in their analysis. So-called "proof-theoretic semantics" approaches to logic, for example, do not fit neatly into what is said in this note.

[4]Or «Parent(Alex, Pat)», as it would be notated in logic.

second highest mountain in North America without that reference involving any activity; the arrow of intentional directedness may span both space and time, but it does not take time to do so.

Second, declarative systems are paradigmatically (though not necessarily) what I will call **statical**:[5] what represents—what the semantics is defined over—are typically *states* of the system, not processes or activities. Again, this is not necessary. If, in response to a child's question about a spiral stairway, you respond by moving your hand upwards in a helical gesture, the (shape of the) staircase would be represented by your hand's dynamic motion, but the representation would still count as declarative.

Third, a state's being representational, even in the statical case, does not imply that that which is represented need be a state. A statical calculus expression can readily represent dynamic motion (as for example in a differential equation in physics).

Fourth, what matters most, from the VI perspective, is that the states over which semantics is defined (symbols, expressions, formulae), and the meanings or interpretations defined on them, are generally assumed to be ontologically and explanatorily prior to process and behaviour—prior to and independent of what happens to the states over which they are defined. That is: any dynamics (process, behaviour, results) of such expressions' treatment—the ways in which they are manipulated, and the consequences thereof—are not typically viewed as being constitutive of their semantic value. Rather, as I have emphasized throughout, in these traditions the relation between semantics and behaviour is usually taken not only to be fundamentally normative but also

---

[5]*Statical* because defined on *states*—not *static*, in the sense of not itself changing over time.

to be "staged." Process and activity is held accountable to whether it honours the (statical[6]) semantic interpretation of the states and expressions over which it is defined, where that semantic interpretation is assumed to be established in advance.[7]

This approach, laid out more carefully in chapter 6, is familiar from logical syllogisms and standard examples of elementary reasoning. In a derivation such as the classic:

$$
\vdash \quad \frac{\forall X (MAN(X) \supset MORTAL(X))}{MAN(SOCRATES)}
$$
$$
\overline{\hspace{1em} MORTAL(SOCRATES) \hspace{1em}}
$$

it is assumed that the semantic facts—that «MAN» represents the property of being a man, «MORTAL» the property of being mortal, and «SOCRATES» the Athenian philosopher—are established or stipulated prior to, and independent of, the derivation.[8]

β) V2 (computer science as theorized), in contrast, would seem to count as PROCEDURAL, viewing semantics as having primarily to do with activity, process,

---

[6]Cf. footnote 5.

[7]*Inference*, we say, should be *truth-preserving*, where *truth* is a semantico-normative condition defined on the static sentences; its *preservation*, the derivative dynamical norm defined in terms of it.

[8]Strictly speaking, the argument would be held to be *valid* independent of the interpretation, and so «man» and «mortal» and «Socrates» could theoretically, or so it is usually assumed, be mapped onto any properties at all; all that is guaranteed by the structure of logic is that, *if the premises are true, then the conclusion will follow* (be true as well). But though rarely acknowledged, background conventions govern any such potential assignment so as to enable the validity of such analyses to obtain. (If «MORTAL» were mapped onto the property "the clause in which this property has been predicated is part of a premise of an argument, but not a conclusion," and «MAN» onto "is a premise in an argument," then the argument would no longer be valid.)

behaviour. "What happens" is taken to *be* the symbols'
and expressions' semantic value or importance, rather
than the activity and behaviour merely being *account-able* to that value.

It is this fact that semantic content and causal conse-
quence are taken to be the same thing—a single prop-
erty of symbols or programs—rather than being a rela-
tion between two at least partially orthogonal aspects,
that makes v2's approach so philosophically peculiar
(and that explains, among other things, how opera-
tional and denotational semantical accounts are pre-
sumed to coincide, why the idea of requiring proofs of
soundness and completeness are category errors within
the computer science context, etc.). How a symbol is
processed is simply "how a symbol is processed"—how-
ever the underlying architecture works—rather than
being subject to imposed normative constraint.[9]

As this description makes clear, theoretical computer science
(v2) and FSM/logic (v1) use the term 'semantics' in approxi-
mately orthogonal ways. What is considered semantic in CS
(v2)—i.e., behaviour—is considered *non*-semantic in logic
(v1), although that behaviour is *accountable* to semantics.
What is considered semantic in logic (v1)—paradigmatically,

---

[9]An *implementation* may be held normatively accountable to the seman-
tics—the semantics specifying how the implementation *should* behave,
the implementation actually behaving in some way. Such an approach
would allow one to ask whether the implementation was *correct*. But that
is a separate issue; it would merely be to have the singular content-as-
consequence articulated in the semantic account play the role of a *require-
ment* on the implementation. Requirements (of this or any other form),
however, are not considered to fall within the scope of the *semantics* of
the system they apply to. Moreover, there is nothing special to computa-
tion in being subject to requirements; the same is true of any engineered
system. (One could, of course, state in a symbol system's requirements
that its task domain be *correctly represented*—but that merely establishes
a second normative relation.)

relations between the symbols and the task domain—is not treated as semantic in CS (V2), not because it is treated as being something else, but more profoundly because it is not considered to be relevant to semantical discussion at all.

γ) V3 (programmer's tacit intuitions) has a richer conception of semantics than either V1 (FSM) or V2 (computer science)—suggesting one way in which a tenable semantics incorporating both of their insights (of which V4 is an example), will be non-standard in form. Like V2, V3 (usually) places behavioural issues in the driver's seat, but unlike V2 it is undergirded by something like a representational dimension—manifested in the choice of names for variables and procedures, and more generally in the presumptive representational significance of data structures.[10] Though this tacitly recognized representational character is not officially theorized, virtually every programmer, or so anyway I have maintained throughout, would agree that this representational dimension is fundamental to software's design, comprehensibility, maintenance, and use.

δ) V4 (2/3Lisp) also has a richer conception of semantics than either FSM (V1) or theoretical computer science (V2). In that respect it goes some way towards capturing programmer's tacit intuitions (V3). On the other hand, whereas V3 takes behaviour to be primary, 2/3Lisp (V4) sides with FSM and logic (V1) in giving (what I am calling) declarative issues pride of place,

---

[10]It is not just a question of the names being recognized as having meaning by programmers. In all likelihood, the successful functioning of the program will depend on the data structures being set up in such a way as to "track" or otherwise represent salient entities, events, phenomena, etc., in the program's task domain, to say nothing of meshing appropriately with sensors and effectors that connect the system to the task domain (being correctly updated, leading to intended actions, etc.).

both ontologically and explanatorily. Yet in spite of being affiliated with FSM and logic (V1), 2/3Lisp (V4) is unlike both of them in two critical ways—both of which draw it closer to V3.

First, 2/3Lisp (V4) does not take the declarative (referential or designational) aspect of expressions to be *independent* of the procedural; rather, it is founded only on the view that, *overall*, the declarative facts transcend, and via that transcendence normatively govern, the behavioural facts. It is this overarching commitment that maintains V4 as fundamentally deferential, while at the same time making room for the fact that, in particular cases, declarative content and procedural consequence may be constitutively interdefined.

Second, 2/3Lisp (V4) treats *both* declarative and procedural dimensions of an expression as "genuinely semantic"—theorizing them as partial, interrelated constituents of a single, overall encompassing intentional significance function (Σ). I believe that this integrated view of representation and behaviour as dual, integrated, equal-weight dimensions of the semantic is 2Lisp's most distinctive, most important, and least understood characteristic.[11] As well as attempting to merge some of the powers of both V1 and V2, it has the consequence of moving the normative governance of the declarative/representational *inside* the semantic framework (in FSM/logic the norms are taken to be defined in terms of the semantics, but nevertheless to be external to them, in the sense that that which they govern—the activity—does not affect or contribute to anything's semantic value). In 2/3Lisp it is the integrated significance function as a whole (Σ) that must—and must be shown

---

[11]Its closest intellectual affinities are to be found in philosophy of mind—particularly in the area of what is called "conceptual role semantics" [CRS].

to—satisfy overarching norms reflecting the system's overall deferential attitude to the world.

2 Specifying Behaviour vs. Behavioural Specification

The foregoing summary (call it **S1**) is not empty, but alas it is untenably glib. Why so? Because we have not yet turned up the intensity of the analysis enough to force the various views to break open and release their innermost secrets. Not only does it ignore some fundamental issues, a few of which I have already touched upon; even among those it does it address, it glosses over a welter of conflations and confusions. The problems are especially serious regarding V2: the theoretical approach that underlies present-day computer science.

2a Data structures

In chapter 3, I noted that contemporary programming language semantics does not treat the *ingredients* of computational processes (the states of the computation, including the states of all data structures) as semantic at all. Neither data structures, nor states, nor memory, nor behaviour, nor input, nor output, nor anything else affected by or resulting from program execution, is considered, according to V2, to be part of the domain of semantical analysis—not considered for procedural semantical analysis, or declarative, or behavioural, or representational, *at all*.[12] These entities are all taken to be in the *range*, not in the *domain*, of the semantic interpretation function.

So at least I claimed. But note that there is something

---

[12]It is this blindness, on the part of received theory, that explains the huge gap, noted earlier, between the *semantics of programming languages* and the *semantics of programs*. I will consider it to be a requirement on a successor account that it bridge this gap: that any account it provides for the semantics of a language include, perhaps among other things, a framework for accounting for the semantics of programs written in it (including for data structures), and for the processes that result from their execution.

puzzling about this statement, at odds with the glib summary. Consider data structures (though similar considerations would apply to input, memory, etc.). It might have been reasonable to expect that data structures would not be semantically interpreted in V2—if, for example, we wanted (like Newell) to keep the analysis "wholly defined within the structure of a symbol system." But, peculiarly, this expectation only makes sense on a *declarative* understanding of semantics. That is, one might imagine that a plausible rationale for ignoring the semantics of data structures might run as follows:

> Sometimes—perhaps even usually—data structures and the like will designate (represent, etc.) entities, phenomena, activities, states of affairs, etc., in the world or task domain that lies outside, beyond the limits of, the computational process itself. Suppose, at least for the sake of this argument,[13] that for methodological purposes, like Newell, we impose a requirement that the scope of theoretical inquiry remain "within" the computational process itself. Doing so will enable us to guarantee that the resulting analysis will be completely internal, formal, closed, etc. (in something of a computation-theoretic analogue of Fodor's "methodological solipsism" approach to cognitive science[14]). To satisfy this criterion, we must bracket the semantic content of data structures—leaving them out of the purview of the analysis of the programming language's semantics.

The rationale makes sense, however, *only from within a declarative conception of semantics*—i.e., from a point of view that takes the (potentially ignored) semantics of these data structures to have to do with their process-to-world *representational* content.

---

[13]With which, needless to say, I disagree.

[14]Needless to say, this is a methodological constraint with which I profoundly disagree—but even if pernicious it is *intelligible*, and I take it that some such commitment must underlie the traditional CS view.

And, being declarative in orientation, the rationale thus runs afoul of what we concluded in the analysis of V2, and repeated in the glib summary: that computer science takes "semantics" to have to do with *behavioural consequence* in an internal sense—i.e., with how things behave, inside and at the edges of the machine, with how they are (mechanically) treated. And, contra the rationale, data structures manifestly *do* have procedural consequence—do play a role in affecting behaviour that, if construed mechanically, falls within the confines of the computational process. So if semantics is taken to have to do with (effective, mechanical) behaviour, not with deferential relations to distal subject matters, why is the profile of behaviour associated with data structures not theorized as *their (procedural) semantics?* That is: if one's approach to semantics is fundamentally procedural, then why is a rationale for ignoring the semantics of data structures even needed? Surely a card-carrying proceduralist should embrace the task of providing a semantical analysis for data structures as well as programs—viz., a procedural semantics—rather than leaving them outside the scope of semantical analysis entirely?

Is this what a procedural (V2) semantics for object-oriented programming languages comes to? The objects, on such a suggestion, could perhaps then be mapped by a resolutely CS (V2) conception of semantics onto the behaviours that result from their methods being called.

This suggestion has legs. It merits investigation.

2b Declarative specification of procedural behaviour

A second hint that something is awry with the glib summary has to with programs themselves. What are we to make of the fact that computer scientists view what they call a *denotational* approach to semantics to be the strictest, most rigorous approach? Sure one might have expected the term 'denotation' to be associated with (and thus to preferred by those who embrace) the declarative/referential/truth-value tradition? The use of mathematical techniques fails to explain the choice of

terms. Why is the mathematical modeling of the behaviour that results from running programs called *denotational*—rather than *behavioural* or *procedural?*

The issue is more than terminological. If programs are treated as the "procedural" form of expression *par excellence*, then one might expect them to be a "locus classicus" for a behavioural conception, wherein the meaning of the program would be something like *how the program will be used.* But the awkwardness of that wording is telling. Programs themselves (programs *qua programs*), or at any rate the expressions within them, are not exactly *used*—certainly not in the logical sense of 'use.' Surely what is "used," in standard programs, are data structures! "What the program will *do*" would be the more natural phrase, when speaking about programs.

If one thinks about it, in fact, it is clear that, if one is forced to make the distinction, programs are both understood and theorized: (i) *not* as indicating how *they themselves* will be used (treated, manipulated, processed, "interpreted" in the computer-scientist's sense, etc.); but rather (ii) as specifying how the *data structures* that they "deal with" will be used—constructed, deleted, manipulated, rearranged, etc. And 'deal with,' in this phrasing, has an intentional, "directed towards" meaning.

The point is that, on the classical computer science view, programs, at least theoretically, seem, contra to what I have said before, to specify, at least somewhat *declaratively*, the behaviour in terms of which they are semantically analysed. "What a program means," that is, is taken to be "what behaviour the program effectively *specifies*," where the word 'specifies' is interpreted in a manner not all that different from how it might be used in English—e.g., in specifying the procedure to follow when applying for a visa, or specifying the equipment one should take on a canoe trip. That is, like a calculus expression signifying activity in physics, there is at least some sense to be found in the idea that the activity germane to programs may be in the semantic content of the program's terms, rather than as properties of those terms themselves.

Contra the glib summary, that is, rather than saying that programs have *behavioural semantics*, it seems as if it might make more sense to say that programs *declaratively*, but nevertheless *effectively*, *specify* behaviour—or, perhaps more insightfully, *effectively but declaratively specify effective behaviour*. The two conditions—*being procedurally meaningful*, on the one hand, and *declaratively representing something procedural*, on the other—are not the same. Saying "You are a despicable fool!" may have behavioral consequences, but at least in the first instance the statement is not *about* behaviour.[15] Conversely, such phrases as "the transition of an electron from a 2s to a 2p orbital" or "adding milk to a roux"—or for that matter, the mathematical terms in the equation:

$$x(t) = x_0 + v_0 t + \tfrac{1}{2}\, at^2$$

are about temporal behaviour, but in ordinary contexts[16] would be expected to function as unexceptional *declarative* expressions. They are perfectly ordinary referring terms, which happen to *refer* to activity and process.

So should we say that programs are effective, declarative specifications of behaviour that takes place at the level of data structures, on a much more traditional conception of semantics than the analysis in chapter 3 would lead one to expect? The suggestion fits with the comment made earlier (and depicted in the specification view of Fig. 1): that programs are often taken to be at one level of semantic remove from the activity or behaviour that their execution engenders. Not incidentally, the suggestion also fits with my choice, 45 years ago, to call the traditional computer science view 'specificational.' It squares, too, with the widespread sense that, at least in general (and especially in the case of statically typed languages), it is possible to

---

[15]Even if it was the addressee's behaviour that led the speaker to come to hold, and utter, the judgment.

[16]E.g., when used as referring terms in referentially transparent contexts.

"determine the meaning" of a program *statically*—as for example during processes of compilation, and when a programmer "figures out what the code means" by reading it. Both facts suggest that the dynamic execution context is not needed in order to determine, in at least some pre-theoretic sense, what programs *mean*.

Which view—*themselves being procedurally consequential*, or *being a declaratively (if nevertheless effectively) meaningful way of specifying (effective) behaviour*—is closer to the one that computer science attributes to programs? The glib summary endorsed the first; these last considerations suggest the second. Perhaps the distinction is otiose; perhaps neither construal is sufficient on its own. Perhaps the framing conflates two senses of "behave": one being the form of behaviour that takes place in the process to which the program gives rise, where objects and data structures are the "objects" or "patients" of the behaviour; the other being the fact, mentioned but as yet unanalysed, that programs have to specify that behaviour *effectively*, in the sense of being causally sufficient configurations so as to produce that (effective) behaviour. Does that means that the computer science view (V2) is intended to be an admixture of both? And if an admixture, how do the two behaviours relate? Is the answer that programs are supposed to combine both aspects at once— to be a *behaviourally effective way of specifying effective behaviour*? Perhaps, to make it more explicit yet, programs should be understood as effective in a double sense, and as implicating two distinct forms of declarative semantics as well, along something like the following lines:

S2) *Programs are an effective (as well as declarative) way of specifying effective behaviour conducted over declaratively meaningful data structures (plus inputs and outputs).*

Pedantry looms. But no matter how obscure, there is insight in suggestion s2. Buried within it are hints about the answers to four issues that got us here: (i) why computer science calls the most rigorous form of programming language semantics 'denotational'; (ii) the idea that programs exist at one level of

semantic remove from data structures (which very statement, it should be noted, is framed in terms of a declarative notion of semantics, not a procedural one); (iii) why processes that take programs and produce the behaviour that they specify are called *interpreters*; and (iv) why, as addressed in the rationale, the declarative semantics of data structures is debarred from computer-science internal theoretical analysis.

3 Data Structures Revisited

Look at the situation in yet a different way. Suppose we were *not* to take programs to be procedural in the way claimed in the glib summary, but were rather, as suggested in the last paragraphs, to view them as (effective in some as yet to be explicated sense, but nevertheless roughly declarative) *specifications* of effective procedures or behaviour. And second, and contrary to what has been said so far, suppose we were to claim that data structures (not programs) *are* being assigned semantics after all—*procedural* semantics, at that—but in a roundabout way. Could we not say, in particular, that, for whatever reason, computer science has settled on the following practice:

> **S3**) Rather than (i) analysing the (procedural) semantics of data structures *directly*, by having the data structures figure as the explicit domain of a (procedural) semantic interpretation function, which explicates the behavioural or causal consequences they would play a role in engendering in the course of a process's temporal evolution, theoretical computer science has instead adopted the custom of (ii) assigning procedural semantics to data structures *indirectly*, via the (denotational) semantics of the programs that (effectively) specify how those data structures are to be (effectively) manipulated?

That is, to caricature a bit: the suggestion is to understand the *declarative* semantics of a program to be the *procedural semantics* of the data structures over which that program specifies behaviour.

Suggestion **S3** is convoluted, but has merit. It is *programs*, after all, that are thought to determine how data structures "are used"[17]—so the suggestion seems appropriate in allocating *responsibility* for that behaviour to programs, while acknowledging that what the behaviour *concerns* is the data structures. It also renders intelligible the fact that the range of program behaviour is restricted, in these traditional analyses, to the "interior" of computational processes—since this is procedural semantics we are talking about—and the interior of the process, along with the input and output activities that cross its border, are what falls within effective (mechanical) range. Ironically, third, it makes sense of something I have mentioned several times: why we say that programs are *interpreted*, and of why program reference to data structures is taken to be genuine *reference*. (I call this point ironic because it requires a *declarative* notion of semantics to understand the sentence's use of the terms 'interpret' and 'reference'.) Fourth, if we take programs to be effective, declarative specifications of effective behaviour to be conducted over data structures, then it at least seems as if much of our familiar logical, philosophical, and natural-language based conceptions of *naming*, *reference*, etc., might be able to be retained.

Moreover, suggestion **S3** raises a theoretically intriguing possibility. If we wanted to convert the program's indirect specification (of the procedural semantics of its data structural objects) to a direct one—if, that is, we wanted to know the procedural semantics of some particular data structure or structural type—we could do something analogous to "Ramsifying" the entire program,[18] thereby associating with each data structure (or type) the full suite of behaviours in which it could play a

---

[17]They *do* determine how they are used, of course—in one obvious sense. But the data structures themselves also play a role in causally influencing the outcome. We *think* that the program is "the director" of this activity, and the data structures more like its objects or "patients." But the merit of that asymmetry is partly what is on the examining table.
[18]Rey (2024).

part. That is, to use a visual metaphor, instead of having the semantical analysis "pick the program up" by the functions, methods, and procedures etc., with the whole web of actions and behaviours and interconnections over myriad data structures hung from that "factoring" of it, we could imagine a (Ramsified) analysis that instead picked up the web of program-cum-data-structures by each *data structural type*, with the program and procedural interactions hanging from or spread out underneath that.

I.e., to put this in language that will be more familiar to computer scientists: the proposal is to conduct something like a more fundamental, higher-order refactoring than normal: organizing the program by the data structures, first, with the behaviour-specifying procedures being subsidiarily organized under them, rather than the other way around.

4 Object-oriented languages

As already intimated, this last suggestion is basically the strategy underlying the shift from functional/procedural to object-oriented programming, with "methods" (specializations of procedures) organized by the types of data and records and so on they apply to, rather than being assembled into a single large procedure.[19] The proposal may seem different in one respect, though presumably not an orthogonal one: whereas object-

---

[19]Informally, if one views the complex somewhat matrix-like graph of a program plus its entire associated plethora of data structures as a mass of spaghetti, "functional" or "procedural" programming can be viewed as a strategy of "picking the graph up" by each of the procedures, and having the specialization to different types of data fan out below them as cases, whereas object-oriented programming (class-based, etc.) can be viewed as picking up the same mass of spaghetti by the types of data structure, and allowing the various types of procedure (called "methods" in this case) fanning out below them. The difference, thus, can at least at a superficial level be viewed as one of how programs are *presented*, rather than as an inherent property of abstract structure—though that is not to deny that the difference may lead to substantial differences in how problems are factored, how code is (otherwise) organized, etc.

oriented programming is a practice of organizing programs in this way for effective reasons (how they are arranged as texts, and how they are processed), our proposal is framed in semantical terms, as regards their *analysis*. The fundamental insight underlying the idea of compositionality, however, is that both syntax (i.e., effective structure) and content (semantic meaning) are compositionally configured in roughly parallel ways. So the two approaches may have more in common than might at first appear. Perhaps they can be—maybe they even already are—combined, as follows: architecturally, if a procedure-cum-data-structural configuration is organized as an object-oriented program, then the way theoretical computer science (V2) presently analyses its semantics could be counted as—again, contrary to the glib summary—a direct analysis of the objects' (not their methods'!) procedural semantics.

This analysis suggests that, with respect to V2, the summary should at a minimum be revised as follows, yielding a somewhat less glib version that I will call the "revised summary," consisting of α, γ, and δ, describing V1, V3, and V4, respectively, as above, but with β, describing V2, modified as follows:

β′)   V2 provides an *indirect* account of the *procedural* semantics of the process' constituent data structures, inputs, outputs, etc. (and also, as we will see in a moment, of its objects, classes, types, etc.), by giving a *direct* account of the *declarative* semantics of the program, where programs are taken to be effective specifications or prescriptions *about* (in the classic declarative sense) *effective activity, process, and behaviour.* "What happens," on this reading, can be understood both (i) as the declarative interpretation of the program, and (ii) as the procedural semantics of the data structures, objects, etc. to which the program (declaratively) refers.

If the revised summary makes more sense than the glib one, which I believe it does, should it be adopted straightaway?

Alas, no—for a host of reasons, including at least the following four. (We are not "there yet," but we are making progress; while the revised summary β′ cannot be accepted as is, the suggestion contains within it ingredients that will figure in the proposed synthesis of V2–V4 to be suggested in chapter 8.)

1. As stated, revised suggestion β′ remains vulnerable to the Meccano critique. It overstates the case to interpret any suite of behaviours that a set of complex configurations of parts can undergo as their procedural *semantics*. Some warrant must be provided for interpreting the situation as semantic or intentional in the first place. It is not enough that the program indicates that suite of behaviour in a recognizably semantic way—viz., by specifying them in something like a language, which is mapped onto those behaviours in something like a traditional denotational fashion. That might justify using the term "the *declarative semantics* of the program," but it does not justify calling that which is thereby specified "the *procedural semantics*" of anything. One could as well say, to return to a previous example, that a program for a milling machine (declaratively but effectively) specifies the "procedural semantics" of the pieces of metal upon which it operates.[20] Or claim that what a recipe specifies are the *procedural semantics* of the flour, butter, and yeast that it recommends combining.

2. The revised suggestion (β′) provides no way of connecting the program's "specification of procedural behaviour" with the as-yet-untheorized but, as we saw, fundamentally important "reaching into the task domain" semantics that programmers attribute not only to data structures but to procedures (program fragments) too—taking them to designate, or represent, or bear some sort of intelligibly intentional relationship to entities and phenomena in the program's task domain. For

[20]Roughly this point is made in "100 Billion Lines of C++" (1997).

example, the revised suggestion would provide no ac-
count of what program fragment

```
current-floor ← current-floor + 1
```

has to do with elevators or floors, or what

```
(highest-paid-employee-of institution-27)
```

might have to do with employees, institutions, and sal-
aries. Because of this, the proposed framework does
not even begin to provide wherewithal with which to
begin to understand the *norms* that, in any intentional
system, must govern the system's behaviour overall.
That is: it does not work to rationalize V2 with what I
took to be the substantial merit of V3.

3. The third point is related to the first two. Although, in
the revised view, the semantics of programs is charac-
terized as declarative, no genuine sense of *deference* has
been retained, emptying the suggestion of substance
that I take to be fundamental to any enterprise worthy
of the label 'semantics.' Programs, on this view, specify
whatever behaviour over data structures they specify.
Nothing more—and therefore nothing interesting—
can be said. Since the data structures being manipulated
are not in turn understood as standing in deferential se-
mantic relation to any domain (task or otherwise), no
norms on their treatment are theoretically visible. De-
rivatively, as a consequence, no norms come into view
for programs, either.[21]

4. To bring the analysis full circle, the revised suggestion
remains utterly ill-suited for dealing with reflection. As
is so often true, what is foregrounded in—in fact what
is crucial to—the reflective case turns out to be

---

[21]Cooking recipes are presumably held accountable to a background
norm that what is cooked be good to eat. On the revised suggestion
(without declaratively interpreting the data structures), nothing even
that general is on the table for programs.

diagnostic of problems in our contemporary under-
standing of computation in general. The problem, to
put it most baldly, is this:

> *Reflection can neither be defined nor understood*
> *in purely mechanical or behavioural terms.*

Reflective systems are systems that reason, manipulate
structures, etc., that are *about* the system itself, where
the critical notion of "about" is, and must be, declara-
tive or representational—and in virtue of that fact, *def-*
erential. And because data structures are theorized, in
the revised conception, only procedurally, that revised
conception is not powerful enough to comprehend re-
flection at all.

The equivocation between the term 'program' and 'process' in
the last point (#4) indicates the real difficulty with the revised
suggestion. If it is the *program* to which reflective reference is
made, *during the course of execution*, then the program must be
part of the computational process—i.e., must exist, along with
other data structures, as a structural form capable of being in-
spected, manipulated, changed, etc. But then a problem arises.
On the suggestion under examination, two forms of *declarative*
semantics are implicated for reflection: one in order to allow the
reflective program to refer to the program, program fragment,
contextual information about the program's dynamic behav-
iour, and the like, of a sort reminiscent of (if not the same as)
that in terms of which we have understood VI and called "de-
clarative," and a second one, relating the thereby-referred-to
program to the behaviour that it will or does engender, upon
being executed. That is, the program-process relation (labeled
'α' in figure 1), stipulated on the current suggestion to be un-
derstood effectively but nevertheless *declaratively*, becomes a
constituent in the on-going dynamic reflective process (and in
the general case, as is certainly true in 3Lisp, becomes a relation

that itself needs to be able to be referred to[22]). But according to the suggestion, the only "semantical" properties of process constituents visible to theoretical analysis are *procedural* ones. Not only would the revised summary render invisible, for purposes of theoretical analysis, the fact that the program/process is *reflective*, but the constitutive relation between programs and the processes or behaviours they give rise to when executed—the *very subject matter of the semantical analysis of programs*, according to the revised suggestion, to say nothing of being evidently fundamental to the reflective situation—would also not be considered semantic, and would therefore also be rendered invisible, on the terms of the suggestion.

(In passing, it is worth noting that one of the signal virtues of 2/3Lisp's integration of both declarative and procedural aspects of a program within a general integrated notion of semantical significance is that, by design, it allows for all of these relations to be considered explicitly: the procedural consequences of the data structures, the declarative import of those same data structures, the procedural consequences and declarative import of reflective programs, and the procedural consequences and declarative import of the programs being considered by the reflective code.)

With respect to the current discussion, however, it becomes clear that, even as revised, the suggestion starts to disintegrate. And not just the suggestion, but the entire discussion. So abstruse do the intricacies grow between and among programs and processes, procedural and declarative interpretations, exterior and interior processes, implementing processes and implemented processes, behaviour mechanically understood and behaviour understood under (representational) interpretation, etc., that confidence slips away as to whether the morass will ever be sorted out. (I hereby offer a bottle of single malt to any

---

[22]Cf. footnote 42 in §4.V4 about 3Lisp's '↓'. Note the *declarative import* of '↓' is that it denotes the denotation function: $\varphi(\downarrow) = \varphi$.

reader who has not yet abandoned ship.)

On the other hand, for better or worse, the accumulating intricacies in no way undermine the importance of the overarching goal: to understand how deferential, representational content can be accommodated within a programming framework in such a way that the analysis of that declarative, representational semantics meshes coherently with what computer science currently calls semantics, in turn in such a way that the (mechanically engendered) behaviour can be held normatively accountable to it.

Meeting that goal is a prerequisite to defining reflection.

5 Discussion

At least half a dozen issues are on the table.

1. I have argued that programs, data structures, and the objects and classes in object-oriented and class-based languages all warrant semantical analysis—semantical analysis that, perhaps among other things (in particular, in addition to their role in and impact on processing) deals with their *deferential relation to their task domains or worlds*. At a general level, both the form of analysis that this will require, and the relations between and among these types of entity, remain unspecified.

    Are programs *about* data structures, or are data structures *parts* of programs? If the program/data structure distinction is real in classical cases (e.g., in functional or imperative languages), is the same distinction maintained, even if structurally configured differently, in the object-oriented case—e.g., with the "program" function shouldered by "methods"—or is the merit of the object-oriented approach that it dissolves the very distinction? How are we to understand referential and other "aboutness" relations *internal* to a computation (and fundamental to reflection) in relation to referential and "aboutness" relations to the wider world? And so on.

    (All of these questions are pertinent to anything that

might be swept up in the phrase "programming 2.0," based on machine learning networks.)

2. An implicit but intuitive distinction, as yet unmentioned, infects our informal understanding and formal analysis of programs and data structures, even if it is not yet clear how to define it: between *passive entities*, on the one hand and *active processes or behaviour*, on the other. Needless to say, 'passive' does not mean *static*. Data structures are usually (though not always) *dynamic*, in the sense of being created, changed, updated, deleted, etc., during the course of program execution.[23] These changes, though, typically result (or so at least we conceive of it) from their being patients or targets of agency imposed from the outside—typically, agency arising from the "running" of the program or methods associated with them. Not being active agents or agencies in their own right, data structures do not typically "up and do things" on their own—or anyway, again, that is not how we understand them. In spite of being dynamic, they are not (or we do not typically conceive of them as) active agents in their own right.[24]

Strictly speaking, of course—and curiously—programs, *qua programs*,[25] count as passive, too, even if their raison d'être is to engender behaviour. If they are "self-modifying," as may be true in cases of reflection, they may be dynamic in the sense of being altered during the course of execution. But like data structures, programs

---

[23]If they were not dynamic in this sense, a good optimizing compiler would or should convert them to constants when the program is compiled.

[24]This is even true in contexts, such as in Haskell, based on lazy evaluation: the activity that affects (or even creates) data structures happens "at run time," but one says that the data structures are *evaluated* then, not so much that they *do* things according to such a temporal regimen.

[25]I.e., the "texts" or symbolic structures of which semantic analyses are given.

are also not usually understood to be processes or agencies in their own right. Rather, they are taken as guides to or specifications of active behaviour—behaviour that arises from, or as I have been saying, is engendered by, their being "executed" or "run," during which they are treated as patients or objects of their activity-engendering "interpreters" (what in *Legacy* I call "processors").[26] This is manifest when we edit programs, for example: we edit something both passive and, at that moment, static (the "program text," as one might put it), not an active agency.[27]

Admittedly, we often use the term 'program' to refer to the active process resulting from its execution: "the program is indexing the array," "the program is looping," "wait till the program issues a prompt," and so on. Yet it is probably most consonant with current theoretical frameworks (V2), and perhaps with tacit programmer understanding as well (V3), to understand such phrases metonymically.[28] The passive, textual or structural entity—the edited program, written in a programming language, to which (crucially) we assign "semantics" (even if it is semantics of a Meccano variety)—has

---

[26]The separation of the locus of agency from the millions of data structures is a critical part of all contemporary architectures—not obviously a feature of the brain, for example. What it would be to have a genuinely active field is an interesting question.

[27]Think of the difference between working on an airborne drone, on the worktable, between flights, to adjust its motors or wings or whatever—vs. trying to catch an errant bat loose in one's living room. Editing programs is like the former activity, not the latter.

[28]At least it is metonymic given current views of computational ontology. Cf. "The Correspondence Continuum" and the discussion of the fan calculus at §3.3. The distinction between the static specification of the activity and the activity thereby produced (when the program is "run") could be treated, in the fan calculus, as a distinction that can be "opened" or "closed," depending on context.

ontological pride of place.[29] So we have two types of
passive entity, programs and data structures, contrasted
with active processes or dynamic behaviour.[30]

Except in cases of self-modifying programs, that is,
we might say that: (i) data structures are typically
viewed as passive, even if (typically) dynamic; (ii) pro-
grams, as static (unless "self-modifying") but dynami*cal*,
in the sense of denoting—and engendering—activity.
Whether either distinction (program vs. data structure
and active vs. passive) is maintained or dismantled in
the case of object-oriented languages is, as I have said,
unclear. Illumination awaits proper theoretical treat-
ment.

3. Within the realm of processes, in §4.ᴠɪ I made a three-
way distinction among: (i) an outer process ʀ; (ii) an in-
ner process ʀ′ that results from the execution or inter-
pretation of the program, whose "domain" is the field of
data structures or records on which it operates (a realm
of data structures that in 2/3Lisp I call the "structural
field," and which, as we have seen, denotational analyses
of programming language semantics mathematically
model and take to be the semantic *range* of programs);
and (iii) a doubly-interior process ʀ″ that is the locus of
anima or agency that "runs" (or as many computer

---

[29]Though pedantic, it does not contravene sense to say "the process that
this program engenders when being executed," but "the passive text that
causes this program to exist when it (the text) is run" is not just akward
but conceptually awry.

[30]Though 'procedure' is not a term I use technically, here, I will take pro-
cedures not to be active or behavioural, per se—but also not to be passive
like programs, in the sense of being something that could *lead* to a pro-
cess being created through the agency of an interpreter or processor. Ra-
ther, I will take procedures to be "abstract" entities, roughly on the model
of being "a way of doing something." Thus a program might be taken as
specifying a procedure; a process, as *realizing* or *implementing* it.

scientists would say, "interprets") the program.[31]

4. There is the issue of input and output (I/O). Questions arise not only about how to treat I/O in its own right, but also about its relation to the types of entity just discussed: programs, data structures, and both outer and interior processes. And of course the question remains open—in fact as yet unasked—as to the semantics, if any, of the input and output "signals." If light reflected off a traffic arrow impinges on an autonomous vehicle's light receptor, causing an internal data structure or memory record T to be constructed, which data structure T in turns leads the vehicle to veer to the right, should structure T be considered to represent the (proximal) impinging pattern of light, the (distal) arrow, the route towards which the arrow points, the direction the system should (or does) turn, or something else entirely—or not to represent anything at all? And what about the pattern of light that impinges on the light receptor?[32]

5. Though I keep broaching the topic of object-oriented languages and their kin,[33] we can hardly be said to have

---

[31]One might view analysing the outer process R as the computational equivalent of understanding the human case at the "personal" level, and treatments of both interior processes R′ and R″ as computational analogues of "sub-personal" analysis—something I will in general call "interior" analyses. (One might think that 'sub-system' would be a more natural generic term. But as this section will argue, I believe we get a deeper understanding of computation by giving processes and activity greater ontological priority, rather than (as is almost universally done today) treating them as "systems that behave.") Moreover, as is evident, this three-way distinction is relative to frame of reference (and potentially recursive); what is "interior" from one perspective may be "outer" from another, and vice-versa.

[32]Note that the confusion of this situation should not lead us to abandon semantic analysis, because *the normative deference remains*.

[33]By 'their kin' I mean to include class-based systems such as Simula.

a semantical grip on them of the requisite sort. At least paradigmatically, it would seem that the sorts of type or class[34] defined in object-oriented programs represent (stand for, model, or in some other way intentionally signify) *kinds of entity in those programs' task domains*— or perhaps more significantly, *kinds of entity* that, if not themselves part of the task domain, are nevertheless *normatively accountable to that domain*, or that *play a role in the normatively accountable behaviour of the program as a whole towards that domain*. Needless to say, as I have been saying throughout, what currently goes by the name of "programming language semantics" for object-oriented languages does not deal with these deferential real-world relations at all.

   The point holds even for entities that might be considered "within the program's purview," if not outright

---

[34]At this level of generality I am not distinguishing types from classes— just as I am not distinguishing a variety of semantical notions (semantic value, meaning, significance, denotation, etc.). Both sets of distinctions matter, and would need to be made once a tenable semantico-ontological frame was in place. My aim here, though, is prior: to determine what would make a proposed semantico-ontological frame tenable in the first place.

   It should be noted that the topic of what, if anything, distinguishes types from classes is vexed in computer science. On one common intuition, the 'type' of a (computational) object has to do only with its "interface" (the sorts of messages or requests to which it can respond; the sorts of "answers" or "behaviours" it can produce), whereas classes deal with how the object is "implemented"—a conception that would assign types but not classes to primitive entities (where no issue of implementation is supposed to be revealed). This distinction is not universally agreed, but it betrays a fact that likely is: that what it is to be a type, or a class, or for that matter an object, has to do with its (mechanical) behaviour within the computational setting. Nothing about what I am here calling genuine (deferential) semantics is admitted into the discussion—in spite of the fact that, as evident in §4.v3's discussion of programmers' tacit intuitions, such types and classes are invariably named for, and in my judgment represent or in some other way (genuinely) semantically signify.

internal. Suppose a graphics program defines a class RECTANGLE, for example, of which it then passes around instance T_1. Needless to say, T_1 is not *itself* a rectangle, being neither two-dimensional nor rectangular. Rather, T_1 would presumably "correspond" to a rectangle in some way—it might correspond to (and thus to represent, in at least some sense) a rectangle already drawn on a screen, or perhaps it might serve as a template or procedure that could be used to (actively) *draw* such a thing.[35] Similarly, the program fragments discussed above, if written in object-oriented form, might define such classes as EMPLOYEE or ELEVATOR. What are (how should we understand) the semantics of such classes— of their instances, their "methods," their behaviour, etc.—including both the behaviour that they engender, upon being "called," and their relation to actual rectangles, in-the-world employees and elevators, etc.?

6. A tremendously important question, as yet also un-addressed—of particular but not unique importance for reflection—has to do with how issues of semantics relate to compositionality and complexity. Note, for starters, that the five foregoing points attest to the complexity of even the simplest computational systems. Adequate analysis will need to be able to expand to include streams, modules, APIs, processes, servers, packets, locks, recursively-defined data structures, distributed processes, and a veritable host of other by-now common computational sorts of entity. And at a more mundane level, beyond the suggestion in §4.V.2 that program definitions be semantically treated as *double claims*, we have yet to say anything very detailed about the semantics of complex expressions, or about entities "defined" by complex expressions that take definitional form.

---

[35]Or to a class of such rectangles!

Whether, from a semantical point of view, the semantics of such computational types can or should be able to be defined in terms of the semantics of the smaller or more elementary pieces in terms of which they are defined is yet another issue that is not clear. That is: if a computational construct α is effectively "definable," in the ordinary computational sense of that word, in terms of a suite of more elementary constructs $\alpha_1-\alpha_n$,[36] grammatically assembled according to arrangement or configuration β,[37] the question has to do with whether the deferential semantics of α is or should be definable in terms of, or even be determined by, the deferential semantics of $\alpha_1-\alpha_n$—perhaps in ways specified by a rule associated with β, as would be the case if the semantics of α were traditionally compositional, or perhaps in some other way. The "double claim" proposal of §4.V.2 suggests that the answer is *no*. If that is correct, then current practices of semantical analysis will require an even more thoroughgoing overhaul.

For example: suppose class POINT (designed to represent points on a two-dimensional plane or surface) is defined in terms of a pair of representations «x» and «y» taken to represent real numbers. At issue is the question of whether the deferential (representational) semantics of instances of the class POINT can be defined in terms of the deferential (representational) semantics of «x» and «y». The evident answer is again *no*. Instances of POINT, in particular, deferentially represent *points in a two-dimensional plane*, whereas «x» and «y» would deferentially represent real numbers. There is of course a relationship between the point and the represented numbers—with «x» and «y» most likely representing

---

[36]Perhaps including recursive mention of α itself.

[37] I.e., so that the constituents $\alpha_1-\alpha_n$ would be indicated as the leaves of the parse-tree ζ of the complex expression α.

either the x and y (or ρ and θ) coordinates of the point with reference to a presumed measuring frame. But the point itself is not the pair of real numbers, any more than an airplane is its balsa model.[38]

I will moot just one suggestion, here, to be picked up later, about why this discussion might ultimately lead to a positive outcome. Note, to set the stage, that the practice of defining a class or abstract data type is often described as being pragmatically beneficial—beneficial for purposes of program readability, maintenance, modularity, type-checking, etc.—but nevertheless as being "semantically neutral." The program would be the "same program"—or at least so it is commonly thought—if the abstraction were eliminated and the implementation details that it hides exposed. This is manifest in norms on compilation, which (unless the classes are exported) is usually free to dispense with the abstraction overhead and merely "implement" the behaviours of the methods and procedures directly in terms of the implementing structures.

Given this context, suppose that a programmer defines a class C in terms of a variety of "class-internal" methods and data structures $c_1 \ldots c_n$. On the analysis suggested here, C itself, and $c_1 \ldots c_n$, will have *declarative* meaning or semantic significance, in addition to whatever *procedural* effect they give rise to, when executed. It may be—likely will be, in fact, given the way these languages work—that the procedural consequence of C may be "implemented" by the procedural consequences

---

[38]It will not do to argue that the point and the pair of real numbers can be "identified" because they are isomorphic domains, under certain circumstantially operative conditions. They remain different things. In the reflective case, if a system is not only to represent but to operate on, and perhaps to modify, some of what it represents, it must know the difference between two things that are isomorphic, lest it modify the wrong one.

of $c_1 \ldots c_n$. However the *declarative import* of C may not be evident at all, from the declarative import of $c_1 \ldots c_n$— and, more importantly for the present discussion, as just argued in the case of class POINT, will very likely not merely be an assembly of the declarative imports of $c_1 \ldots c_n$ in any sense.

Why does that matter? Because the *norms* that govern the behaviour of C (implemented by the behaviours of $c_1 \ldots c_n$) may be intelligible only with respect to the declarative semantics of C. Understanding the norms governing the behaviour of the class's methods, that is, will always depend on understanding *what the class is intended to represent*. Suppose class CIRCLE is defined with the following methods:

```
SET-RADIUS(R)           ; SET RADIUS OF CIRCLE TO R
SET-CENTER(X,Y)         ; SET CENTER OF CIRCLE TO <X,Y>
RADIUS()                ; RETURN RADIUS OF CIRCLE
X-CENTER()              ; RETURN X-COORDINATE OF CENTER OF CIRCLE
Y-CENTER()              ; RETURN Y-COORDINATE OF CENTER OF CIRCLE
ON-CIRCLE(X, Y)         ; TRUE IF <X,Y> LIES ON CIRCLE, ELSE FALSE
```

It will presumably be a *norm* on the behaviour of ON-CIRCLE (assuming SQRT "is" the positive square root function[39]) that:

```
C.ON-CIRCLE(X, Y) will be TRUE just in case
      C.RADIUS() == SQRT((X - C.X-CENTER())^2 + (Y - C.Y-CENTER())^2)
```

The reason that this norm governs is because instances of C represent *circles*. Only knowing that that is C's declarative import makes that fact intelligible. (Ontologically: it is circles being C's declarative import that makes the fact *true*.)

That is: far from being a matter of mere convenience, defining classes and abstract data types may be

---

[39]"Is" that function in the sense that it would be universally understood as computing it. In 2/3Lisp terminology, SQRT would be understood as designating the positive square root function, and returning the numeral (fixed or real) that is a normal form designator of that square-root.

semantically critical to the issue (let alone to the determination of the issue) of whether a program is *sound*. We are a long way from having programming environments (IDEs) that check the soundness of our programs, but the possibility of working towards such a goal is the kind of consequence that ought to come out of an adequate semantical analysis along the lines suggested here.

It is unlikely that these six issues exhaust the challenges that need to be addressed before we can genuinely claim to have a grip on the semantics of computational procedures. But at least these six can stand witness to the character of the task ahead. They also stand witness to the astoundingly impressive tacit understanding that programmers bring to their work; navigating all of the complexities cited in this chapter, and others besides, are part and parcel of the workaday life of any computational practitioner. And they remind us, finally, that our current semantical frameworks occupy a very modest corner of the full semantical-cum-ontological territory of contemporary computational systems.

6 Logic

What can we conclude from the analysis so far? At least this: what separated the 2/3Lisp approach from standard construals of computation ran deeper than a simple divergence in understandings of the term *program*. That was just the tip of an iceberg. In an attempt to reach deeper, chapters 4 and 5 shifted the focus to semantics and associated models of computing. But these analyses also failed to lead to a satisfactory resolution. Though the investigation has yielded undeniable insights en route, it has also been dauntingly complex, verging more than once on the scholastic.

Can we find a simpler approach, in terms of which the intricacies revealed so far can be explained—maybe even predicted?

Developing such a vantage point is the goal of chapters 6 and 7. The result is no panacea. It resolves none of the detailed technical issues. But it explains, in relatively simple terms, why we have ended up in the situation we are in, makes sense of the baroqueness of the first two diagnoses, and explains various additional oddities encountered en route, including our emblematic example of Lisp's taking the value of both «3» and «(QUOTE 3)» to be «3». It also reveals, in stark terms, the task that must be undertaken to develop a better alternative—a successor account of meaning and mechanism adequate to computational practice in general, including to the notion of reflection.

As explained here, the fundamental issue bedeviling the analysis springs from an irreconcilable conflict between (i) the thoroughly mechanist (effective) conception of computing that pervades computer science, which has been a stalking horse in this discussion since the beginning, and (ii) the non-effective

semantical nature of symbols, meaning, interpretation, etc., that has permeated our understanding of intentional phenomena for centuries, including the model of logic on which computer science was founded. More specifically, computer science has given the effective mechanism dimension of computing pride of place, and sidelined the non-effective semantical or meaningful dimension: issues of representation, semantics, and intentionality more broadly. An adequate understanding of computing, I have argued since the beginning, must give these non-effective meaning dimensions as much attention as the dimension dealing with effective mechanism, and provide a framework for understanding how they are integrated in computational systems. Instead of doing this, the field has swept all genuinely semantical issues under a blanket, projected the entire subject matter onto pure mechanism, and distractingly redefined all semantical vocabulary to refer to mechanical entities, phenomena, and effective relations.

To explain the situation, and put it in historical context, this chapter takes a step back and lays out the framing assumptions of logic, prior to and free from any computational considerations. In some respects this will be familiar ground, but the analysis is unusual in five respects:

1. It pays greater than usual attention to the background intellectual and metaphysical assumptions on which our classical understanding of logic relies;

2. It resists premature tendencies to reduce logic to syntax or formal operations;

3. It postpones analysis of the mechanization of logic until the complete picture of logic is in place;

4. It eschews all notions of modeling (e.g., as in model theory), in order to get at the underlying issues being modeled; and

5. It is otherwise framed in very general terms—so that, to take just one example, the resulting framework applies just as well to clocks and other "meaningful mechanical

systems" as to the sorts of sentential systems we take as paradigmatically logical.

One reason for reviewing the basics is because many computational perspectives on logic omit or gloss over what I take to be essential not only *to* but *about* the logical tradition. Evidence that something is awry can be seen in the clash in the two fields' respective use of technical terms. As we have seen throughout, they have a tendency to use the same terms for different things: *interpretation*, *reference*, *semantics*, etc. But the terminological discrepancy is symptomatic of more profound conceptual divergence. Disambiguating the terms reveals the extent of the difficulties; per se, it does not resolve them. That task remains for us to take on—or at least to take primitive steps towards—in a later chapter.

1 Preliminaries

Half a dozen preliminaries.

1. PROPERTIES. It is customary in the basic sciences, and often in the philosophy of science, to assume that everything fundamental is causal—that the only "real" properties are causal properties: that those causal properties are what science is about; that causal properties are "intrinsic," as opposed to others characterized as "extrinsic" or "relational"; that scientific explanations must be given in terms of causal properties; and so on. Being square, made of titanium, or having a density of 10 grams per cubic centimeter would on this account be considered real or intrinsic—"proper" to their subjects. Being more than 100 miles from the nearest bristlecone pine, having been owned by Elvis Presley, or resembling the Mona Lisa would be viewed as relational or extrinsic—and, for that reason, as outside the realm of scientific interest, at least as science is traditionally conceived.

   Cognitive science and the philosophy of mind, which

are not basic sciences, have extended their compass to deal with what are called *functional* properties. Functional properties are defined not in terms of the physical constitution of the item that exemplifies them, in the ways that strictly causal properties are, but in terms of the way that the item functions, or the role it plays, in a larger system. "How something functions," or "what role it plays," in such characterizations, is nevertheless informally considered to be something like "how it functions causally," or "what causal role it plays." For this reason, *instances* of functional properties (i.e., the functional property exemplified by a particular concrete individual—sometimes called "property instances" or "tropes") are assumed to be causal. In one system the functional property of opening a door might be played by pressing a lever; in another, by releasing a spring. The fact that each instance of a functional property is causal does not legitimate characterizing the overall property causally, however. The fact that functional properties support multiple realization blocks any specific causal power from being part of the property's identity conditions. This is one reason why I have used the term 'effective' to characterize such properties; an effective property is one that on each occasion is instantiated by a causally effective property.

These considerations might suggest, if we are aiming for an intellectual account of computation, that it would suffice to restrict our discussion to effective properties. While this is a mandate that computer science has adopted, in my own view *nothing could be further from the mark*. To impose such a restriction, especially in advance, is to abrogate the entire discussion of what computation is. At issue in this chapter are issues of semantics, logical interpretation, meaning, reference, intentionality, and so forth. And as I have emphasized since the outset, and as I will repeatedly stress here, many of these phenomena essentially involve properties and relations that are not effective—not causal even in particular cases, and thus not

functionally effective in the current sense. One of my primary intellectual goals, in fact, is to identify which properties (and property instances) in both logical and computing systems must be functionally effective, and which ones not. Doing so requires approaching the subject with a methodologically open mind.

In what follows, therefore, I use 'property' broadly, to include not only intrinsic, causal, and effective properties, but also those that would classically be deemed relational and/or extrinsic, and other sorts. Similarly for such affiliated terms as *state*. It follows that every object, by my account, at every metaphysical instant, exemplifies an unbounded number of properties, and is in an unbounded number of states—many of which are constantly changing (such as whether, at any given instant, the object in question is, or is not, or most closely approximates being, an even number of centimeters from the apex of the Washington Monument). So be it. My concern is to identify which vanishingly small subset of this unmanageable profusion matter to its status as a logical or computational system—which of them, to put it etymologically, are "proper to" their exemplification of being logical or computational, or at least of being parts of systems of those types.

2. EFFECTIVENESS: It does not follow from the foregoing that issues of causal efficacy will not be of central importance here. I have used the term **effective** to label those properties that, when exemplified (i.e., in their property instances), are capable of causing an at least potentially observable or consequential physical reaction or change in something else (that is: of causing a correspondingly effective reaction in something else). I mean, in characterizing it this way, to get at the sense in which theoretical computer science is considered to be

a theory of *effective* computability.[1]

Some may protest that the term 'effective' is used in computer science to refer to entirely abstract properties of mathematical functions, unconstrained by considerations of physical causality. As made evident in chapter 4, I disagree with this claim, but in any case will presently set aside ascribing computational properties to purely abstract structures altogether, so the objection is moot at least in the present context.

3. MECHANISM: Again as has been suggested since the Introduction, I am using the term **mechanism** broadly, to include any causally-connected nexus or closure bound together by a network of effective properties, and functioning as a unit (again, in some appropriate sense of 'unit').

There are some, especially in biology and cognitive science, who opt to use the term 'mechanism' more narrowly, to include only those systems or devices whose overall operation can be understood via something like the sum of the operations and functions of their parts, in the manner in which they imagine that traditional mechanisms such as clockworks and engines are understood by their designers. Their aim is to contrast mechanisms so conceived with complex systems exhibiting (so-called) emergent behaviour, with some supporters of this distinction believing that the properties of the parts of such "non-mechanical"

---

[1]There are those who believe that computability and complexity results, which implicate the notion of effectiveness, can be formulated completely abstractly, as a branch of mathematics. This is not the place to address this issue (though see point #5, below), but for the record I do not believe such claims—and it is my sense that, although this was once a widely-held view, it receives little endorsement today, especially among young computer scientists. As is increasingly recognized, constraints of physical causality permeate the foundational assumptions that underwrite the mathematical assumptions of computation and computability theory.

(in their view) systems require an inverted form of expla-
nation, with the parts derivatively understood in terms of
the function of the whole (using such notions as "down-
wards causation"). Systems with emergent behavior, on
such a view, are dubbed "non-mechanical."

I make no such distinction here. For one thing, I be-
lieve the considerations motivating views on emergence
are largely epistemic, whereas my present concern is with
the ontological conditions on logic and computing. Sec-
ond, as an engineer, I am suspicious of the idea that tradi-
tional mechanisms can be understood solely in terms of
the workings of the parts, treated as isolated individual
(physical) units. Even ontologically, the type identity of
parts of traditional mechanisms (e.g., being a power sup-
ply) is surely at least functional, rather than purely causal,
depending on systemic properties of the system into
which it fits. And it is not traditional to call such func-
tional properties emergent. But be all this as it may, it will
not help the present project to separate out emergent
properties or complex systems, or to restrict the use of the
term 'mechanism' to a subset of those systems that work
in virtue of the total causal interaction of their ingredients
taken as an ensemble.

In what follows, therefore, I include, within the cate-
gory of mechanisms, non-linear and complex systems
with what is called emergent behavior—including many
contemporary machine learning networks.

I will admit one further fact about effective (and me-
chanical) properties, and about mechanisms themselves:
that they apply to entities that we would informally say
have "functions"—have a telos, are for a purpose (though
that purpose may well, like intentional interpretation-L,
arise from external attribution or context). If I say that, on
a walk, I found an old mechanism in the woods, it is rea-
sonable for you to ask me "What is it for?", "What do you

suppose it was intended to do?", etc. While this notion of function is relevant to the idea of functional properties, it is not the construal of (proper) function popularized in theoretical biology, derived from its role in evolutionary fitness or survival. Beyond that I will not say, as the notion of function will not play a central role in the analyses to be developed here.

4. OSTENSION: Systems of formal logic, and not a small number of computational ones, are often introduced by example, or at least via schematic template—i.e., by *ostension*—in such a way that various aspects of critical importance for our purposes are exhibited without comment or analysis (*shown*, as it is put, not *said*).

    It is common for an introduction to a logic, for example, to present it as consisting of a set of syntactic expressions formed from variables $x_i, x_2, …,$ predicates $P_1, P_2, …,$ relation symbols $R_1, R_2, …,$ atomic sentences $P_i(x), R_j(x_1, x_2, … x_k),$ etc., plus various complex forms recursively defined in terms of them, some assembled using logical operators '∧', '∨', '¬', and '⊃', and so on. But as discussed below, little if any attention is paid, in such ostensive definitions, to the question of what constraints govern the notion *being syntactic*—i.e., to what sorts of things would be legitimate, within such an enumeration. These and similar questions will come up for discussion below. As much as possible, therefore, in this discussion I will avoid reliance on ostensive definitions.

5. MATHEMATICS: In part because of the support of multiple realizability that underlies talk of functional rather than causal properties, effective properties (in logical and computing systems) are typically not identified in terms of units (kilograms, seconds, etc.), in the way that is standard in the physical sciences. When logical and computational (and perhaps mathematical[2]) subject

---

[2]I would considered mathematics itself to be mathematized when

matters are mathematized, therefore, numbers and other mathematical structures are used *simpliciter*; viewed as dimensionless, not ratios to standard metrics.

Whether for this or other reason, as has already been pointed out, logical and computing systems are frequently not only mathematically modeled, which I believe is what underlies this practice, but theorized as if *they themselves were mathematical structures*. In our case there are two problems with this practice. The first was identified in the Introduction: confusing/conflating the concrete and the abstract obscures the critical role played by issues of effectiveness in computing—a notion foundational to the whole subject matter. A second problem is one that often plagues ostensive definitions: implicit norms satisfied by the provided exemplars can play a crucial but unacknowledged role in the ensuing theory. A salient example in the case at hand—definitions of Turing machines—is reliance on "reasonable encodings," obscuring the centrality of considerations of efficacy and physical realization.[3]

In what follows, therefore, in order to make these fundamental constraints maximally visible, I will eschew mathematical models entirely (for syntax, semantics, and interpretation in logic, and in all discussions of computational systems).

6. The final preliminary has to do with time—specifically, with the fact that whereas computation is fundamentally concerned with process, the same is not true of

---

mathematical entities are defined in terms of models—e.g., when the number *two* is defined as "the set of all sets that have two members." But nothing hangs on this here; one might take mathematics to be inherently mathematized by its very nature—or to be dimensionless per se, or something else.

[3]See "Solving the Halting Problem, and Other Skullduggery in the Foundations of Computing" (Smith, 2014).

logic, especially as classically conceived. In spite of that issue's great overall importance, and indeed centrality to computing, however, temporality will not impinge on the analysis to be given here, given the level of abstraction at which the issues will be considered.

2 Overview

Turn then to the nature of logic—in particular, to what is known as "formal" logic, as it has been framed over approximately the last 150 years.

At the simplest level, logic is concerned with meaningful expressions in a language, and with relations between and among them. The expressions are assumed to be *about* something—to have a what is called a *semantic interpretation*, in the logical (not computational) sense of that term. As already noted, I label this sense interpretation-L, to distinguish it from the computational sense of interpretation, as for example in claims that Java is an interpreted language, which I label *interpretation-C*.

A diagram of the simplest possible such expression is given in figure 4. A single expression σ is taken to be "about" some entity or state of affairs or world τ. But logic does not just study individual expressions—even complex ones. Rather, it has to do with relations among expressions—in particular, with relations between one or more expressions $\sigma_1, \sigma_2, \ldots \sigma_k$, sometimes called the *premises* to an inference, and another, σ', sometimes called the *conclusion*, that—still speaking
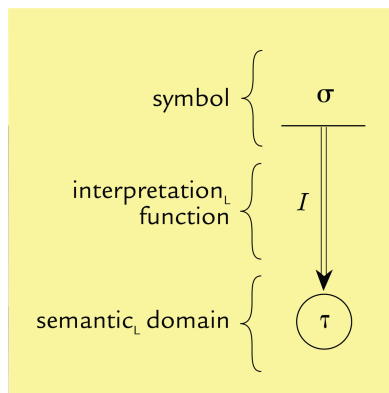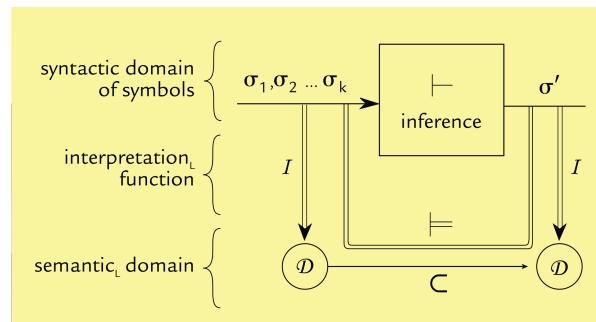

Figure 4 — Symbol interpretation-L

Figure 5 — Logical inference-L

237

informally—*follows* from $\sigma_1, \sigma_2, \ldots \sigma_k$. A more useful diagram, therefore, is given in figure 5. S is the realm of expressions; $\mathcal{D}$, the world of entities or situations the expressions are about (i.e., the realm of semantic-L values); and I, the interpretation-L relation between the expressions and their semantic values, the elements in the semantic domain that they are about. Everything having to do with the semantic interpretations-L—the relations I, and the entities in realm $\mathcal{D}$, are informally (and often not very systematically) referred to under the label "semantics."

All sorts of complexities are typically defined, allowing the construction of complex expressions from simple ones, distinguishing notions of meaning from interpretation-L, etc. (e.g., in Frege's famous example, in order to say that "the morning star" and "the evening star" differ in meaning—or, as it is said, in *sense*—even though they coincide on their interpretation-L, what in this case might be called their *denotation-L*). Nevertheless, the fundamental structure of figure 5 underwrites all more elaborate versions. Moreover, this simple version will be sufficient in terms of which to develop the vantage point under discussion here, to compare logic and computing.

At least in formal logic, the realm of expressions S is taken to be *syntactically* individuated. As noted above, what 'syntactic' means is not usually explained (a convenience of ostensive introduction). Intuitively, expressions are thought to be the sort of thing we can write down—to be immediately accessible to perception, to "wear their syntactic properties on their sleeve." Three criteria are essential to that idea. First, the syntactic properties of an expression are assumed to be definable without reference to their semantic interpretation-L—that is, are assumed to be *formal*, on one understanding of that vexed term. Second, the claim that syntactic properties are *individuating* means that if α and β are syntactically indistinguishable, then they are assumed to be identical (to be the same expression) for all other logical purposes as well. Third—an issue that will

come to the fore when we talk about the mechanization of logic and computing—the syntactic properties of an expression are assumed to be properties of the sort to which a machine or automatic procedure can differentially respond. If α and β differ syntactically (implying that $\alpha \neq \beta$, by the second criterion), then it is assumed that it must be possible, at least in principle, to construct a machine that can turn on a light switch in response to a concrete realization of α and turn it off in response to a concrete realization of β. Similarly, assuming again that α and β differ syntactically, it is assumed that a machine could produce concrete tokens of some other syntactically distinct expressions α' and β' in response to concrete tokens of α and β, respectively, in such a way that a machine or automatic procedure could similarly respond differentially to tokens of α' and β'. That is, to employ the vocabulary of this book, in machinic contexts syntactic properties are assumed to be *effective*.

The fundamental challenge that logic addresses can now be characterized—again a challenge not usually formulated as such, but fundamental to the entire logical project. Logic is interested in what sentences or expressions "follow from" others. There are two ways in which this question can be addressed. Intuitively, whether a statement follows from another one is a matter of meaning and semantics—of what they are about, about what they mean, about the world or those situations that the statements point to. If I say that Alex is currently in Tromsø, then whether it follows that it is currently raining where Alex is, or that Alex is now north of the Arctic circle, depends on facts about a place a fair ways from where this is being written. If I claim that Bobbie is taller than Caitlin, on the other hand, then, independent of the truth of my claim, whether it would follow, if what I said were true, that Caitlin would then be shorter than Bobbie, depends not on facts about Bobbie or Caitlin, but about the notions (meanings) of *tall* and *short*. If, in contrast to both of these cases, I say that Alex believes that Bobbie is taller than Caitlin, then whether it follows that Alex believes that Caitlin is shorter than Bobbie depends, not on facts about Bobbie's and Caitlin's heights, but on facts

about Alex's mind.

The problem, for inference, is that we typically do not have direct access to all of the facts pertinent to whether one statement follows from another. The world—the total situation—is not directly inspectable. How then do we know anything? How can we figure anything out? For those facts or states of affairs that are effectively manifested in situations we have causal (effective) access to, we can often learn from perception and direct encounter. But it is fundamental to thinking that we can also *reason*—derive knowledge, or at least come to conclusions, from beliefs we already have, from things we have learned from others, by reading, etc. And—the crucial point, or anyway so it is assumed in logic—in such situations we can learn things because they follow from other things we know or have learned merely (or so anyway it is supposed in formal logic) *from their syntactic structure or "shape."*

If transitive relation R (such as being taller than) holds between x and y, and between y and z—that is, if $R(x, y)$ and $R(y, z)$—and if the transitivity of R is also explicitly represented, such as by the expression $\forall x, y, z\ [[R(x, y) \land R(y, z)] \supset R(x, z)]$, then it not only follows that R holds of x and z, but that fact (that $R(x, z)$), can be determined without needing access to x, y, and z (without needing effective access, that is, to the entities that the terms «x», «y», and «z» denote), or to the meaning of R. The sentence $R(x, z)$ can be concluded merely from the syntactic structures of the expressions $R(x, y)$, $R(x, y)$, and the foregoing statement of the transitivity relation. And as I have just emphasized, syntactic structure, to be syntactic structure, must be immediately and directly accessible—must be effective.

Consider the canonical case. If, in the world, all men are mortal, and if Socrates is a man, then Socrates must be mortal. This holds independent of anyone's representing it, knowing it,

or figuring it out.[4] Socrates' being mortal is a blunt ontological consequence of how the world is laid out. If, however, in addition to these ontological facts, we *know* that all men are mortal, or at least write it down, and we know or write down that Socrates is a man, then we can *figure out* that Socrates must be mortal, merely by examining the syntactic structure of the sentential representations of those two facts.

It follows, as depicted in figure 5, that there are two separate relations between the premises ($\sigma_1, \sigma_2, \ldots \sigma_k$) and the conclusion ($\sigma'$). One, called *formal* **derivation** or *formal* **inference**, symbolized as «⊢», as in «$\sigma_1, \sigma_2 \ldots \sigma_k \vdash \sigma'$», and contained wholly within the upper half of fig. 2, corresponds to what can be determined to follow from the premises purely in virtue of their syntactic properties[5]—those properties in virtue of which they are the elements of the syntactic domain S. The other, called **entailment**, and symbolized as '⊨' (i.e., «$\sigma_1, \sigma_2 \ldots \sigma_k \vDash \sigma'$»), corresponds to what follows, in the world or semantic domain, from the semantic interpretation-L of the premises— that is, to do with what must be the case, in semantic realm $\mathcal{D}$, given the semantic interpretations-L $\tau_1, \tau_2, \ldots \tau_k$ of $\sigma_1, \sigma_2, \ldots \sigma_k$. Or to put it more precisely, entailment corresponds to what follows, in the world, from other situations in the world— namely, from the semantic interpretations-L of the premises, as represented in the language of S. In terms of figure 5, entailment, like derivation, is a relation between items (sentences) in the upper half of the diagram, but, crucially, whereas derivation stays within the upper half, entailment reaches through the lower half.

In sum, to continue with this exercise of stating the obvious,

---

[4]This is not the place to consider constructivism, whether the property of being a man, or mortal, exists independently of anyone "registering" the world in such terms, as I would put it. What matters is that the moral towards which this chapter is driving holds just as crucially and resolutely under such metaphysical views as they do in the naïve realism view being assumed here.

[5]These are sometimes called their *formal* properties.

derivation has to do with what can be figured out, *syntactically*, from the premises; entailment, with what follows from them, *semantically*—i.e., what must be the case in the world, given their interpretations-L (or rather, again, more precisely, how one represents, in the language being used, what must be the case in the world, given the interpretations-L of the premises). The overall system is considered *formal* just in case the syntactic properties are independent of the semantic properties (independent of the interpretation-L) of any of the expressions involved.[6]

Nothing guarantees that derivation ($\vdash$) and entailment ($\vDash$) will align. On the contrary, as discussed more below, it is a normative condition on the design of a logic that it be formulated or set up in such a way *so that they align as much as possible*. The fact that it is possible to set a system up, at least in limited contexts, in such a way that this correlation holds, at least approximately, is at heart the fundamental insight of the logical tradition. Derivation is what is formally possible—something that an automatic effective procedure can do (or at least check). The derivation will be *correct*, however, only if what can be derived is also entailed. This is where the two norms considered earlier (§4.ⅵ), and so aptly summarized by John Etchemendy, come into play. A system is said to be **sound** just in case what can be derived is entailed. The rough idea is that, if something can be derived that is not entailed, then the system is *unsound*, and results of the inference procedure can no longer be trusted—the system has made a mistake. A system is said to be **complete** just in case everything that is entailed can be derived—a goal achievable in only the very simplest systems.

---

[6] Or, given σ_1, σ_2, and σ', one can check that σ' is in fact derivable from σ_1 and σ_2.

3 Framework

To facilitate later comparisons, and at the risk of stating what is obvious even more pedantically, this basic sketch of logic can be summarized in a scheme. I will say that a system of (formal) logic consists of six ingredients: two domains, one function, two relations, and a norm.

1. DOMAINS

   a. A **syntactic domain** S,[7] consisting of a set of expressions or formulae σ_i, typically written or constructed in a formal, recursively-specified, compositional language, where *formal*, as well as meaning precise, unambiguous, determinate, and a number of other things, is taken to imply that the syntactic properties of the expressions are defined without regards to their semantic interpretation-L (I, below); and

   b. A **semantic domain** $\mathcal{D}$, whose structure is subject to no a priori constraints whatsoever, though for the logic to be interesting $\mathcal{D}$ will usually have a structure relevant to the interpretation-L of the expressions in S. One common strategy is to take $\mathcal{D}$ to be a set of possible worlds, of which one—the so-called "standard interpretation-L"—is assumed to be the real (actual) world, and where worlds in general are taken to comprise objects, properties, and relations, plus facts and/or propositions, possibly states of affairs (objects exemplifying properties and standing in relations), etc.—i.e., to exemplify a configuration of what in *Promise* I called *formal ontology*.

2. FUNCTION: An interpretation-L function I, mapping elements of S onto elements of $\mathcal{D}$—i.e., onto their semantic values or "interpretations-L." (The expression

---

[7] The names S, $\mathcal{D}$, and I in this exposition are my own, as is the use of σ (σ_1, σ_2, σ_i, etc.); the symbols '⊢' and '⊨' are standard.

«*I(x)*» is commonly written «⟦*x*⟧».) Different kinds of expression (subtypes of *S*) are typically mapped onto different kinds of semantic interpretation-L (in *𝒟*), with *terms* being mapped onto *objects* (often called the term's *denotation* or *referent*);[8] *predicates* onto *properties*; and n-place *relation symbols* onto n-place *relations*. If σ is a *sentence*, then its interpretation-L *I(σ)* (or ⟦σ⟧) is usually taken to be either *truth* or *falsity*, the set of possible worlds in which that sentence is true, or sometimes the situation in the world which makes the claim true.[9]

Thus in our example, the expressions «MAN», «SOCRATES», and «MORTAL» would be stipulated to be elements of the syntactic domain *S;* and Socrates, the fifth-century BCE philosopher, plus the properties of being a man and being mortal (perhaps along with other people, facts about who exists, who is a man, who is mortal, etc.), to be elements of the semantic domain *𝒟*.[10] The interpretation-L function *I* would be set up so that the interpretation-L of «MAN» was the in-the-world property in *𝒟* of being a man, «MORTAL», the property of being mortal, and

---

[8]Chomsky once suggested that the linguistic category 'noun' was used for elements of natural language that were used to refer to objects—suggesting, that is, that in natural language the syntactic category *noun* derived its identity from its elements' semantic values. That would be illegal in formal logic; the identity conditions of any subcategory of expressions must themselves be syntactic.

[9]This is far too simple; see discussion point 1 below.

[10]In general, for any syntactic domain, there will be rules of composition, formulated via a recursive grammar over elements of *S* in virtue of their exemplification of syntactic properties, , specifying classes of complex expressions (and thus also elements of *S*) that may be formed from simpler ones. If our example was of a quantificational logic, the rules of composition would likely license (among other things) these three complex expressions, all classified as sentences: 'MAN(SOCRATES)', '∀x[MAN(*x*) ⊃ MORTAL(*x*)]', and 'MORTAL(SOCRATES).'

«SOCRATES», the once-living and breathing philosopher. The premises (σ_1 and σ_2) and conclusion (σ') in the classic inference would be sentences—symbolized, in quantificational logic, as «$\forall(x)[\text{MAN}(x) \supset \text{MORTAL}(x)]$», «$\text{MAN}(\text{SOCRATES})$», and «$\text{MORTAL}(\text{SOCRATES})$».)

3. RELATIONS

   a. A **derivability relation**, $\vdash$, instances of which hold between one or more expressions σ_1, σ_2 … σ_j of S and another element σ' of S.

   c. An **entailment relation**, $\vDash$, instances of which, like $\vdash$, hold of one or more expressions σ_1, σ_2 … σ_k of S and another element σ' of S, just in case, in the semantic domain $\mathcal{D}$, the interpretation-L $I(\sigma')$ is a *semantic consequence* of the interpretations-L $I(\sigma_1)$, $I(\sigma_2)$ … $I(\sigma_k)$.

A standard notion of semantic consequence is that $I(\sigma')$ be true in every possible world in which the interpretations$_L$ $I(\sigma_1)$, $I(\sigma_2)$ … $I(\sigma_k)$ are all true.

Derivability and entailment are typically defined over those elements of S that are sentences[11]—not over particular sentences, but over sentence *types*: that is, sentential composites in virtue of the exemplification of syntactic types of their constituents (with those types assumed to be intrinsic).[12] While

---

[11]It is not an absolute requirement that derivation and entailment be defined over sentences. For example, they could be defined over states of affairs (objects exemplifying properties and standing in relations, as for example denoted by such gerundial phrases as "my windbreaker's being torn"), in which case what it is entailed might be taken to be other situations that must be the case, if the "premise" state of affairs obtains, and what can be derived to be what can be figured out to be the case, from (the syntactic representation of) the premise.

[12]If the sentence «FEMALE(US-PRESIDENT(2032))» were my favourite sentence in some syntactic domain S_1, the interpretation-L function I, derivability relation «$\vdash$», and entailment relation «$\vDash$» might all apply to it, but none can do so in virtue of that sentence's being my favourite.

theoretically any old syntactic relation over sentences could be
called derivation, logic systems are set up (because of the ubiq-
uity of governing norms) so that derivation corresponds to
what, informally, can be "inferred" from them.[13]

4. NORM: A norm—or rather an interrelated system of
   norms—placing conditions on how the whole system is
   tied together.

Logic is a fundamentally normative enterprise in at least two
senses.

First, a variety of overarching background norms govern the
whole project. Logic is intended to be a model of thought, or of
the rational ideal to which thought should be held accountable,
or at least of the truth relations between and among sentences.
The aim of logic, that is, is to focus on sentences that are true—
to focus on *truth*—and to elucidate how true conclusions relate
to true premises. It is in order to deal appropriately with truth,
and with truth relations among sentences, that logic is

---

[13]From «P(x) ∧ Q(y),» for example, understood to symbolize the conjunc-
tion of I(P(x)) and I(Q(y)), the derivation rules will typically license the der-
ivation of either of the conjuncts, «P(x)» or «Q(y)», on its own (a derivation
rule called "and elimination"). They would similarly allow, from
«∀x[MORTAL(x)]», the derivation of any instance without the quantifier
(«∀…») in which the variable («x» or «y») is replaced by a name, such as
«MORTAL(SOCRATES)». And if «$\sigma_1 \wedge \sigma_2 \vdash \sigma_3$» is an expression, for sen-
tences σ_1, σ_2, and σ_3, it will generally be the case that that σ_3 can be for-
mally derived from σ_1 and σ_2.

Technically, once a given system of logic is established, then what can
be inferred, syntactically, from a sentence, is typically defined to be what-
ever «$\vdash$» maps it onto. My point here is that an intuitive notion of what
can be inferred, presumably based on an intuitive sense of what the sen-
tences mean or designate (i.e., with reference to an intuitive sense of their
interpretation) guides the way the syntax is defined, so that formal in-
ference can, as much as possible, mirror not only entailment but also that
prior intuitive sense.

fashioned in the way it is.

Logical inference is concerned with valid or worthy reasoning, with deriving true (not false) conclusions from true premises.[14] More generally, the discipline of logic represents an attempt to formalize *good* reasoning, or at least to articulate the conditions on truth relations among sentences that good reasoning should honour—reasoning we can trust, reasoning on which our understanding of the world can rely. A system of symbol manipulation in which sentences were not mapped onto anything resembling truth, or at least onto some normatively interpreted value or otherwise assigned some degree of worth—or even in which symbols were mapped onto "the true" and "the false" but in a haphazard way, of no consequence or relation to how the inference procedure worked—would have no claim to being a *logic* at all.

The normative dependence on truth, taken as a semantic notion, arises from a more general normative stance with respect to semantics that underlies all of logic. As I put it in *Promise*, and have said here earlier, it is a constitutive condition on anything warranting the name 'semantics' that the relation between sign, symbols, and other interpreted$_{\iota}$ entities and the entities, states of affairs, etc., towards which they are semantically oriented, be **deferential**. The world holds the cards, as regards what matters. It sets the standard on sentences and expressions; it establishes the conditions that hold on inference and reasoning. When words and world part company, the world wins; we must adjust our words to match. Yes, we can make things happen, through action; we can stipulate, for purposes of exegesis and hypothesis; we can order and promise and otherwise "do things with words."[15] All these things will come to the fore when we turn to computing. But everything that is done is done in a world that transcends the doer and the

---

[14]Deriving false sentences—or at least responses without caring about whether they are true or false—is generally trivial.
[15]Austin (1975).

doing.[16]

When logic, representation, and intentionality are in play, what it is to do anything, what can be done, what consequences arise from their being done, all arise not purely in the syntactic realm, but in the semantic domains towards which our words point, towards which the users of the words are oriented. It is thus not possible to understand what matters about logic except in terms of this overarching deferential stance. One does not understand logic, that is, if all that one thinks about is syntax and mechanism (proof theory). Even if one *imagines* that one is thinking only about syntax and mechanism, matters of semantic interpretation-L undergird that thinking, if it makes any sense whatsoever.

Logic is a normative endeavour in a second sense. In addition to the background deferential norm on sentences, inference is mandated to meet a foreground normative criterion that (i) derives from the (deferential) semantics, and (ii) applies to the formal or syntactic operations. In the sentential case, it is most easily stated as that truths should lead to truths; more generally, it is that the outputs of the formal derivation should semantically follow from the inputs. In the sentential case, in particular, in any situation in which the inputs σ_i are true (i.e., $I(\sigma_i) = true$, as it is sometimes put, or $[\![\sigma_i]\!] = true$), then the output σ' should be true as well ($I(\sigma') = true$, $[\![\sigma']\!] = true$). Or slightly more generally, as it is normally stated: an output σ' should be derivable from inputs σ_i *just in case in all situations*

---

[16]It may be argued that in performative cases, as for example in the matrimonial "I do," that the state of the world defers to the action undertaken, rather than the other way around. But even in such cases, the *overall* nature of the situation remains deferential; who it is that has promised, for example, and to whom the promise has been made, have their identity in the world in which, and towards which, the performative action is taken.

(worlds) when the inputs σ$_i$ are true, the output σ′ is true.[17]

That is, the aim of developing a logic system is to define the system and its inference rules so that *what is derivable is entailed*—i.e., such that what can be formally produced semantically follows from the inputs (technically: where $\underline{\sigma_i} \vDash \sigma′$ obtains in every case where $\underline{\sigma_i} \vdash \sigma′$). It is when this criterion is met— when what is derivable is entailed—that, as stated above, the system is said to be **sound**. A logical system is worthwhile only if it is sound. Unsound systems, in the context of logic, are to be discarded.

These normative criteria, foreground and background, arise from the most important fact about intentional (semantic-L or interpreted-L) systems: they are *oriented to the world*. To use Brentano's metaphor,[18] intentional systems point to the world, manifesting an "arrow of directedness." That is not to say that the logical system, *per se*, says or places restrictions on the expressions' semantic values or interpretations-L. In a way that will matter when we come to talk about computing, that role is supplied, in the case of logic, by the interpretation function—a function that a computer scientist might call a "user-supplied parameter." What matters about logic, about semantical systems in general, and about computing, derives not, or anyway not solely, from the local, causal, behavioural, effective consequences of the system, but from the wider context: from the extent to which those local, causal, observable behaviours honour the state or configuration of the world towards which they are semantically directed.

Overall, as I have been putting it, although all logical, semantical, and computational systems must "work," syntactically or causally, in virtue of material or effective properties, they are **normatively governed** by facts deriving from their semantic

---

[17]MORTAL(SOCRATES) is considered to be entailed by the premises '∀(X)[MAN(X) ⊃ MORTAL(X)]' and 'MAN(SOCRATES)' because in all worlds in which Socrates is a man and all men are mortal, Socrates will be mortal.
[18]Or at least Brentano as interpreted by Chisholm (1957, p. 168). Also see McAllister (1974, pp. 328–338) and Dupuy (1994, p. 105).

interpretation-L.[19]

A couple of quick points of discussion.

1. In any realistic system, the logical structure would be far more complex than suggested above. The interpretation-L relation *I*, for example, is usually taken to be at least two-stage, with a first stage mapping expressions onto something like their *intension*, *meaning*, or *sense*; and a second stage mapping that first-stage value onto the expressions' *extensions*, *referents*, or *denotations*. It is the latter that I have, rather informally, gathered under the term '*interpretations-L.*' This double-stage strategy allows logics to deal with belief and other so-called "non-transparent" contexts, in which the meaning, but not the interpretation-L, is relevant to the truth of an enclosing sentence—such as "Robin did not believe that 17 is the square root of 289," even though the interpretation-L of "17," the number seventeen, is identical to the interpretation-L of "the square root of 289."

   Some of these complexities will come to the fore later, especially in chapter 6; but we need not be concerned with them here.

2. The word 'formal' is sometimes used in place of 'syntactic'—for example, to refer to the *formal* properties of expressions—but I believe this is a conceptual mistake. 'Formal' can be coherently understood as a predicate on any element of a logical framework only with reference to the whole structure—including both the syntactic and semantic aspects. To say that inference is formal may imply that *derivation* is defined in terms of the syntactic properties of the premises and conclusion, but that only makes sense against a background assumption

---

[19]Cf. *The Promise of Artificial Intelligence* (2019), p. 11, footnote 11.

of semantic interpretation-L. Transforming one set of expressions to another without any implicit reference to a background semantical interpretation-L has no claim on being *logical inference*. As I put it in §4.vi, such a regime would be better called *stuff manipulation*.

3. I have focused on the requirements on a logic for it to be called formal not only because it is typically assumed, nowadays, that if one is talking about logic one is talking about formal logic, but also because it is universally assumed that anything logic-like or inferential done by a computer must perforce be formal, and because it was systems of formal logic that gave rise to computation as we have come to understand it. I disagree that computation must be formal in this sense, but the chief insight I want to unearth here from the logical tradition is easiest to appreciate in the formal case, so in this chapter I assume, throughout, that we are talking about a formal logical system.

4 Effectiveness

I assumed, in the development of 2/3Lisp, that the overall nature of logic, as roughly sketched above, and independent of the details of any particular logical system, could be assumed as part of the background intellectual context within which computer science operated. The papers describing the 3Lisp model of reflection assumed it needed neither introduction nor explanation. It soon became evident, however, that was an illegitimate assumption, especially in computer science.

Part of the problem stems from overlapping terminology that I have talked about since the beginning (different conceptions of *program*, for example, and distinctions between interpretation-C and interpretation-L). As noted, though, the miscommunication had a deeper foundation. To bring it into the open, it helps to address the following question:

EFFECTIVENESS: Which of the five elements of a logic— the syntactic and semantic domains (S and $\mathcal{D}$), the

interpretation-L function (I), and the derivability and entailment relations ($\vdash$ and $\vDash$)—are assumed, each in its appropriate way, to be subject to a criterion of *effectiveness*, in the sense of 'effective' that underpins the notion of effective computability, as discussed in §4.v2?

Formulating this question precisely takes some technical work, but the issue can be put into plain English. Which of the following operations are mandated, in logic, to be *effective* or *mechanical*—i.e., to be capable of engendering causal (electronic, mechanical, etc.) consequences?

P1 [SYNTAX] Responding to expressions in virtue of their *syntactic* properties (S);

P2 [SEMANTICS] Responding to expressions in virtue of their *semantic* properties ($\mathcal{D}$);

P3 [INTERPRETATION-L] Mapping expressions onto their *semantic values* or interpretations$_L$ (I);

P4 [DERIVATION] Mapping expressions onto *other expressions formally derivable from them* ($\vdash$); and

P5 [ENTAILMENT] Mapping expressions onto other expressions that they *entail* ($\vDash$).

To address these questions, we need to move from pure logic conceived as a mathematical or abstract structure (in which the above relations simply hold; there is no "active process of mapping") to something closer to a "logical machine." That is, we need to address the **mechanization of logic**.

As suggested earlier, the project of mechanizing logic was not strictly speaking part of the formalization efforts per se, initiated in the mid to late 19th century (by Boole, Peirce, and Frege, among others), and more fully developed in the first half of the 20th (by Russell and Whitehead, Hilbert, and others). It is difficult, today, to appreciate how stunning it was, in those decades, to see that something like reason or rational inference could be exhibited in a "mere machine." It hardly needs to be

said that the development of such mechanisms, among which Turing's 1936/7 paper must be counted, had enormous implications not only within logic and the philosophy of mathematics but for general philosophy and intellectual history more generally. The entire development of computing can be seen as part of this history.

Crucially, however—a point that in some sense it is the overall brief of this chapter to demonstrate—what the mechanization of logic meant, historically, and how computing is currently understood, part company. The five questions listed above allow us to get at that divergence. On the logic side, I will address them from the point of view of a logical inference machine or theorem prover, designed in accord with the five-part framework.

From a default realist position, I take it that in the logical case the answers would be taken to be unambiguous:

P1 *Syntax* must be effective, in the sense that the syntactic properties exemplified by expressions in S must be effective properties. That is: for any expression x in S, it must be effectively determinable whether or not x exemplifies any particular syntactic property S_j—it must be possible, that is, to construct a mechanism to turn on or off a switch, depending on whether x does or does not exemplify S_j.

P4 *Derivability* ($\vdash$) must be effective, in the sense that those expressions derivable from a given expression x in S must be effectively enumerable—or at least, for any other expression y, it must be effectively decidable whether y can be derived from x.

These two answers can be understood in terms of the basic picture given in figure 5: the syntactic realm or syntactic "level of abstraction"—the upper half of the diagram—is the one that is subject to causal or mechanical constraint. If one were to define a "logic" that violated either of these two principles, it would thereby be evacuated of intellectual substance and all

theoretical interest—would be thereby deserving of ridicule. And note that the vacuity of violating either of them holds true of logic in general, not just of logic in the context of mechanization. Suppose, for example—since systems of formal logics are human constructions—that one were to stipulate "true in the standard interpretation-L" to be a *syntactic* property of arithmetic sentences, and were similarly to stipulate, as a rule of logical inference, that β could be derived from α (i.e., that α ⊢ β) just in case β is true in any world in which α is true. Then, contrary to Gödel, common sense, and all that is right and good, one would thereby have generated a sound and complete axiomatization of arithmetic. That is, to put it bluntly, in the development of a logic, to violate either of the above two criteria is to *cheat.*

On the other hand—and this is what matters about logic in general, about the 2Lisp and 3Lisp architectures described in this book, about computation more generally, and about my fundamental opposition to a totalizing mechanism—*there is no reason whatsoever that any of the other three elements be required to be effective, be restricted to the realm of mechanism:*

P2   The semantic domain $\mathcal{D}$ itself (i.e., the exemplification of properties by elements in $\mathcal{D}$ denoted by expressions in S);

P3   The interpretation-L function I; or

P5   The entailment relation ⊨.

On the contrary, it would be the height of perversity—intellectual folly—to suggest that any of the last of these three entities was subject to effectiveness constraints. Formal logic, and the wider mechanistic approach to reason that it unleashed, is undeniably significant and powerful, but it is also strikingly and famously limited. It is important and interesting, moreover, precisely because of the interplay of both factors: the source of the power and the reason for the limitation. And the importance and power, on the one hand, and the limitation and

constraint, on the other, arise, metaphysically, from the same fact: by restricting oneself to the effective or mechanical, one can achieve extraordinary results as regards reasoning and proof about a much, much wider world—specifically, a world that is *not so restricted*.

It was this recognition—that a purely effective mechanism could exhibit the behaviour of rational deliberation—that was of epochal significance to philosophy, psychology, and theories of mind in the first half of the 20th century. It was also the insight that led Haugeland to formulate his famous "formalists' motto":

> Obey the formal rules of arithmetic…and your answers are sure to be true. This is the deep, essential reason why interpreted formal systems are interesting and important at all. Hence it deserves a memorable name and phrase:

> FORMALISTS' MOTTO: If you take care of the syntax, the semantics will take care of itself.[20]

Haugeland's point was never that syntax and formal operations would *generate* the semantic interpretation, as he is sometimes—and unfortunately—misinterpreted as saying. On the contrary, his assumption was that, with the semantical framework in place, then the results of the formal operations would be governed by the same semantical framework, and thereby automatically receive their appropriate interpretation-L.

5 Representation

Mechanizing logic, in sum, did not mean mechanizing the entire subject matter of logic. That would have been lunacy, robbing the result of any warrant to being called logic at all. Rather, it was the role of semantics and interpretation-L—the crucial ingredients in P2, P3, and P5—to relate the mechanically restricted realm to the wider, non-restricted one. Ultimately, in fact, I believe that that is what semantics, and the intentional in

---

[20]Haugeland (1985).

general, is *for*.

We humans, along with any other physically possible sys-
tem, including those things we call computers, are stupefyingly
hobbled in virtue of being physically embodied. Our nerves and
systems can only respond to what is locally proximate, what im-
pinges on us at our immediate laminar surface. I cannot even
detect you, standing there across the room, *directly*. If I am
looking at you, all that presses in on me, in a strictly physical or
mechanical (effective) sense, are waves of electromagnetic radi-
ation that have reflected off your laminar surface, and are trans-
iting my eyeballs and generating patterns of electrical energy on
the cells in my retina. The stunning achievement of my brain
and body, along with long-evolved configurations of environ-
ment and culture, and increasingly the impressive capacity of
the machines we build, is to use these proximal disturbances to
set up *semantic* (non-causal) relations to that which is beyond
my laminar surface—i.e., to *you*, in the present example, but
also to home and to loved ones, to yesterday and tomorrow, to
the world we inhabit, to the distant stars, and indeed to all we
know of and contemplate and imagine.

This tension between the radically limited capabilities of
what is locally effective and the unlimited reach of all that exists
beyond its confines establishes a fundamental dialectic under-
lying all semantic and intentional systems (including logic,
mind, and computing)—something I call the **intentional di-
alectic**. The dialectic, which lies at the heart of the relationship
between meaning and mechanism, poses a profound challenge
for intentionality. All systems directed towards the wider
world—systems that think and represent and believe and com-
pute—in virtue of their concrete physical embodiment, are lim-
ited by the physical nature of the world. All such systems can
do, therefore, in any active and consequential sense, is to acti-
vate, exploit, and adjust their proximal physical organizations
and structures. From a mechanical point of view, that is, all they
can do is to reckon that which is local and effective. But what

matters about them has to do with their relation to the phenomena and entities and states of affairs towards which they are intentionally directed—their relation to what is distal and effectively inaccessible.

This intentional dialectic places all intentional systems under the scope of what I have elsewhere called the Representational Mandate:[21]

> REPRESENTATIONAL MANDATE: The proper functioning of any world-directed system—any system that is thinking about or representing or processing information about the world—must be governed by normative criteria applying to its mechanical operations that are framed in terms of situations and states of affairs in the world that the system represents, reasons about, or is otherwise intentionally directed towards—which situations and states of affairs will not, in the general case, be within effective (causal) reach.

The treasure at the heart of logic—the reason I have taken the time to explicate its structure in such detail—is that logical systems illustrate, in an extremely limited domain, a remarkably general way of meeting the mandate's challenge. Stripped of particularities, the solution is something I will call the **core intentional architecture**. It can be explained by unpacking the mandate into an statement of the fundamental dialectic, followed by a statement of the architectural approach required to meet it.

CORE INTENTIONAL ARCHITECTURE

Conditions (DIALECTIC)

C1. An intentional system must work, locally and effectively, in virtue of the effective properties of its concrete physical embodiment.

---

[21]See *The Promise of Artificial Intelligence* (2019), chapter 2, p. 17.

C2. Overall, it is normatively directed towards the world as a whole, including much that is not effectively available (including what is distal);

C3. Being neither oracle nor angel, it has no divine (magical) access to those non-effectively-available states that it cares about.[22]

So what does it do? (ARCHITECTURE)

C4. It exploits local, effective properties that it *can use, but does not (intrinsically) care about*—including both: (i) interior effective properties (such as the configurations of its internal mechanical structures), and (ii) interactions with local, effectively available aspects of its environment, including effective configurations of proximal entities

C5. To "stand in for" or "serve in place of" properties and relations of state of affairs it cannot be effectively coupled to, in order to

C6. Behave appropriately towards those remote or distal (non-effective) states that it *does care about, but cannot use.*

C1 is an acknowledgement of overarching physicalism, and a recognition of the extraordinarily restrictive constraints of effective causality. C2 summarizes the defining property of intentionality and intentional systems. C3 articulates the intentional dialectic that follows from these two. C4–C6 are effectively a definition of the nature and function of *representation*: any

---

[22]"That it cares about" is informal. On their own, logical inference systems care about nothing. What is meant is that these states—or rather, the semantic relations linking the systems to those states—are what the users of the system care about. To put it in terms of the framework: these states figure constitutively in the governing norms.

available effective configuration whose normatively sanctioned role is to enable the system in or for which it plays a role to be intentionally directed towards its subject matter. C4 is framed so as to include both internal representations (memories, data structures, configurations of ingredients, etc.) and external representations (maps, signs, images, external language, and the like).

In logic, the upper half of figure 5—syntax and proof-theoretic operations—are the local effective properties referred to in C4. "Behaving appropriately" (C6), with respect to the reasoning processes or truth-relevant relations modeled by logic, is defined in terms of the lower half: coming up with true conclusions from true premises, where truth is defined in terms of the interpretation-L relation to the semantic domain $\mathcal{D}$.

A possibly helpful analogy. Imagine the expressions of logic as statues, in the shapes of letters and words, perched on a cliff at the edge of the sea. Each of the words points out into the darkness, towards distal situations far beyond any immediate access. Overall, the words are arranged so as to form sentences describing those distal worlds and continents. Inference rules are like (instructions to) functionaries who run around at the top of the cliff, inspecting expressions, moving them around, constructing new ones, etc. The norms on both the expressions and the functionaries' movements are governed by facts about the distant, inaccessible situations towards which the expressions inaccessibly point.

What is *present*, one might say, are the expressions and the functionaries' effective operations conducted on them. What is *absent* is the world, the semantic domain, towards which such structures are directed. What matters about expressions— what matters about language and logic and mind and computing—is that the sentences be *true*. *Au fond*, that is, what matters is that *the present stands witness to the absent*.

Some things, such as rocks and electrons, are merely present. Some things, such as the tress and the grass, carry information about the absent, but they themselves do not point

towards it, and are not normatively accountable to it. What distinguishes intentional phenomena and entities is that they do so point, and are normatively governed by them, via those far-reaching semantic relations.

6 Discussion

A few points of discussion.

1. NATURALISM: Note that it poses no challenge to an overarching physicalist worldview to admit that although the syntactic and proof-theoretic realms in logic (S and $\vdash$) are required to be subject to constraints of mechanical effectiveness, none of the semantic entities ($\mathcal{D}$, I, and $\vDash$) are so restricted. The bounds of local effectiveness are more stringent than the bounds of overall metaphysical coherence. Relational properties show this immediately: being right now a little more than four light years from Alpha Centauri poses no threat to physicalism, but in no practical sense is it an effective property.

2. REPRESENTATION: Needless to say, the notion of representation implicated in the characterizations given above is extremely broad. Representationalism is widely decried, in contemporary cognitive science and philosophy of mind—viewed as excessively focused on truth, incompatible with contextual dependence, dependent on a sharp subject/object divide, and guilty of a variety of other epistemic ills. But nothing in the mandate, as stated, imposes any such restrictions; it is compatible with embodied, enactive, constructive, pragmatist, and other ontological and epistemic stances. The fundamental insight about representation, to which many of these contemporary movements are blind, is something that any account of intentionality must acknowledge. The straightjacket imposed by the intentional dialectic

is profound. Only by exploiting available degrees of free-dom in the local and effective is it possible for any phys-ically embodied system to be appropriately oriented to that which is beyond its causal reach. That blunt meta-physical truth cannot be obscured by any convenient metaphysical worldview.

3. EXTERNALISM: Philosophical readers may expect the term *externalism* to be used to name the semantic ap-proach that I claim underlies our traditional under-standing of logic. But that term is misleading—espe-cially in the case of reflection. Externalism, as normally understood in philosophy, has to do with the role of the "external world" (i.e., the role of elements of 𝒟) in the determination of the *meaning* or *content* of a sentence or thought—not with the externality of its reference or logical interpretation-L. All but a solipsist philosopher would take reference or logical interpretation to be ex-ternal in this sense. The restriction to a totalizing mech-anism of the sort to be described in the next chapter proscribes even referential externalism of this sort.

7 Summary

Formal logics form a radically narrow, limited subset of the range of possible intentional systems. My aim in this chapter has not been to endorse logic—either as a model of human cog-nition, or as a basis for a theory of computing. Computational systems routinely deal with a wide range of issues—dynamics, context sensitivity, action, input and output, communication, and a myriad other complexities, all of which lie beyond the boundaries of anything considered to be within the scope of a "system of logic." Any realistic model of thinking will similarly need to deal with context, time and process, relations to activity and perception, evidence, subject matter, motivation, hypothe-ses, narratives, and a host of other things. Neither do I endorse logic's strict separation between the realm of effective activity (syntactic and proof theory) and the semantic realm, and in general its traditional reliance on a classical, formal conception

of ontology. As suggested in *Origin*, I believe our commonsense ontology arises out of an inexorable pattern of participatory practices—practices in which, as I put it there, we "register" the world, find it intelligible, in ways that are arise out of our projects, purposes, and practices.

What matters about logic, for our purposes here, and about theorem provers and other systems based on logic, is merely a single and focused, but unutterably important, fact: their demonstration of the core intentional architecture—of how a concrete, mechanically implementable system can be normatively governed by a system of deferential semantics. Though the details of specific formal logical systems have been worked out in highly restricted, formalized contexts, the basic model that underlies them, of a concrete system normatively governed by its semantic relations to the world, is an insight of unsurpassed power. This core architecture is a general condition, applicable to the entire range of intentional systems.

7 Blanket Mechanism
Diagnosis · Third Pass

What then of computation? Is computer science, as it is practiced today, a study of the realization, in systems of great complexity, of the core intentional architecture described in chapter 6?

As should be evident by now, the answer is *no*. But it is a complex *no*. Computation in the wild, as I have argued, *is* genuinely intentional. Real-world programs and processes are in fact instances of the core intentional architecture. I believe these truths are tacitly understood by all programmers. But computer science does not study programs that way. As currently formulated, computer science disappears all the non-effective relations. It thereby disappears the true nature of semantics, and the normative constraints governing programs. Contemporary theoretical computer science operates within the grip of what I call **blanket mechanism**: a fusion of:

1. A substantive belief that the phenomenon of computing is exhaustively constituted by issues of effectiveness— by "what happens," by the structure and behaviour of machines, abstractly or concretely construed, where 'behaviour' is restricted to what happens, causally, within the effective boundaries of the machine; and

2. A corresponding methodological tenet that causal and/or mechanistic explanation, even if analyzed mathematically, is sufficient to explain all of computing's theoretically relevant aspects.

The combination of substantive belief and methodological tenet leads to an overarching assumption that all theoretical discourse about computing should be restricted to the mechanical—to mechanical entities and effective relations among them. Embrace of the substantive belief reflects what is taken to be *true*. Allegiance to the methodological tenet constrains what alternative ideas are treated as *legitimate*. Suggestions running counter to its precepts are not so much thought false as "ruled out of court"—making it difficult to raise the substantive belief into view, so that its adequacy can be questioned.

The aim of this chapter is to unpack blanket mechanism— to understand what it says, how it arose, what it includes and excludes. and how it affects computer science's treatment of the subjects of this book: programs, programming languages, and semantics—and how they come together in reflection.

As we have seen, computer science does not ignore semantic phenomena altogether. Analysis would be much simpler if it did; the alleged mechanical character of computing would be evident, intentional discussion would be obviated, and computing would collapse into stuff manipulation (abstract Meccano). But the enduring use of semantical vocabulary betrays that this is not the field's strategy. Rather, several tactics are employed to move non-effective semantic relations off-stage. Sometimes they are simply assumed—absorbed into the background, without being made theoretically visible, as in the universally recognized but untheorized use of (radix-based) numerals to denote numbers. More generally, in order to honour blanket mechanism's strictures, non-effective semantic relations are projected onto the "nearest available" effective proxy lying within the machine's mechanical boundary. Terms classically used to describe semantical$_L$ phenomena, such as *reference* and *meaning*, are redefined—folded back within the system, and used to refer to whatever local, effective structure is selected to serve as their proxy.

This "projection strategy" works in two ways. In the first, when the true semantic-L relation, though itself non-effective,

would nevertheless point at an effective internal structure, an effective *access* relation to that location serves as a proxy for the non-effective semantic relation—providing an effective connection between the same relata. For example, while a programmer will likely think of an address as a *reference* to an internal memory location, it will be implemented so that its use leads a process to be able to directly *access* the location to which it points. Theoretical analysis can then mirror the implementation.

In the case of binary addresses the difference between reference and access may seem small. In a higher-level program, though, a programmer is likely to understand program identifiers, such as «LIST-OF-FLOORS» or «CURRENT-USER», as references to real-world objects—the objects relevant to norms governing the program's design and behaviour. But theoretical analysis again takes the semantic values of these identifiers to be *that to which they provide effective access*—internal memory locations. In this second case, access and genuine reference part company. And once again theory mirrors implementation.

When a semantic-L relation is directed towards something not effectively accessible, that is, an appropriate accessible internal object is pressed into service as a proxy. That then allows an effective access relation to that proxy object to stand in for the non-effective relation to the original (abstract or distal) target. These moves are hidden from theoretical view by the fact that the appropriate semantical terms—*reference, denotation, semantics*, etc.—are redefined to refer to those proxies, instead of to the original entity.

An analogy may help. Imagine semantical-L relations as beams of light, emanating from the meaningful structures and behaviors of an intentional system, aimed at their referents—i.e., aimed towards their interpretations-L in semantic domain $\mathcal{D}$ in the wider world that the system is about.[1] In terms of this

---

[1]Needless to say, the analogy is fatally flawed. Genuine reference is staggeringly more powerful than light. It is able to reach through solid

image, theoretical computer science can be imagined as having placed mirrors at the boundaries of the mechanism, reflecting all emanating light beams back inside the machine, so that they land on some other causally-individuated internal ingredient. All semantical-L vocabulary that would have referred to the things that the light used to illuminate, and to the relations that the system bore to them—terms such as 'interpretation', 'reference,' etc.—are then redefined so as now to refer to the mechanically individuated proximal ingredients. The result allows the external world to be removed from theoretical view, and theoretical analysis to proceed as Newell originally wanted: "wholly within the machine."

Crucially, however, though the non-effective relations and external objects are no longer theorized, they continue to undergird the normative conditions that the computational systems are designed to honour. As in all intentional architectures, the normative conditions arise from the realms that the systems are "about." But since the official semantical story no longer makes reference to the external world, semantics ceases to be a "topic" in which normative issues (analogous to soundness or correctness in logic) can be discussed.

The projection strategy props up computer science's apparent naturalistic credentials. By treating its subject matter entirely mechanistically, the field is construed to fit within an overarching mechanist philosophy, and can thus take its place within reigning conceptions of science. But these merits are achieved at a price. In fact the strategy is something of a cheat. One consequence, just mentioned, is that it disappears the normative considerations applicable to computation—considerations which in logic are kept within explicit theoretical view. Another consequence is that, for reasons explored in this chapter, the strategy almost inexorably leads to a conflation of the two

---

objects, outside the light cone, backwards to the past and forwards to the future, to non-actual and hypothetical worlds as well as the real one, without fading, losing precision, or taking time.

dimensions of overall significance distinguished in the 2/3Lisp architecture: (i) procedural consequence, ψ, having to do with the effective or causal treatment of a signifying structure, and (ii) declarative or referential import, φ, having to do with what that structure represents, describes, or is intentionally about.

Understanding reflection requires distinguishing these two aspects (ψ and φ): having a clear sense of how they relate, how they differ, what normative conditions accrue to each, and how to integrate them into a single encompassing theoretical framework that does justice to both, including to their interaction. As noted in §4.v4, this is the fundamental idea undergirding the notion of "soundness" embodied in the 3Lisp mantra: that implementing reflection is simple on a *semantically sound base*. If the two aspects are conflated, the base is thereby rendered unsound, reflection cannot be coherently described, and clean reflective architectures are precluded.

Unpacking blanket mechanism will allow us to explain how computer science is blinded to these facts, and to explain many perversities we have encountered en route. It will also reveal both what is right and what is wrong about the first and second diagnoses.

1 Preparation
Four preliminary comments.

1a Mathematics
As noted throughout the book, first, much theoretical analysis in computer science is conducted in mathematical terms. Per se, that is a methodological statement, about the *analysis* of computing. The deeper issue is ontological: (i) whether computer science uses mathematics to analyse, perhaps at a relatively high level of abstraction, what is fundamentally a concrete or mechanically constituted phenomenon; or (ii) whether computation itself is fundamentally abstract, along the lines of prime numbers and Abelian groups.

The following argument is often cited in favour of the second, abstract reading: one can start by enumerating simple

mathematical objects—numbers, elementary functions over them (successor, addition, functional composition, etc.)—and from that basis build up to computability results, complexity classes, and various other fundamental results in theoretical computing. But this argument shares a difficulty with ostensive definitions: it starts with a stipulated basis for the construction (called the "initial functions" in recursion theory), without explaining the warrant for that particular choice. Mathematical functions are by and large understood extensionally: as sets of pairs of numbers $<x_i, y_i>$, where in each case $f(x_i) = y_i$. Why should the successor function $f(x_i) = x_i + 1$ be included in the base, for example, but not $f(x_i) = 0$ or 1 depending on whether some universal machine $\mathcal{M}$ halts when given a numeral designating x_i as its input? "Because the successor function is easy to calculate," someone might reply. But that is circular; analysis would show the halting predicate easy to calculate if it were taken to be *primitive*. Successor is assumed to be simple because we can calculate it easily using standard numerals. But standard numerals are abstractions of concrete arrangements—particular representational arrangements to which we are historically committed.[2]

Think too of Turing's deliberations on the primitive steps taken by a Turing machine: reading or writing a simple token on that square of tape that is immediately in front of or adjacent to the controller, moving the tape one square left and right, etc. These operations were designed to be mechanically simple; there is no warrant for choosing them if one is merely concerned with abstract mathematics. Think too about how complexity results are expressed in terms of "space" and "time." Sure enough, these are rather abstract conceptions of space and time, but they are nevertheless abstractions over the notions of concrete space and time derived from physical reality. This is betrayed by the fact that, in order to implement a computation,

---

[2]As opposed to, say, using a base-π representation, or scheme with a radix consisting (from the right) of each successively larger prime number.

computational time must be realized by real time. A paper chart that "implemented" time spatially, as for example in a log or "dribble" file, would not count as an *implementation* of a computation—a limitation that would be inexplicable if computation were purely abstract. Effectiveness, too, about which more below, can only be defined with respect to some notion of mechanism or machine.

More can be said, but since issues of mechanism and efficacy etc., are in focus, I will henceforth assume that computation must ultimately be grounded in some notion of machine or mechanism, at an appropriate level of abstraction—perhaps above specific concrete physical details, but not at such a high level of abstraction as to have entirely taken leave of the constraints of physical realization. Whether a purely mechanical account is *sufficient* in terms of which to define computing is the topic of this chapter; I will henceforth assume that a mechanical grounding is *necessary*.

1b Semantics

Second, I have talked throughout about computer science's use of semantical vocabulary. I noted in §1.6 that many of the field's technical terms derive from the "rationalist" rather than empiricist side of the Scientific Revolution—from the "meaning" half of the meaning/mechanism dialectic. I have repeatedly claimed that computation-in-the-wild is a genuinely intentional or semantic phenomenon. In chapter 4, when discussing V3, I argued that programmers not only *do* have, but *must* have, a tacit understanding of the non-effective semantic relations that tie the structures and behaviours of their programs to objects and states of affairs in their task domains. And I have emphasized throughout that these semantic relations involve non-effective, and hence non-mechanical, relations.

One might have expected this constellation of facts long since to have defeated the blanket mechanist approach to computing. It would have done so, I believe, were it not for the combination of two things: the strength of computer science's allegiance to the methodological tenet described above, and the

subtlety of the projection strategy. Because computer science has been able to retain semantical vocabulary by using traditional terms with new meanings (meanings tailor-made for projection), its inability to do justice to what I consider to be the genuinely semantical nature of computing has been hidden from theoretical view.

1c Compute

The word 'compute', third, on which the entire field of computer science rests, is curiously ambiguous. It is a transitive verb, but there is no theoretical clarity on what exactly it is that can be computed.[3]

Suppose I tell you "I picked up an old Servel at a flea market." If asked to report on our interaction, you could truthfully say both (i) that I uttered a statement and (ii) that I described a refrigerator. It would be false to say that I described a statement, and malformed to say that I uttered a refrigerator. In ordinary language, even when the distinction is not explicitly marked, we by and large keep semantic levels distinct.[4] 'Utter' and 'describe' are unambiguous with respect to whether their direct objects are expressions themselves, or those expressions' semantic-L interpretation.[5]

Not so with the verb 'compute.' If I claim that my laptop "computed the prime factors of 5,083," it is ambiguous, and would likely not be agreed by practicioners, whether, by analogy with 'utter,' I am saying that my laptop produced the *numerals* '13', '17', and '23', or whether, by analogy with 'describe,' that it computed the *numbers* thirteen, seventeen, and twenty-

---

[3]'Compute' can be used intransitively, but I take it that its meaning is derivative on the transitive sense.

[4]We do not always *mark* the distinction, but context is usually enough to disambiguate, making the sense clear.

[5]Needless to say, I can describe expressions, but at a level of semantic remove. If I say "the word 'facetiously' uses all six vowels in alphabetical order," I will have *mentioned* the word 'facetiously,' as philosophers of language would put it, but not *used* it. Hence the quotation marks.

three. The latter claim could be justified by the fact that the machine produced numeral representations of those numbers, just as the claim that I described a refrigerator can be justified by the fact that I used a word ('Servel') that names a gas refrigerator.

To some extent the usage differences with respect to the term 'compute' are masked by the fact that in simple arithmetic cases involving standard numerals, sign and signified are closely related and less evidently distinct than in concrete cases of everyday artifacts. But the ambiguity runs deep. In the 1980s, when walking to lunch at PARC with some fellow computer scientists, a colleague commented, as we passed a fluttering birch tree, "I wonder how much computational power it takes to compute that tree." The thought made no sense to me. I took computation to be symbol manipulation; I could produce a symbolic representation of the tree with no computational power at all: just print out the words "that birch tree." To my colleagues, I take it, to "compute the tree" meant something like "simulate the tree accurately enough for the simulated leaves to simulate fluttering." What was striking was not that my companion's question was uninterpretable. It was that we were colleagues on a *knowledge representation* project. I was taken aback by the fact that our views could differ so radically on the foundational notions of our field, without that difference ever having surfaced in our work.

This is not the place to resolve the ambiguity. In what follows, however, it will be crucial to be rigorous, at each point, about what is being assumed about exactly what is being computed.

1d Efficacy Predicates

Fourth and finally, another terminological issue involves the other word in the core phrase *effective computability*.

As has been evident here since the beginning, what it is to be effective is fundamental to computing. All theories of computation—semantic or mechanist, concrete or abstract—must refer, implicitly or explicitly, to effective properties: those

E_8· Computational
E_7· Syntactic $\big\}$ **Intentional**
E_6· Formal
E_5· Effective $\big\}$ **Functional**
E_4· Mechanical
E_3· Intrinsic
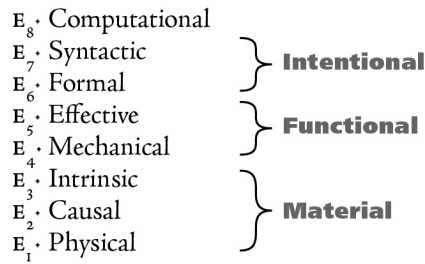E_2· Causal $\big\}$ **Material**
E_1· Physical

Figure 6 — Efficacy Predicates

properties of machine states in virtue of the exemplification of which things can happen, things can be accomplished. A claim that machine M transitions from state σ_0 to state σ_1 given input α, for example, assumes that there is a property π of M being's in state σ_0, and a property π' of α's being present, such that in combination they can "effectively lead" M to enter state σ_1. The very words "in the presence of α" reflects just such an assumption.[6] If there were no such πs—if there were no constraints that M's state and the input α must be effective—then everything would be computable, notions of computation, computability, computational complexity, etc., would be vacuous, and the field would collapse.[7]

This much is evident. What is striking is how many different terms are used to get at these properties. Overall, I will call them **efficacy predicates**.[8] By an *efficacy predicate*, that is, I mean a term used to name the higher-order property exemplified by those first-order properties of computational structures and states in virtue of which computational activity can

---

[6]Technically, one could define a machine M, states σ and σ', and token α, and stipulate that M would transition from σ to σ' just in case α was at that moment being thought about by the Emperor of Japan, but to do so would be "illegal." It is the conditions on legality I am concerned to uncover.

[7]Turing's demonstration that some functions cannot be computed depends on two facts: (i) that such functions, such as the halting function, are well-defined and metaphysically first-class; and (ii) that those functions are nevertheless noncomputable *because of the limitations of what can be done effectively*.

[8]I refer to predicates—i.e., to linguistic items—rather than to the properties they denote, because I do not wish to make or imply any claim about whether distinct efficacy predicates denote different (higher-order) properties.

proceed—change can happen, behavior can occur, concrete activity can take place.

Figure 6 lists eight terms sometimes used as efficacy predicates. Six deserve only brief mention; two will require more discussion.

The use of 'computational' as an efficacy predicate, to start at the top, betrays the grip of blanket mechanism. One could not say, of a logical system of derivation, that inference proceeds in virtue of the *logical* properties of the premises, because "logical" is understood to include both syntactic and semantic aspects. To use 'computational' to name just those properties in virtue of which machines operate is thus to deny (or at least to dismiss the idea) that any other properties—in particular, any non-effective semantic properties—are constitutive of something's being a computation. Even if that were true, which I do not believe, calling such properties "computational" simply passes the buck; it says nothing about what it is to be computational or effective, what properties might be of that type, etc. Since computation must be defined in terms of a notion of efficacy, to use 'computational' as an efficacy predicate is thus at worst false, and at best circular.

The next two, *formal* and *syntactic*, are also commonly used, for example in such statements as that inference is defined over the *syntactic* properties of the premises, or computation works in virtue of the exemplification of a system's *formal* properties. Such phrases are unexceptionable in logic, but that merely betrays the fact that they are both intentional. They make sense only with reference to semantic-L interpretability, or at least the possibility of carrying meaning. One hears talk about "purely formal symbols," but I do not see that as making sense. It would be awry to describe omelets as responding to the formal property of pans' heating up, or to attribute the ability to form benzene rings to the syntax of carbon atoms.[9] Neither

---

[9]In this context we can set aside considerations of Platonic form. If there were any sense to be made, on a Platonic interpretation, of talk about

digitality nor compositionality is the issue; Lego blocks and Meccano pieces are digital, have forms, and support compositional construction, but it would make no sense to say that Lego or Meccano constructions are put together "in virtue of the syntactic shape of their ingredients." So I will assume that if either 'formal' or 'syntactic' is used as an efficacy predicate, it is done so against a background assumption of at least lurking intentionality.

Finally, the lowest three predicates on the list—*intrinsic*, *physical*, and *causal*—are primarily used in philosophical discussions, and are in no way unique to computing. It can be argued that everything in the universe that happens, in the sense of concrete activity, behaviour, etc., results from causal, physical, or (on a common metaphysical assumption I do not share) intrinsic properties of the states of affairs that give rise to it.[10] This intuition has a wide interpretation, called "global supervenience," in which everything that matters about anything depends on the total fundamental physical state of the universe, and a narrow version, called "local supervenience," according to which everything that matters about a particular entity or phenomenon, or at least everything of scientific importance, arises from the physical state of that particular entity or phenomenon.

Because it is the local version of supervenience that is relevant to blanket mechanism, I will take the relevance of all three of the efficacy proposals *intrinsic*, *physical*, and *causal* to be

purely formal states of affairs, that would be a metaphysical thesis, not a substantive claim about computing.

Re "syntactic," Fodor once suggested that 'syntactic' means nothing more than "not being semantic," but the comment was made in the context where it was assumed that there *were* semantic properties, just that they were not the ones responsible for the machine's operation—were not, that is, effective. Sans a notion of semantics or interpretability-L, 'syntactic' makes even less sense than 'formal.'

[10]"Give rise to" is of course a causal property. It is not my aim here to provide a reductive account of causality, which is likely impossible.

captured in the functionally defined notions of 'mechanical.' The only controversial issue surrounding them has to do with the standard stalking horse of whether computation is or should be understood as fundamentally concrete, in which case I take these as foundational properties of being mechanical, or whether it should be understood as fundamentally abstract, in which case their relevance, if any, will be disputed—but as noted above, I take the mechanical grounding of the requisite notion of efficacy to be a *sine qua non* of adequate theory.

That leaves the two efficacy predicates I have concentrated on here: *effective* and *mechanical*. Chapter 6 spoke briefly about their relation to *causal*. The thought was that instances of both effective and mechanical properties must be causal (or causally "efficacious"), even if no one single causal property can be identified with any single mechanical or effective property, taken as a type.[11]

Of the two, I have focused on 'effective' not only because it is the term used in theories of computation, but also because 'mechanical' suggests a particular physical form, evoking images of levers and gears, rather than including a broad spectrum of physical types, including optical and electronic. The terms also have different connotations. While sharing a background sense of function or purpose ("effective at what?", "a mechanism for what?"), 'mechanical' tends to point towards *how something*

---

[11]The motivation for this usage was to accommodate multiple realizability. Suppose being an instance of mark 'o' is taken to be an effective property of marks on tapes of Turing machines of type $\mathcal{M}_i$—that is, to be such that a machine of type $\mathcal{M}_i$ might transition from state σ to state σ' in the presence, and only in the presence, of a mark on the tape of type 'o'. Suppose in addition that $\mathcal{M}_i$ is defined in such a way as that tokens of 'o' for one instance of $\mathcal{M}_i$ are realized by patterns of ink on small squares (squares of "paper," as it were), and by signals of a particular voltage for another. Then the property of tokens of 'o' in virtue of which they can play effectives roles in the life of machines of type $\mathcal{M}_i$ cannot be *causally* defined, since it can take different forms. It is for this reason that being a 'o' would be considered a mechanical or effective property, but not per se a causal one.

works; 'effective,' towards *what it accomplishes.*

What is most curious, and relevant here, is that there is no agreement about this slate of predicates, nor even much discussion—which if any is most important, which beside the point, which foundational, which can be defined in terms of which others, and so on. Whether the notion of efficacy best suited to serve a foundational role should be treated (like 'effective,' in its informal sense) as an issue of achievement, or (like 'mechanical') as a fact of constitution or means of production—this question, too, has no official answer.

Someone might argue that, for computer science, these issues are like issues of interpretation in physics: questions to be wrestled with in the philosophy of the science, or to be discussed at bars after work, but not internally important to the conduct of the field. Perhaps. But until we have answers, we will not have an intellectual grip on what we are talking about when we say that X or Y can or cannot be "computed," or what we are saying when we say that this or that "is computational." More specifically, we need to disentangle these issues if we are to understand how and why computer science uses semantical vocabulary in the way that it does, and how computers do and do not relate to the core intentional architecture.

2 Blanket Mechanism

Consider then the blanket mechanist approach to computation: a restriction of theoretical focus to effective properties, internal or at the periphery of the machine. Various recent philosophical accounts of computing argue for just such a mechanistic account.[12] I will argue against the adequacy of any such approach—any approach that adopts what I will call the **mechanical restriction**.

We need to see how the restriction arose, and what it says.

---

[12]E.g., Piccinini (2015), Glennan (2017).

2a Automata

Start with elementary automata, and their standard analysis in computer science—the theory of "abstract machines" on which (many would say) theoretical computer science is currently founded.

Automata are considered to be "abstract machines" in at least this sense: devices in one of a finite number of states at any given time, which transition from one state to another in response to a discrete unitary (atomic) input. Some automata can "write" an unlimited number of outputs onto a memory structure, such as a tape or push down storage; others, called finite state machines (or simply "state machines"), have no way of storing unbounded information. In general, automata are assumed to be determinate and digital: unambiguously in one state or another, without matters of intermediacy, determinacy, or degree. There is typically no notion of an "instance" of such an internal state, since for theoretical purposes such things would be identical. In the default deterministic case, for any given state and given input, only a single transition may apply, which takes the automaton to a single new state, and potentially inputs from or outputs a token or to the storage. Sometimes probabilities are used to indicate that there are various probabilities of the machine's entering different states. Varieties of automata are also typically defined with respect to an appropriate notion of discrete time.[13,]

What efficacy predicates apply to automata, so described? Not *physical* properties, at this level of generality, because of the abstractness of the definition (famously, Turing machines have

---

[13]A hierarchy of types of automaton is generally distinguished: combinational logical (distinct from combinatorial logic), finite-state machines, pushdown automata, and Turing machines—and sometimes as more refined variants, such as nested stack automata. I will frame the discussion primarily in terms of finite state machines (FSM), in part because they are the most familiar, and in part because they serve as the controllers for pushdown automata and Turing machines. But my comments about automata will mostly be general, applying to all types.

been built of many types of material, including Tinker Toys). *Mechanical?* Yes, but only, at least so far, on a non-goal-oriented version of the predicate, focusing solely on issues of causal organization. Automata are described as *machines*, but until specific functions are attributed to them, or they are interpreted as achieving certain ends, their machinic nature implies no more than they be concretely realisable. Not only must the transitions between and among states be causally implementable; they are also presumed to be able to be made "automatically"—and 'automatic' is a coherent notion only in the concrete world.

As an illustration of the commitment to a mechanical construal of efficacy, note that this condition of causal realisability holds no matter how abstractly the automata are defined. If the automaton is defined concretely, by ostensive characterization, the realisability is typically manifest. In abstract formulations the restriction is absorbed into implicit conditions on the abstract specification. An automaton can in general be characterized by a set of states Q, a set of inputs Σ, a transition function δ mapping $Q \times \Sigma \rightarrow Q$, and possibly (depending on the type) a set of outputs, perhaps with directions of motion on a tape. It would be out of bounds, in defining a particular machine type, to take Σ to be a set of abstract numbers (such as arbitrary reals), or to consist of numerals individuated by relational properties (such as "those occurring a prime number of times in the social security numbers of all U.S. citizens living in Stockholm"). Similarly, if a transition were defined as moving from state q_i to q_j if P=NP, and otherwise to q_k, no automatic procedure could (currently) be built to honor the specification. Fundamentally, inputs and transitions must be automatically implementable by "brute mechanism"—"witlessly," in Haugeland's parlance, by, as is not entirely tautologically said, a "mere automaton."[14]

---

[14]One could try to deflect both cases on the grounds that transition functions (mapping $Q \times \Sigma \rightarrow Q$) are extensionally defined, and so while we cannot presently know, in the examples suggested, which input tokens

That is all. What matters here is that nothing in the account given so far warrants the application of goal-oriented senses of the terms 'effective' or 'mechanical', because nothing has yet been said about what these machines *do*, beyond transitioning from state to state. As a result, there is as yet no warrant for calling automata *computers*, or for describing their behaviour as *computational*. It is not just that detailed questions about programs, semantics, and the rest are premature; whether they compute at all can only be assessed with additional conceptual resources.

In particular, the thought that automata are computers, or can be described as computing, rests on three additional stipulations, often employed in combination:

1. IMPLEMENTATION: You can *implement* computations on these machines—arbitrary computations in the case of Turing machines, on a particularly broad equivalence metric; more restricted forms on automata lower in the hierarchy.

2. FUNCTION: You can understand the behaviors of specific machines, processes built on top of them, and/or the results of those processes, as performing *functions*, satisfying *goals*, serving *purposes*, or achieving certain *ends*.

3. REPRESENTATION: You can *semantically interpret-L* the states, inputs, and outputs of automata as representing or standing for entities lying either inside or outside the machine's boundaries.

---

are instances of which input mark types, and what the transitions are, in an abstract sense both would still be legitimate. The problem with this response is that it risks being so abstract as to eviscerate the notion of effectiveness on which computation is defined. What would then defeat a proposed rule that a machine $\mathcal{M}_1$ should transition from state q_i to q_Y in the presence of an input α just in case another machine $\mathcal{M}_2$ will halt if given α as input, and to q_N if $\mathcal{M}_2$ will not? That function too is extensionally well-defined.
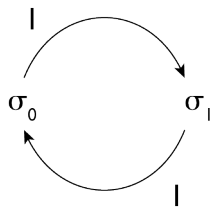
For example, figure 7 diagrams a finite state machine PARITY which, if started in state σ_0 will stop in state σ_0 or state σ_1, respectively, depending on whether its input consists of an even or odd number of strokes (prior to a terminating «#»). Similarly, one can readily define an automaton ADD (figure 8) that takes two binary numerals as input, and, as it is said, "computes their sum"—i.e., produces as output that binary numeral that represents the sum of the numbers denoted by the inputs, also interpreted as binary numerals.[15] Describing PARITY as a machine "to determine the parity of the length of the input" is an exercise of FUNCTION; understanding ADD as "adding numbers," an exercise of both FUNCTION and REPRESENTATION.
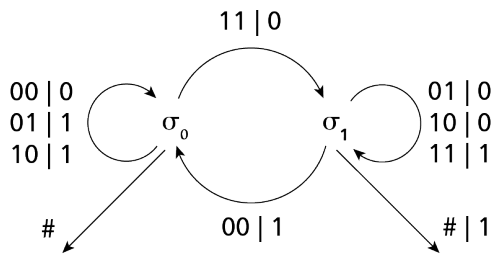
Figure 7 — PARITY

The fact that we need to amplify our understanding of automata with these additional conceptual resources, especially REPRESENTATION, in order to see them as computing suggests that automata without these additional resources may best be understood as computational **substrates**—active media, as it were, on top of which computational processes can be constructed, rather like sheets of paper and supplies of characters in regards to written language. What is conceptually critical is this: from the fact that automata are adequate material components to serve as the

Figure 8 — ADD

---

[15]For simplicity, this finite state machine requires the two binary numerals to be presented in reverse order, terminated by a #. They must be zero-padded to the same length, and the output will be written in reverse order.

basis of computation, it does not follow that the analysis of them *as automata*—as mechanical apparatuses—contains within it sufficient resources to analyse the processes built out of them as computational. No one would describe a theory of arbitrary sequences of letters as a theory of written language, even if it were true that any written language could be expressed as particular sequences of letters. It is not just that legitimate linguistic expressions might constitute a subset of all possible sequences of letters; that constraint might be specifiable by a mechanical grammar. The more serious problem is that, in order to be a legitimate language, linguistic expressions need to *mean* something—and the "theory of arbitrary sequences of letters" would likely lack resources for accounting for meaning. In sum, an analysis of (some) sequences of letters as written language requires a theory of what it is to be a language.[16] By the same token, there is no warrant for calling finite state machines and other automata computers until an account is given of something they *do*—something they *compute*.

2b FUNCTION: Mechanical goals

Although most of the examples considered in this book involve INTERPRETATION-L, it is not yet evident that semantics is required for everything considered computation. Finite state machine PARITY described above (figure 7) can be described and understood without semantic or arithmetic interpretation-L (so long as parity is not considered an arithmetic predicate, which seems arguable given that what is being computed is the parity of the *length* of the input, not of anything it represents). Can we nevertheless say that P *computes* the parity of the length

---

[16]My own view would be that a theory of letter configurations *as language* must at least in part be a theory of how they can convey meaning, but I am of course in the minority: there is a large body of work in computer science called "formal language theory," where issues of meaning and interpretation$_L$ do not arise. Even in that context, however, to say that an automaton is able to parse or produce a language requires the addition of a grammar.

of its input? Yes, under this provision: that determining the parity of the number of strokes in the input sequence is taken as a *goal* or *function* of PARITY or its user.

Why do we need a goal? If PARITY sits in front of us, and we feed it an odd number of tokens, terminated with «#», we might note that it ends up in state σ_O, and say "Look; it has computed the parity of the length of its input." But that is not quite right. Yes, it halted in state σ_I; σ_I is correlated with its input having an odd length; and it would not have ended up in that state had there been no input, or had the length of the input been even. So the final state it reached is counter-factually correlated with the length of the input—"carries information about it," as it is said. But to *compute* implies more—that there was a drive—an "urge," as it were—towards the production of that output, a sense of direction towards that state. This purpose or drive must either be an authentic property of the machine, or imputed by its user or designer.

By analogy, consider tree rings. Do they compute the ages of trees? They too are counter-factually correlated with those ages, and thus carry information about them. But we would not say that they *compute* the age unless computing age were imposed or discovered or hypothesized to be their function or goal.

Under an ascription of FUNCTION, the problem evaporates. There would then be no difficulty in saying that PARITY computed the parity, in calling PARITY's states and inputs mechanical, and in dubbing PARITY a *mechanism*. Since it provides an effective means of satisfying that goal, PARITY could even be described as an *effective mechanism*. But without function, we simply have a mechanical device whose behaviour is lawfully correlated with its internal states and the environmental forces impinging upon it.

The requirement for additional resources, beyond those provided in the bare characterization of them as automata, in order to describe PARITY as "computing parity," and ADD as adding numbers, is not just epistemic. It is not just that, as observers of these machines, we require the concepts of goal,

purpose, interpretation, etc., in order to describe these automata's behaviour in such terms. In order for them in fact to be doing such things there must be goals, satisfaction, representation, and the like, none of which exist solely in virtue of the automata-theoretic framing. These additional properties are required in order for the respective devices to be *successful* at determining parity, in order for its implementation of addition to be *correct*, in order for it to be *effective*. Similar points hold for the mechanical function of any machine: sorting inputs by length, moving disks in the Tower of Hanoi problem, sharpening pencils, etc.

Does that mean that even these simple cases betray the limits of blanket mechanism? Not yet, but they underscore an important point. Just as I suggested that automata may best be viewed as (ontological) substrates for computation, it may be that blanket mechanism, at least in pure form, may best be (methodologically) considered as a framework in terms of which to formulate the *implementation requirements* or *satisfaction conditions* necessary in order for these substrate machines to compute.[17]

Since automata are often viewed as the very foundation of computing, these statements may sound off-kilter, especially to a computer scientist. There are at least three possible reasons. The first is epistemic. IMPLEMENTATION, FUNCTION, and REPRESENTATION are so deeply enmeshed in computational thinking that it is liable to feel wrenching to bring them into explicit focus. Especially since computer science is in many ways an engineering discipline, the ideas that machines have functions, and are subject to norms, or involve simple representation, may simply be presupposed. "Of course machines have functions," a programmer might say. "Of course there are normative requirements governing what they do. Why else would we build them?"

---

[17]To the extent that computer science is an engineering field, it makes sense for satisfaction conditions to feature large in theoretical analysis.

A second reason the comments may sound odd has to do with construction. "You don't need more than an automaton and a memory to build adders, or to construct systems to sort strings by length, or fashion devices to solve any number of other standard computational tasks," a critic might say. Sure enough, the automaton and memory are sufficient material out of which to IMPLEMENT a computer. But *implementation* is not *constitution*. As we saw in the case of logic, and as is generally true of intentional architectures, there can be more to a system's being the system that it is than its effective parts and their effective or effective arrangement.

"Surely," the critic might press on, raising the third reason, "physical configurations and causal forces (i.e., mechanical$_C$ configuration) is all there ever is. That's the foundation of science. It's how the world works. Is more than mechanism$_C$ required in order to say that the heart is a successful pump, or that a particular symbol manipulator is a sound inference system?"

Yes, more is required—and crucially so. That is the point. 'Successful' and 'sound' are normative properties. Mechanisms, machines and their causal ingredients, are not by themselves sufficient to establish the norms that apply to them.

In the biological case, it is well-recognized that biological function is not something that supervenes purely on local causal facts—on proximate pushing and shoving, on immediate causes and effects. If it were, hearts could be described as successful mechanisms for going "lub-dub," since they make that sound. But they are not *successful* at doing so (unless such a goal is attributed to them), since going "lub-dub" is not their biological function. Theoretical biology recognizes the need for *two* accounts, that is: one explaining how hearts pump blood, another establishing that pumping blood is their biological function. The former account can be given causally, and so is unproblematically naturalistic. In the biological case, current practice is to provide the second account in terms of evolutionary role: the heart's function is to pump blood, it is claimed, because pumping blood is what has led to the heart's evolutionary

survival, via the survival of the organisms in which it plays an essential role. This effort to derive biological function from evolutionary role (posterchild of the "naturalizing normativity" project) is assumed also to make the second account naturalistically palatable, thereby allowing biology to remain within the purview of science. But note that while both accounts are in some ultimate sense causal, only one derives from the local causal facts. The other is grounded in the larger evolutionary context—distal in time and space, "outside" the mechanism in which the heart plays its function.

Similarly for logic. The aim of chapter 6 was to show that being an inference system also fails to derive solely from these systems' local mechanical or syntactic manoeuvrings. In order to describe logical systems in inferential terms—as devices that derive consequences from premises, that "prove" that things are the case, and so on—requires locating those mechanical operations within an encompassing intentional architecture, interpreted-L in terms of, and governed by norms involving, non-effective semantic-L relations. Once again, two accounts are required: a local one, about "what happens," mechanistically; and a non-local one, about what the sentences *mean*, about semantics-L. The phenomenon "is what it is"—logical inference, theorem proving, etc.—only as understood in terms of the two accounts in conjunction.

No such resource—no "second account"—makes itself immediately available in terms of which to establish computational function. Computer science is replete with demonstrations of how automata of various sorts can and cannot compute results of various kinds. The goals and their satisfaction constitute the substance of the theoretical claims. But the results are imagined to be results *about automata themselves*; not about what computations can be *implemented on automata*—what computations can be *achieved with* or *by* automata. Half of the story about computing is front and center: the story of its mechanical-C workings. The other half—the representational and normative half—remains hidden in current theoretical practice. Like biology and logic, a full account of the

computation will require this "second" account to be formu-
lated and brought forward, in addition to the automata-theo-
retic "first" account, in order to demonstrate that the latter can
accomplish the former.

2c REPRESENTATION: Numerals and Numbers

Consider REPRESENTATION next.

The simplest and most common attribution of semantics-L
to computers involves taking causally-organized states and
their configurations to represent numbers. The practice is so
universal that it props up the idea that computation can be
studied as wholly abstract. When combined with FUNCTION,
the use of numerals to denote numbers allows computers to be
viewed as doing simple arithmetic computations.

The strategy of taking standard (unary, binary, decimal,
etc.) numerals to denote numbers seems innocent enough. It is
transparent, well-understood by programmers and theorists,
and essentially analytic (immune to error). The other direction,
of projecting numbers onto numerals, is more telling. It is the
simplest case of the projection strategy, described above, of re-
placing non-effective relations and inaccessible objects with ef-
fective proxies.

Consider a program «LENGTH» designed to count (i.e., given
the FUNCTION of counting) the number of elements in a list.
Lists have a *number* of elements, not a *numeral* of elements.
Suppose we use 'three' to name the successor of two, and 'trez'
to name the numeral «3»—so that we can truthfully say that
trez denotes three. What is important is that the list «<A B C>»
has *three* elements, not *trez* elements. But since the number
three is an abstract object, which cannot be returned by an ef-
fective computational procedure, the only appropriate entity
for the procedure to return is (an instance of) the concrete nu-
meral trez. That statement is close to the 2/3Lisp analysis. If
variable «X» is bound to the list «<A B C>», the 2/3Lisp expres-
sion «(LENGTH X)» is taken to represent the length of that list—
the abstract number three—and therefore to return the nu-
meral trez. The number three is the designation or declarative

import (φ) of «(LENGTH X)»; the numeral trez, its procedural consequence (ψ). That trez is the *correct* item for the procedure to return is normatively warranted by the fact that the numeral trez is the normal-form designator of the number three.

But that is not how the situation is viewed in contemporary computer science. First, as noted, the numeral-to-number representation relation is absorbed, unremarked, into the theory, keeping it out of theoretical view. Second, in deference to blanket mechanism, numbers are projected back onto numerals. The result is that the distinction between numeral and number is elided.

Along with ambiguity about the word 'compute' noted earlier, this elision contributes to unclarity about the nature of computing. Programmers and computational theorists alike will say that the procedure has computed that the list has three elements, that three is the *value* of the expression, and that three is returned—using the same term in each case. 2/3Lisp would concur on the first statement: that (to put it a little awkwardly) the procedure *computes that the list has three* (not trez) *elements*. But that is because 2/3Lisp behaviour is described under a representational conception of computing. In 2/3Lisp accounts, that is, I use the term 'compute' by analogy with 'describe'; what is computed is the *denotation* of that which is returned. 2/3Lisp does not use the word 'value' at all; and, as noted above, trez is what is returned. This practice mirrors natural language: if I ask you what the length of «<A B C>» is, you will answer "three"—i.e., will return "three" to me, as the "procedural consequence" of my query, as it were—a *representation* of the length, since you can no more answer with a number than a computer can return a number as the result of a procedure call. Neither natural language nor 2/3Lisp confuse use and mention.

While efficacy considerations require what is returned to be concrete, it is not enough merely to require that it be a representation of three. «(LENGTH X)»—i.e., "the length of «<A B C>»—is already such a representation. Rather, as is clear from other arithmetical cases as well, the representation is presumed to be

a numeral (binary, decimal, whatever). Suppose a procedure is said to "return the number 6,738,743,321." It would be expected to be computationally demanding to determine that result's prime factors—but such a claim is entirely dependent on how the number is represented. If a representation scheme were used which represented numbers by the traditional decimal numeral representations of their prime factors, this number would be represented as «[67829, 99349]», and the calculation of its prime factors would be immediate (though addition would be challenging).

The conflation of numeral and number may seem innocent. To say that «(LENGTH '(A B C))» "returns three, one less than the length of «(A B C D)"» would be unexceptional statement—even though what is returned and what is one less than another length cannot actually be the same thing. But suppose I ask "has any other information been recorded about the person currently logged in," and the system essentially says *no*. If the computational routine to "return the person logged in" returns «$A75BX»—the person's login handle—then the negative reply is likely to mean that no information is stored under this representation of the person. Whether the system has any information on that person, indexed by any other representation, is likely to be beyond the system's ken.

Needless to say, programmers know perfectly well that representation and represented cannot be identified in such cases, even if taking the identity of internal structures to mirror the identity of the objects they represent is a staple of current practice. But it is recognized to be part of computationalists' skill and responsibility: to monitor whether computational (i.e., representational) identity tracks personal identity, and to navigate through any potential confusion. But given the increased use of artificial intelligence and autonomous systems, data mining strategies for training, and the like, for computer science to lack standardized theoretical techniques for tracking such distinctions is troubling. Moreover, the distinction between a representation and what it represents is a foundational issue in semantics. The fact that computer science uses 'semantics' for

something else is also not reassuring.

Return to the case that got us here: numerical computation. As we have seen, in order to be considered *computation*, arithmetic cases, like everything else, require two accounts. The first, hewing to blanket mechanism and staying within the realm of the effective, invokes FUNCTION; the second invokes REPRESENTATION. The fact that two accounts are required even in these simplest of cases underscores the general applicability of the core intentional-L architecture. Non-effective relations between symbols and their referents establish the norms that effective manipulations are constructed to honour. We blur the distinction between the two at our peril.

2d IMPLEMENTATION—I: Programs and Instructions

IMPLEMENTATION is arguably the most important notion in all of computer science—the facility underlying the fact that you can configure computers to perform a virtually endless variety of tasks. The simplest form of implementation is to arrange the hardware directly, so that the machine directly exhibits the desired behaviour. This was the approach with PARITY and ADD. The downside of the approach is that it produces single-purpose devices. As it is said, PARITY and ADD each "do exactly one thing"—perform just that "task" that was attributed to them as their FUNCTION.[18]

---

[18]"One thing" must be understood in context. Neither machine is defined to operate over a single predefined input, in which case that input could be considered to be part of the machine's state. Rather, what is considered a "single" automaton is invariably defined to work over a class of inputs—a class typically defined in terms of allowable configurations of tokens of the input *alphabet* (input token types). PARITY will compute the parity of arbitrary-length sequences of input tokens; ADD will produce that numeral which denotes the sum of the numbers denoted by two arbitrary equal-length sequences of 0s and 1s. PARITY and ADD each do "one thing," that is, in the sense of computing a single function, though as always the function may be applied to multiple different arguments, and the "same result" exhibited in many different instances of its

The innovation that spurred computer science on to greatness was the development of *general-purpose* computers—machines that could be *programmed* to do a variety of tasks. The earliest versions were controlled by hardware (plug-ins, special wiring, punched cards reminiscent of Jacquard looms, etc.), but the real innovation took place with what are called "stored-program" computers, in which what controlled the machine was stored in memory.

The ability to orchestrate behaviour α on general-purpose machine X by using some of X's memory to store a "program" *p* that leads X to exhibit α is an extraordinarily powerful idea— likely the most important underlying the computational revolution. It is also understood in a distinctive way, crucial to our fine-grained tracking of the conceptual resources programmers use to understand computational phenomena. Consider the situation just described of X+*p* manifesting behaviour α. Suppose X+*p* is given an input *s*, yielding output *t*. Rather than viewing this as a case of X being given three inputs—X, *p*, and *s*—the situation is instead usually understood in two stages: first, X+*p* is taken to **implement** another machine Y, where α is Y's behaviour—the "function that Y computes"; and then implemented machine Y is given the input *s*, outputting $t=α(s)$.

Viewing X+*p* as implementing another machine Y is partly motivated by the fact that computers, including automata, are designed to produce, not a single output for a single input, but a range of outputs for a range of inputs. Even calling PARITY and ADD single-purpose machines betrays this assumption: though they do not change from one run to another, they are each capable of being given—in fact are designed to be given— an unlimited range of inputs. This one-to-many structure is a fundamental assumption; it could be said to underlie the very notion of a computer, and perhaps even of machines in general. What we consider a single machine, held or viewed as constant, can be used in many different circumstances, including on

answer.

many different inputs, and will produce many different effects, including many different outputs, depending on the use case. This is why computers are often modeled with mathematical functions: one function yields different values for different arguments.

As suggested above, this one-to-many structure is iterated in the case of programs. In general, machine X will be capable of being given arbitrarily many different programs p. For each p_i, it will be assumed that the particular combination X+p_i (i.e., Y) can in turn at least potentially be given a range of possible inputs. Implemented machine Y is the abstraction of X plus the implementing program p_i, with p_i held constant over the range of inputs to the Y—i.e., X+p_i—combination.

How do computer scientists and programmers understand these notions of program and implementation?

Consider first an account restricted to automata-theoretic language. We can say a program p for an automaton $\mathcal{M}_1$ is a partial input configuration (a configuration of inputs on a portion of the input tape, in the case of Turing machines), such that the combination of $\mathcal{M}_1$ and p can be considered to constitute a different automaton $\mathcal{M}_2$ that behaves in some particular way with respect to various "inputs to $\mathcal{M}_2$"—which is to say, with respect to configurations of inputs to $\mathcal{M}_1$ on the "remaining" portion of its input tape (that portion of the input tape not occupied by p). That is, the behaviour of $\mathcal{M}_2$ is basically the behaviour of $\mathcal{M}_1$ given that a certain portion of $\mathcal{M}_1$'s input has been "locked down" to the configuration p.

One reason this characterization will sound odd is that, from this automata-theoretic point of view, any portion of the input to $\mathcal{M}_1$ could be considered a "program"; $\mathcal{M}_2$ would simply be that machine that exhibits whatever behaviour $\mathcal{M}_1$ would exhibit in response to varying inputs in the remainder of the input, given that the input on the initial portion is held fixed. Even if we invoke FUNCTION, one could characterize p only as a mechanical perturbation that causes a particular effect on the mechanism, or that skews it towards a specific outcome, like a

lobe on a camshaft, or the shape of indentations on a key. Or to put it mathematically, we could say that M_2 computes the same function M_1 assuming that some arguments of M_1 are held constant.

But none of this is how we understand programs. Programs are *interpreted-L*—taken to consist of *instructions*, directives to do something. To understand a program as a program involves the exercise not only of FUNCTION but also of REPRESENTA-TION. To call an input configuration a program is to make an intentional attribution—to take the program to *mean* some-thing, to view it as a direction to conduct specific effective op-erations on the states and structures of the machines in which it is employed. This interpretive stance is evident in contrast with other types of machine. No matter how carefully designed so as to cause certain effects, the shape of a projectile is not an instruction as to how the things it hits should shatter; nor can the reaction of a plant to incident sunlight be described in terms of whether the plant obeyed the sun's instructions. Instructions are normative entities, which come with satisfaction condi-tions—making room for the question of whether the instruc-tion was *properly* responded to.

It may seem heavy machinery to claim that in order to un-derstand the codes that control today's CPUs as instructions re-quires an exercise of REPRESENTATION, but it is only against the sorts of normative framework provided by REPRESENTA-TION and FUNCTION that an implementation can be deemed *correct*.

Very occasionally, instructions are atomic, with no argu-ments—as in the instruction «HALT», leading an automaton to come to a halt; or "skip the next instruction if the content of the accumulator equals the numeral representation of zero," where the accumulator is a unique register and therefore does not need to be addressed, and determining which instruction should be skipped is deictically related to the location of the skip instruction itself, and so also does not need explicit indica-tion either. But it is far more common for instructions to be at

least bipartite: some kind of activity or effective change is indicated, and one or more *arguments* required as well, to specify which objects should play a role in the carrying out of that activity (as source or destination or both).

How are the arguments interpreted-L? At the lowest level, they are usually binary addresses or numerals. In both cases they are understood by programmers as *terms*—names of abstract numbers or of effectively accessible locations. Numerals were discussed above. Binary addresses are parasitic on numerals. A binary address is typically the address of that location which, if it were interpreted as a numeral designating the number n, is n (or $n+1$) locations from the beginning of memory (or designated section thereof). So «110111», the binary numeral for the number fifty-five, would be the address of the fifty-fifth (or fifty-sixth) location in memory. But the number fifty-five can play no effective role in anything, and it would be wasteful for the computer to have to "count" memory locations. So, in line with blanket mechanism, addresses are implemented in the hardware in such a way as to provide effective access to the locations that they are taken to be the names of.

Thus the instruction «(INCR X)», where «X» is an address, might have roughly the following meaning and procedural consequence. The programmer understands it as an instruction to *increment the number stored at location X*, where, in that thought, "number" refers to a number, "location X" refers to a location, and "increment" means increase by one the number currently stored there. On the blanket mechanism construal, the instruction means "increment the numeral stored at the location to which the term «X» provides effective access," where "incrementing a numeral" means replacing that numeral with the numeral that designates the number that is one greater than the number designated by the original.

"This is all obvious," the programmer is likely to respond. "Why make a big deal out of it?" Sure enough, at these elementary levels it may seem innocent. But as in the case of users and login handles described above, that will change when we get to employees, social media, and fake news. And note that the

norms on the effective operations, including norms governing what memory locations are accessed by what addresses, are all defined in terms of the interpretations-L of the various constituent structures, not in terms of their effective projections.

3 Discussion

The case of automata shows how blanket mechanism gets its grip on our computational imagination. At the lowest levels, programs are interpreted_L as *instructions to do this or that*, where "this or that" is usually specified in terms of an operation on internal computational structures, potentially amplified by simple numeric functions. "Load the contents of memory location X and store it in location Y," "increment register Z by one," "read in a byte from the input stream," "test whether memory location W is zero," "shift the control point from point α to point β if condition γ is satisfied," etc. The operations performed by these instructions (*load, increment, read, test, jump*, etc.) impart effective changes to effective computational states and structures. The targets of the operations (*bytes, memory locations, registers, control points*, etc.)—i.e., the entities accessed and potentially affected during their execution—are again effective internal states and configurations. The actions taken by the operations either involve such structures directly, or make changes to them interpretable in terms of arithmetic operations on and conditions about these structures, projected onto structurally effective numerals (*increment, equal zero?, add*, etc.).

That at least is what happens when the instructions are "executed." To put it in 2/3Lisp parlance, that is an account of the "procedural consequence" (ψ) of the instructions. Crucially, however, given (i) the absorption of the numeral-to-number relation into the theoretical background, and (ii) the projection of the number-to-numeral relation back into the effective structure of the machine, this account of *what happens* when the instructions are executed is essentially identical to a representationally interpreted_L account of what the instructions *mean*. For low-level instructions, that is, their semantic-L analysis, under REPRESENTATION, coincides with an account of their effective

consequence, under FUNCTION, modulo absorption and projection.[19]

The coincidence, in the simplest computational cases, of representational meaning and procedural consequence, modulo absorption, projection, and the removal of normative considerations from theoretical view, is the "smoking gun" that has allowed semantics in the computational realm to part company so profoundly with semantics in logic, representation, and natural language.

1. At the simplest automata-theoretic level, the semantic-L subject matters of computational instructions, are the internal states, structures, inputs, and outputs of the machines—states, structures, inputs, and outputs that are examined, tested, and affected in their operation of these instructions, along with simple mathematical entities (numbers primarily) capable of being internally projected.

    When representationally interpreted$_L$, that is, programs' task domains—the realms to which the instructions bear non-effective semantic-L relations—are the machines' mechanical innards and peripheral connections.

2. What instructions mean, in these simple cases, is taken to be what happens when the instruction is processed. In the case of machine instructions, that is, declarative import (φ) is defined to *be* the procedural consequence

---

[19]The statement that the received computer science view "elides the difference between procedural and declarative accounts" is phrased in 2/3Lisp terminology. It is of course not how the situation is described in present-day computer science. From that perspective the two views were never distinguished in the first place. Mapping instructions onto their effective consequences—onto what happens within the confines of the machine—is simply taken to be "what semantics is," an approach that glosses distinctions critical to programmers' understanding of both the mechanism and the governing norms.

(ψ)—to be an instruction to carry out the actions that constitute the procedural consequence.

The elision, in these proscribed circumstances, of the differences between procedural consequence and declarative import suggests that the computational approach fits into the long history of taking the (semantic-L) denotation of commands to be their satisfaction conditions: what happens, or needs to happen, in order for the commands to be obeyed. Or so at least it seems, so long as the satisfaction conditions do not involve any representational or intentional states. If person A reminds person B, who is giving a research talk, to thank the graduates who made the project possible, the satisfaction conditions of the request apply to the *content-L* of B, not to its physical, causal, or effective nature. Satisfaction conditions like "thank α," "ask β for a favor," or "reassure γ" are not the kinds of satisfaction conditions that blanket mechanism would license.

If computer science elides the difference between the representational and procedural views of instructions, can we say that it *conflates* them? Not quite—because programmers and computer scientists must understand the distinction between numbers and numerals, and more generally between representations and what they represent. Machine instructions, such as "test whether x equals 0," "increment location y," "add z and w," etc., are unarguably understood, at least in the first instance, in terms of numbers—i.e., in terms of REPRESENTATION. Loop termination tests are used to determine the *number* of times the loop has been executed, not the *numeral* of times (which is not a well-formed idea). As in all core intentional architectures, it is these non-effectively represented numbers that structure the norms governing the implementations of these instructions— the norms that dictate what the these instructions *should do*, the norms that engineers must ensure that the hardware honours. The mandate on machinic computation is that it be effective, however—that it produce mechanical changes to causally instantiated configurations of bits. So there is no ambiguity as to how the machinery must be built. On pain of going

metaphysically astray, it must deal with numerals. It follows that mechanism, abstract objects, and warrant for governing norms must all be kept distinct in the programmer's mind.

Put it this way: no artificial intelligence will truly understand programming until it knows the difference between numerals and numbers, knows which properties are "proper" to each, and understands (even if implicitly) the normative structure governing the overall situation.

4 Third Diagnosis

This brings us to the third and final diagnosis: not only of what blocked computer science's understanding of 2/3Lisp, but of something that I believe has since the beginning blocked computer science from understanding computing *as computing*.

The first diagnosis tried to explain the inscrutability of 2/3Lisp from most computational perspectives in terms of differing conceptions of the notion of a *program*. There was merit to noting the different ways that programs are understood, but attempts to push hard on the diagnosis led to untenably baroque complexities, many of them semantic. The second diagnosis shifted the focus to semantics, attempting to pin the differences on how intentional structures are understood in computer science in general. But that, too, led into intellectual cul-de-sacs, forcing us to confront a variety of odd claims, such as that computers cannot do arithmetic. It also seemed difficult, at least in the context of programs, to draw a line between declaratively specifying behavior and procedurally engendering that behavior directly—in spite of the fact that the distinction makes sense in natural language. This raised perplexities about what was going on that seemed hard to answer. The third diagnosis shifted focus again, in order to unearth an underlying tenet to which computer science is methodologically committed, and examining how its conception of both program and semantics have been reconfigured in its terms.

In particular, the third diagnosis claims that computer science is committed to what I have called *blanket mechanism*: an

overarching assumption that the theoretical discourse of computer science can be restricted to what is mechanical—to mechanical entities and effective relations among them (even if both are abstractly or mathematically described). More specifically, under blanket mechanism's mandate:

1. Non-effective semantic relations are debarred from theoretical analysis. The simplest of these, such as the relations tying numerals to numbers, and the relations of both of those to addresses and to locations, are absorbed into the methodological backdrop, out of theoretical view.

2. The normative conditions governing computation, constituted by non-effective relations to external and abstract semantic realms (including task domains), are not considered part of semantical analysis, or indeed to have any role in formal computational theory at all.

3. It nevertheless continues to be tacitly recognized that computing is fundamentally a semantic/intentional phenomenon. Without this recognition, it would be impossible to understand it as *computing*. This recognition is betrayed in the fact that computational discourse remains permeated with intentional vocabulary (*data, program, language, reference, interpretation*, etc.), and is reflected in the fact that something called "semantics" remains prominent in theorical analyses.

4. All traditional semantical vocabulary is redefined to describe or refer to internal mechanical phenomena, not randomly, but in ways that allow formal analysis and informal intuition to track common sense, at least in some approximate ways. However there remain major differences. The main use of the term 'semantics' in computer science is to describe the *procedural consequence* of what happens, effectively, when internal structures are processed. What they *represent* is not dealt with (except when what they represent *is* the procedural consequence.)

The situation is thus almost the exact opposite of logic, where what structures represent is taken as primary, and to *constitute the semantics*, and where procedural consequence (inference) is viewed as derivative, normatively governed by the extent to which it does justice to semantics. In computer science, in contrast, the procedural consequence of a procedure is identified as *being the semantics*, foreclosing analysis of what it is that computational structures represent, and also of what norms it is thereby enjoined to honour.

(Human language and thought are also normatively governed by the extent to which they do justice to semantics. In logic, however, the semantics are generally defined independently of, and prior to, processes of inference. In the human case—or so I will argue in chapter 7—semantics instead arises pragmatically in conjunction with use.)

5. To enable these accommodations to blanket mechanism, computer science adopts a *projection strategy*, in which all relevant non-effective semantic relations and intentionally targeted objects are "replaced," as it were, with mechanically effective internal proxies to serve in their stead (often access relations, in place of representation or reference). Semantic terminology is "folded back" into the machine so as to refer to these effective proxies.

6. Ambiguities introduced by these practices of absorption, projection, and erasure have not only hindered full analysis, but also obscured the true nature of computing—as witnessed for example by equivocation in the use of the verb 'compute,' confusion about whether computation is fundamentally mechanical or semantic, and misleading implications of such technical terms as 'correct.'

It is instructive to review some examples of this "folding back in upon itself" procedural construal of semantics that we have encountered in this exploration—cases where, from our external

point of view, we can understand how the ingredients, processes, behaviors, etc. of a computational system bear non-effective semantic relations to entities in the program's task domain, but where, in the official theoretical account, those relations, and the entities to which they are related, are projected back inside the mechanical systems and rendered mechanical/effective:

1. In chapter 2 I pointed out something that must perplex anyone of an intentional bent: in all standard Lisps (i.e., all besides 2/3Lisp) both «(QUOTE 3)» (i.e., «'3») and «3» evaluate to «3». Evaluation is not a standard logical notion, but if it means anything in logic, it has to do with mapping a symbol or designator onto (what logic would understand to be) its "semantic value." If Lisp evaluation meant "produce as a result what logic would take to be its value," that would explain why «(QUOTE 3)» should evaluate to (the numeral) «3», but would predict that evaluating (the bare) «3» would generate an error, because abstract numbers cannot be returned. If evaluation simply meant "yielded as the result of processing," there is no reason why (QUOTE 3) should not also be self-evaluating—or, for that reason, evaluate to something else entirely ("yielded as the result of processing" betrays that normative considerations have been banished from theoretical view).

    In sum, the classical sense of logical value (reference or interpretation-L) is required in order to explain the first result, why «(QUOTE 3)» evaluates to «3», but fails to explain the second, why «3» is self-evaluating. There is no explanation of the second within the official computational analysis; it is simply stipulated.

    Both results, however, are predicted by the third diagnosis. Since the (logical) semantic value of «(QUOTE 3)» is *in fact* the numeral «3», and since that numeral is an effective ingredient within Lisp systems, and since tokens of the semantic relation between «(QUOTE 3)» and «3» can be projected onto an effective access relation, then Lisp

evaluation can be defined to return it. Since the interpretation-L of the bare numeral «3», however, is not an effective structure (is not "within the system," in Newell's phrase), Lisp evaluation must be reflected back inside the machine to the closest analogue of what it "should" be—i.e., to its projection, which is the numeral itself.

The situation with respect to truth-values is similar: «NIL» is self-evaluating in traditional Lisps, because falsehood is not an effective mechanical ingredient. (Most Lisps do not have a distinguished internal token for "true," but if they did, that too would be returned as the "result" of processing «(EQUAL 'A 'A)».) Similarly with structures (s-expressions) whose interpretation-L is other s-expressions within the system. The s-expression «(CAR '(A B C))», which Lisp programmers understand to mean "the first element of the list «(A B C)»," evaluates to the atom «A», which is "correct" (i.e., is its interpretation-L), since «A» is an effective internal structure. It is only in those cases where the interpretation-L is not an effective machine-internal structure that evaluation (interpretation) needs to "reflect back inside" the machine to refer instead to its proxy.

2. In his "Physical Symbol System" paper, as noted under PRO-3 in chapter 3, Newell claims—correctly in my view—that "[t]he most fundamental concept for a symbol system is that which gives symbols their symbolic character, i.e., which lets them stand for some entity." This is music to the ears of anyone with intentional predilections. Three sentences later, however, Newell goes on to say, "[o]ur concept is wholly defined within the structure of a symbol system." He takes "the structure of a symbol system" to mean the *effective* structure—a restriction of the system to its locally-connected effective (causally efficacious) ingredients. This is betrayed another three sentences on: "[a]n entity X designates an entity Y relative to a process P, if, when P takes X as input, *its behavior depends on Y.*

The juxtaposition of the two quotes again validates the third diagnosis. The first stands as evidence that it is not Newell's intention to banish everything about semantics or symbolism. On the contrary, he wants to retain what he can of the intentional, within the scope of his metaphysical commitments. The second gives voice to those commitments: everything must lie inside the mechanical restriction. Similarly, 'depends on,' in that statement clearly means "*effectively* depends on"—and more generally, it is clear that he takes P, X, and Y all to be effectively individuated. So what he is able to "retain" of the intentional character of computing is its internal projection, in line with blanket mechanism.

3. The mechanical restriction is evident in programming language semantics, as we saw in chapter 3. All programmers would understand at least one of the roles of the identifier «HIGHEST-PAID-EMPLOYEE» (discussed in §3.2a) to be to serve as a representation (in the classical sense) of the real-live human being who earned the most income. But this is another case where the true semantic relation (interpretation-L) of a computational structure is non-effective—a semantic relation to a person outside of the machine—so it is blocked from consideration in official accounts. In its place—and, interestingly, contravening the conventional wisdom that theoretical analyses of programs should not advert to issues of implementation—accounts of programming language semantics take the "referent" of «HIGHEST-PAID-EMPLOYEE», along with all such identifiers, to be a location in an associated memory, onto which the person is projected.[20]

4. As well as targeting only procedural consequence, not declarative import or their relation, computer science also construes processes purely effectively or

---

[20]Again, as always, theoretical treatments may *model* that memory as an abstract "store," but that does not impinge on the point.

mechanistically—i.e., *narrowly* rather than *widely*, as might be said in a philosophical context. So if a payroll system changed the numeral in Robin's monthly pay record from «5,000» to «4,000», the official theoretical accounts would not be able to formulate a claim that the action *reduced her monthly pay from $5,000 to $4,000*, even though that is how a programmer (and she herself) might describe it. Rather, it must content itself with an account that merely says that there has been a numerical change in a data structure.

5. Yet another symptom of the mechanical restriction, in programming language semantics, is the widespread assumption that operational and denotational accounts of the semantics of a programming language should be *provably equivalent*.

   As mentioned earlier, it is common for people outside of computer science to assume that these two approaches, operational and denotational, must be analogous to proof-theoretic accounts of what can be formally derived, and to semantic[21] accounts of what is semantically entailed—and therefore to think that a proof of their equivalence be a substantive result—essentially a *completeness* proof, in a computational context. But as pointed out in chapter 4, this is not the case. Operational and denotational accounts of a programming language's "semantics" are in fact two different styles of *describing the very same thing*—one via concrete analysis, usually in terms of an abstract model of an implementation, another via mathematical modeling. They are both analyses of the effectively individuated behavior to which programs in the language give rise, when "interpreted-C"—i.e., when executed or run, according to the rules of the language processor.

6. In §2.5 I stated a thesis (RSS) on which 3Lisp is based: that reflection is simple to build on a "semantically

---

[21]Or, as is sometimes said, model-theoretic.

sound base." The idea was that the design for the mechanical architecture would effectively "fall out" of the semantic framework, once that semantic framework was properly understood. As noted in §..., above, the most important aspects of being "properly understood" included: distinguishing procedural consequence from representational import; having a clear sense of how the two relate, and of what normative conditions accrue to each, and to their relationship; and knowing how to integrate them into a single encompassing theoretical framework.

If, under the mechanical restriction, 'semantics' is understood to mean what it means in contemporary computer science—whatever behavior results from processing the program—this thesis is rendered either vapid or tautological. At best, it could be taken as the empty claim that "a system is simple to design if you (already?) understand how it is supposed to work."

7. Numbers are a special case. As noted in chapter 3, and as elaborated above, if computing is understood merely as transformations among concrete, effective symbols, without non-effective semantic relations being theorized, then computers cannot properly be said to *add numbers* at all; they can merely be described as producing numerals that denote or represent numbers on the (classical!) sense of denotation or representation. According to blanket mechanism, that is, the computational theorist should be prohibited from saying anything about mathematical entities, and should therefore have to deny that computers have to do with them.

On the surface, the third diagnosis removes this difficulty. If relations between numerals and numbers are freely absorbed and projected, computer scientists can discuss numbers with abandon. The practice also explains why talk about "32-bit numbers," or "floating-point numbers" does not sound oxymoronic ('32-bit' and 'floating-

point' make sense only as properties of numerals).[22]

The practice is certainly widespread—permeating such descriptions as this:

> As the name implies, floating point numbers are numbers that contain floating decimal points. For example, the numbers 5.5, 0.001, and 2,345.6789 are floating point numbers. Numbers that do not have decimal places are called integers.[23]

Needless to say, these are descriptions of numerals. Decimal points and decimal places are facts about the representation of numbers, not about the numbers themselves.

8. Finally, the mechanical restriction explains something noted in chapter 3: why it is that computer science has some claim to understanding the semantics of programming languages, but not of the programs written in them. On any of several construals of 'program,' but perhaps especially the prescriptive/specificational reading, programs are "about" the behavior they engender, on a classical understanding of 'about'—and so staying within the confines of the mechanical restriction still allows these officially-licensed accounts to do at least partial justice to how a programmer might understand how the language works, mechanically. When it comes to individual programs, however, as explored in chapter 4, programmers universally understand the symbols (variables, constants, etc.) to have an interpretation-L— sometimes within the operations of the system itself, but often in the world or task domain with respect to which the program is written (and by which it is

---

[22]Compare these examples with 'Roman numbers,' which is obviously malformed. As everyone knows, the proper term is 'Roman numerals.'
[23]https://techterms.com/definition/floatingpoint; downloaded Aug 12, 2020.

normatively governed). The semantics of individual programs, that is, cannot be understood within the mechanical restriction—since computation (at least computation in the wild) is in fact an intentional phenomenon, exhibiting genuine unrestricted semantical relations to the world, to which it is normatively accountable.

Many more such examples could be cited. But one final point seals the case. We can use the third diagnosis—computer science's pledge of allegiance to blanket mechanism—to explain both the first and the second diagnoses, having to do, respectively, with the meaning of the term 'program' and the construal of semantics.

Re the first diagnosis: one advantage of the ingrediential viewpoint on programs, of the sort inchoately incorporated in 2/3Lisp, but planned to be fully embraced in 4Lisp, is that it recognizes that programs use terms, as programmers well understand, that refer to entities in the task domain. Since many or most of those relations are non-effective, however, they are ruled out of court as legitimate subject matter for analysis under blanket mechanism. That pushes programming language semantics to give an account of the *operational consequences* of the programs' execution—a move that in turn suggests that they must be "about" those structures and operations, like instructions in low-level machine-language programs. That in turn pushes the conception of programs towards a specificational/prescriptive view, bolstering the idea that program identifiers such as «CURRENT-FLOOR» and «HIGHEST-PAID-EMPLOYEE» are genuinely about the memory locations to which they provide effective access. This conception of what a program is "about" dovetails with the projection strategy: the respective memory locations serve as proxies for the real floor and human employee.

But there are two problems. First, when internal structures are the genuine interpretation-L of instructions, as in machine-language programs, they can serve as legitimate grounds for the

normative standards for such programs, even if that normative governance is not theorized (at least not as normative governance). In higher-level programs, however, when the projection strategy replaces task domain entities with effective internal proxies, it loses access to the situation and states of affairs in terms of which norms on programs can be stated.

Second, the projection strategy's removal from theoretical view of the non-effective semantical relations targeting task domain elements means that theory sidesteps the semantical ambiguities we encountered earlier—such as whether the denotation of a data structure is the person in the world about which it contains information, or the storage location where that information is kept. As in so many cases, projection make the answers to such questions easy: computational structures are "about" that to which they provide effective access—memory locations, and operations upon them.

Re the second diagnosis: since blanket mechanism restricts theoretical accounts to effective structures and operations on them, differences between declarative and procedural interpretations of programs (or indeed of any computationally internal structures) are either minimized or erased. This is especially true of "base-level" symbols, which an outsider would take to represent entities, phenomena, and behavior in the real world. The situation gets more complex when metalevel-L entities play a role, particularly when the entity thereby referred to is itself an effective ingredient within the system. In such circumstances, the referred-to entity itself can be dealt with directly in the theory, without needing a proxy. This is why, in all Lisps except 2/3 Lisp, «(CAR '(A B C))» returns (evaluates to) «A», not to «'A». But not all metalevel-L structures refer to effective internal ingredients. Closures are an example. In Lisps, lambda expressions («(LAMBDA ...)») represent functions. Functions are abstract objects, even when considered intensionally. The effective proxy for a function-in-intension is a closure, but closures are typically not representable in full detail within the *implemented* language, often generating an expression marked with a

special purpose tag (such as «<CLOSURE (#F X)>»), which cannot be "applied".

Two final comments.

1. Nothing compels computer science to adopt the mechanical restriction. Nothing requires computing to be analysed within a methodological (let alone metaphysical) commitment to blanket mechanism. Our intellectual experience with formal logic over the past century and a half should make that evident—but so too can 2/3Lisp. Both dialects were designed and analysed, and functioned perfectly well as effective concrete systems, from within a logic-inspired sense of semantics as primarily consisting of deferential, non-effective relations towards realms that can and often do transcend the limits of what is "within the machine"—transcend what falls within the mechanical restriction.

2. Some readers will have been troubled, throughout this whole analysis, by something like the following argument. "All these semantical-L relations are irrelevant to computation qua computation," they might argue. "After all, automata theory treats computers merely as uninterpreted$_L$ digital state machines. Complexity theory, though framed mathematically, similarly views computation in terms of states of semantically uninterpreted-L (*purely formal*, as is sometimes said) digital machines. Potentially even more seriously, even run-of-the-mill programs—as texts, as code, and in terms of the unfolding processes they engender—*can* be analysed without analysing any semantical-L interpretations."

    Much of this is true. The last point is true in particular cases. It does not follow, however, that the types and categories in terms of which we understand computation—especially computation in the wild—can be defined in terms of their projection onto uninterpreted$_L$ physical states. The same could be argued for a theorem prover or system of logic. One can always provide a step-by-step

account of what any logical system is doing, from a purely syntactic or mechanical point of view, without making reference to the interpretation-L (*I*) of any of the expressions, premises or conclusion. But that would not be an account of it *as inference*, as steps in proving a theorem, or coming to a logical conclusion from a set of premises. By the same token, consider the attempt to characterize a program in automata-theoretic terms, as simply "locking down" some of the inputs to a machine, and leaving the rest open. While not false, it is not a characterization of *what it is to be a program*.

By analogy, consider writing. Any given piece of paper on which something is written can be described in terms of distributed points of ink, without reference to what configurations represent letters, which combinations of letters form words, which arrangements of words constitute sentences, etc.—let alone what language the text is written in, or, most seriously, what any of it means. Someone who objects to understanding written language in terms of meaning, someone who argues that "in principle" there is no more to written language than configurations of marks on contrasting backgrounds or strings of letters, can be accused of changing the subject. Such an account would no more be a theory of *written language* than an accounting of the spatio-temporal distribution of organic molecules would serve as an account of mammals, irony, or fame.

Just as an account of the steps taken by a mechanical device is not an account of doing *logic*, and an account of possible configurations of letters or marks is not a theory of *language*, so too an account of state changes of an uninterpreted digital state machine is not an account of *computing*. The fact that the first two examples will likely garner sympathetic readings, but that the last will unleash resistance, shows how deeply the mechanical restriction has taken root in our intellectual imaginations. So much the worse, I believe, for the current state of theory.

One sign of the problems of characterizing writing as no more than configurations of ink on paper is combinatoric. The number of possible configuration of black spots on white paper is astronomical, of which those that constitute inscriptions of intelligible language constitute a vanishingly small subset. To understand computing merely as the causal (effective) consequences of arbitrary configurations of digital states is similarly vast, compared to the number of the programs that we actually write—the number of programs that we could *understand*, the number that could be understood in programmers' default understanding (V3 in chapter 4's typology). Sure enough, for purposes of digital design, it may be useful to have some abstractions of computers as digital substrates that treat only their unrestricted compositional binary states. But that is not *computing*—not the phenomenon that for close to a century has upended the world.

8 Conclusion

1 Summary

What have we learned?

Three things at the outset.

First, at the ontological level, computation—especially computation in the wild, computation as it is used and cared about—is inextricably and ineliminably intentional. The fact has been clear since the outset, betrayed in the fact that computer science's technical vocabulary is more reminiscent of rationalist accounts of logic and knowledge than of empirical studies of the physical world. Considerations of representation and semantics permeate computational practice: the notion of a program, the idea that automata and other computational devices are capable of performing arithmetic operations, the idea of data, notions of addresses, pointers, and references in computer systems, the use of identifiers in programs and labels for data structures that refer to entities and situations in the program's task domain, and so forth. The vanishingly small subset of automata-theoretic configurations of interest in computer science are those that support semantic interpretation. Sans representation, in fact, an automaton cannot be described as computing, at all; it merely transitions from one state to another, depending on the effective properties of its internal states and of any perturbations impinging upon it. Pure automata—uninterpreted digital state machines—do play a role in computer science, but as substrates for computing, not as the phenomenon of computation itself.

Second, the descriptive/theoretical enterprise of computer

science focuses on the *effective machinery* underlying computing—on the automata-theoretic and other effective properties of the (usually digital) material substrates on which computational processes run. This effective focus is fueled by two things: (i) an abiding concern with implementation—with the effective conditions and requirements on any material substrate capable of running a computation, rather than with the constitutive properties of the computations thereby run; and (ii) an overarching methodological commitment to mechanical explanation and to scientifically recognizable forms of theoretical description. The two considerations come together in the field's embrace of *blanket mechanism*: a joint ontological and methodological restriction of attention to mechanical phenomena and relations, eschewing any considerations of representation and genuine semantics.

Third—substantially complicating the situation and obscuring the true nature of the situation—in spite of this effective focus, computer science continues to use semantical vocabulary to frame its analyses, descriptions, and claims. This continues even when the subject matter is restricted to (or projected upon) mechanical configurations and states. We have seen this in standard accounts of programming language semantics, where the "semantic value" of an expression α is a priori restricted to be an effectively-identifiable entity or activity β within effective reach of α—something that can be returned or performed when α is run or executed. Sometimes, for example in the case of a memory pointer in a low-level language, or in the case of a quoted expression such as «'x» in Lisp, the effectively reachable item β is at least arguably the genuine semantic value of α (the memory location pointed at, and «x», respectively), and so the two readings coincide: semantic-C value and semantic-L value are identical. But when the *numeral* «5» is claimed to be the "semantic value" both of the expression «(+ 2 3)» and of the numeral «5» itself, the only plausible reading is that what is meant by 'semantic value' in such a claim is semantic-C value; there is no doubt in anyone's mind that in both cases the semantic-L value is the abstract number five.

More generally, the study of what has been called programming languages semantics, on either the ingrediential or specificational view of programs examined in the first diagnosis, is about the effective conditions on the machines that implement programs written in those languages, without regard to what the terms or identifiers in those programs mean or signify in any representational sense. This is why, as noted in chapter 3, the discipline studies the semantics of programming languages, but not the semantics of programs written in those languages.

Confusion is increased by the profusion of ways in which semantic/representational entities are classified.

The use of mathematical entities to classify subject matter phenomena is a staple of scientific inquiry. The classificatory situation is relatively straightforward in cases where the phenomenon being classified is itself neither abstract nor intentional: 997 kgs of water per cubic meter, ~11.2 km/sec for the escape velocity from our planet, etc. When the classified phenomenon is itself intentional or semantic, such as a natural language sentence or a human thought, things grow more complex. In informal practice, it is standard to classify such phenomena by their semantic content. "I'm thinking *that the election is just 19 days away*," someone might respond when asked "What's on your mind?"—where the election's being just 19 days away is the (semantic-L) content of the thought being described, and the thinker uses that proposition to classify the thought they are having.[1] In such cases of classifying intentional states, that is, there are two semantic relations involving the state and its content: one relating the thought to its content; another relating the thinker to the thought. In the former case, the semantic relation runs from state to external world—from thought to electoral situation. In the latter, because the proposition is used to classify the thought, the "arrow of directedness," and hence the deference, runs the other way. The base-

---

[1]We have essentially no introspective access to thoughts except via their contents, and so classifying them that way is our only practical option.

level thought is normatively accountable to the electoral facts (if the election is sooner, we can correct the thinker), but for the meta-level cognitive theorist, the content is accountable to the thought. For such a theorist to use the proposition "that the election is just 18 days away" to classify the thought, even if it happens to be true, would be wrong.
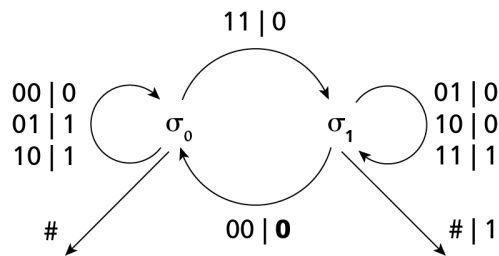
In folk psychology these facts are so evident that it is pedantic to spell them out. But given that computer science explicitly theorizes neither norms nor genuine semantics, it can be difficult to discern the direction of deference. When ADD was described in chapter 7 as an automaton that *adds numbers*, I did not say explicitly whether that account was using addition to classify the automaton independent of the semantic content of its inputs and outputs, or whether it was classifying it *by* their representational content. On reflection, it is clear that the latter would normally be the case. If instead of figure 8 (§7.2a) I had presented the state diagram shown in figure 9, the reader would have been appropriate in responding "that is wrong," "that doesn't work," etc.—betraying that the automaton is governed by a norm of adding that applies in virtue of a presumed binary representation relation from inputs and outputs to numbers.[2] That is, it was classified by the content its inputs and outputs were *supposed* to have—i.e., by a normative account of their content.

Figure 9 — wrong ADD

---

[2]Due to the fact that the boldface zero being written is forgetting the "carry bit" implied by being in state σ_1. For example, this machine will say the sum of 1 and 1 is 0.

The bottom line is something that this book has demonstrated since the outset: current computer science does nothing to illuminate issues of semantics and interpretation-L, and "uses up" the semantic language that would normally be used to describe the situation.

Per se, this is a dismal conclusion—not one it should have taken this much work to get to. By digging deeper, however, we have uncovered considerable consequential richness under the surface.

The pressures fueling blanket mechanism—the practical concern with implementation, and the theoretical attempt to corral computer science into the familiar mould of causal explanation[3]—have submerged discussion of the intentional character of computing in two distinct ways.

On the theoretical side, because the field's aims have been unified around no more than accounting for computing's effective dimensions, theory has been free to make use of unmarked, untheorized representational relations. So long as focus on (and deference to) effective properties and relations is maintained, theoretical discourse can be conducted either concretely or mathematically, and in terms of either sign or signified, without worry that either indiscriminate mathematization or unremarked signification will mar the theoretical results. One striking example is the presumptive isomorphism between operational and denotational approaches to programming language semantics. Another is in the characterization of computability in terms of "computable mathematical functions." Theories of what functions are computable not only rest on a constrained concept of effectiveness, but also assume a background notion of a "reasonable encoding," violation of which wreaks havoc on the results.

Similarly, the routine elision of any distinction between

---

[3]Even if it eschews outright physical considerations, by theorizing effectiveness rather than causality, modeling computational states mathematically, etc.

numerals and numbers, as in descriptions of simple hardware as doing arithmetic, the idea of a "32-bit integer," etc., are harmless so long as the overarching theoretical purpose is to expose and theorize effectiveness constraints, not to investigate semantic relations. In general, that is, so long as practicioners recognize and honor the abiding concern with the effective (as Goguen and Meseguer did in developing a "programming language semantics" for 2Lisp), the relation between theory and phenomenon can be permeated with unremarked mathematization, modeling, and semantic relations, without causing confusion.

On the practical side the situation is more complex. Programmers are concerned with implementation and concrete details of the programs they write, but as has been noted here throughout, especially in chapter 4's characterization of their understanding of computing (v3), they must also understand the myriad intricate semantic relations between and among their processes and the task domains for which they write programs. They navigate issues of when what is at stake is a computational structure, and when what matters is the state of the world that the structure represents. Not only that; they understand, and write programs to deal appropriately with, the equally intricate ontological issues relevant to the norms and semantic descriptions: when two "files" are the same file, for what purposes two "copies" of a data structure have to be considered as different and when they can be considered the same, and so on.

The more one studies the situation, in fact, the clearer it becomes how much of the conceptual structure of programming is ignored or glossed over in current computer science. A not insignificant corollary of this state of affairs is that programmers' untheorized expertise emerges as stunningly impressive. One of the mandates we should place on developing a more adequate theoretical framework is not only to provide programmers with assistance in managing the intricacies of these intentional and ontological realms, but to recognize and honour their superlative skill, bring into view the subtlety of their

intuition, and raise the stakes for what it will be for these skills to be shouldered by the next generation of artificial intelligence system.

2 Directions for Computational Theory

How should theory develop, in light of what we have learned?

First, by far and away the most important fact about any new account of computing, if it is to be an account of computing *as computing*, must be its recognition of both the effective (semantic-C) and the non-effective (semantic-L) dimensions of the semantics of any computational situation. Analyses of logic, complete with accounts of syntax, proof, semantics, entailment, and completeness, show the way, including in their employment of user-supplied interpretation functions as parameters in order to recognize that the semantic interpretation of predicate symbols and terms will depend on users and use. Computational structures will in general be far more complex than logical statements, but the pattern will be the same: facts specific to users and uses will need to be "externally supplied," as it were, before interpretations-L can be "tied down." Analogues of soundness will be definable in terms of those user-supplied parameters.

Second, in a major difference from logic, the story will need to account for both static and dynamic dependencies between and among the effective operations and non-effective interpretations-L, reminiscent of the relations between procedural consequence (ψ) and representational import (φ) in 2/3Lisp. The actual relations theorized for those dialects were a hack, but the framework was designed to accommodate more substantial relations. As well as including causal interactions, the relations will bring in dependencies and normative conditions. In particular, perhaps the most important fact about the multidimensional analysis must be its recognition that the normative conditions governing the system will be generally (if not always) involve non-effective properties of the system's place in the world.

Moreover, we should not expect the accounts of what

happens (ψ) and what things signify (φ) to be independent. That is, semantic content is likely to depend on dynamic use, and use to depend on semantic content, especially at any level at which either is effable—any level at which constitutive regularities can be intelligibly articulated.

There are myriad accounts in the philosophy of language and mind of ways in which semantic content may depend on factors involving use—including pragmatism, Wittgensteinian accounts of meaning as use, conceptual role semantics, and the like. By and large, these efforts are not theorized in anything like the detail that would be required for technical computational use, but the considerations they involve are likely to apply to computational cases as well. In some cases these philosophical approaches make simplifying assumptions inapplicable to general computation, such as a presupposition in most accounts of conceptual role semantics that the realm of conceptual activity is internal to the head of the speaker or thinker, and the realm of reference external (in general neither assumption will hold in a general computational setting).

Third, as we have seen, the proper understanding of a computational system will require recognition that classes, abstract data types, and other forms of abstraction are critical to its non-effective analysis. As we saw, even if such abstractions are eliminable for purposes of a purely effective account, they will be framed in terms of objects, structures, and relations critical to—perhaps even defined in terms of—its non-effective and normative analysis. Even if a class defined for a university course is (effectively) implemented in terms of a suite of atomic identifiers—course number, department, enrolment status (graduate or undergraduate), instructor, etc.—the fact that it represents a course may be essential to know in order to determine whether it honors a variety of applicable norms (e.g., involving student-teacher ratios).

Fourth, in order to accommodate these substantial relations and interactions between referential and other non-effective dimensions of the situation and dynamic forms of effective activity, it will likely be best in general not to associate the label

"semantics" merely with the former, and not to assume that an account of the semantics of a language or system can be analytically defined on its own. Instead, the 2/3Lisp experience suggests taking general significance (Σ in 2/3Lisp) as the backbone analytic category, and then defining referential (φ) and effective (ψ) projections in terms of it, at any relevant level of abstraction at which they are intelligible. For example, in the 2/3Lisp case it is only Σ that it makes sense to define compositionally. Sure enough, because of causal closure, effective activity (ψ) can be compositionally defined in 2/3Lisp, but the resulting account is theoretically barren. Stripped of what matters, analysis loses any account of what the system implements, or what norms the system is enjoined to honor. A purely effective account (pure ψ, without φ) treats its subject matter as stuff manipulation, losing the warrant for calling the system computational.

It is not the mandate of this book to lay out a new framework for computational analysis. It is not even its mandate to identify the concepts in terms of which the framework will be describable, except at the broadest level of abstractions ("an integrated account of input, output, and state, subject to overarching conditions of causal efficacy and material realizability"). Needless to say, the development of such accounts, and of ontological frameworks requisite for their articulation, will be an enormous task, requiring investigation and time to develop. But the examples adduced throughout the present volume should give an indication of the directions where a theory of computation must develop if it is to live up to its name.

Postscript—Naturalizing the Non-effective

This is not the end of the story.

Regularities involving non-effective relations have been a prime actor in this drama. In particular, we have seen that non-effective semantic relations are central to any notion of computing. Yet non-effective relations are not primitively supported in what we take to be the ultimate structure of the universe. That raises some foundational questions. What are these things, how do we know about

them, how are they sustained?

The first question does not seem particularly metaphysically challenging. Non-effective relations are presumably constituted by relational facts among potentially large numbers of physically effective properties and relations. As regards the second—how we know about them—almost by definition, non-effective relations must be correlated somehow, albeit possibly in extremely complex ways, between and among aspects of the web of basic effective properties. Non-effective relations could therefore be represented by ineffably complex abstractions over the huge number of weights in a trained network model. It is the third question that is challenging: what maintains them—what ensures that those representations are correct?

Equation E1 from §4.V4 is a technical expression indicating that the simplification process is designation-preserving. Designation is a non-effective relation, yet the architecture of 2/3Lisp shows, at least in a simple context, how a strictly effective computational process can be made to honor constraints that, in and of themselves, are non-effective. In a similar way, human cognition demonstrates that, taken as a whole, our brain processes, which from one point of view must be constituted purely effectively, can nevertheless deferentially honor non-effective relations with distal parts of the world. An adequate account of how this occurs will undoubtedly be extremely elaborate, involving neuroscience, evolutionary psychology, and much more—perhaps including the normative standards of objectivity. While our minds constitute an existence proof that it is possible, however, at least to date they provide little practical guidance on how such a non-effectively deferential system might be constructed.

Computation offers a potential path toward an answer. A complete picture of computation will require an account that discharges the naturalistic debts incurred by bringing at least some non-effective relations forward into view. While nothing can affect the non-effective directly, if a system's non-effective properties or relations can be effectively represented, or if the system can be attuned to effective correlates of non-effective properties or relations, it should become possible to alter the status of what is non-effective indirectly, by adjusting that with which what is non-effective is correlated. In a sense,

that is, at least in principle the non-effective could be adjusted via something like correlated inference.

I have not told any such story here, but something of its sort will need to be part of any adequate successor theory.

———————————●●———————————

Epilogue

The impossibility of understanding what it would be like not to be able to think can blind us to the achievement of concretely realized thinking. Of the surpassing abundance and ontological richness in the world, only a tiny subset—a subset of measure zero, to use the set theorists' phrase—meets the conditions of being mechanically effective. Or to put it in plain language: *almost nothing is effective*. Being the shirt my grandmother sewed for me, being a thousand miles from Abilene, being about to receive a letter from a long-lost friend, being the far-away galaxy that my 27th-century descendants will visit—none of these properties are effective, none are of the sort that, so described, could be harnessed to run an electric motor or turn on a mechanical switch.

To take another example, relevant to the assessments of computation and reflection, so too is *being referred to* not an effective property. It is especially true that being referred to *right now* is not effective. In spite of its being more than two million light years away, the Andromeda galaxy at this very moment enjoys the property of being so referred to—of being the referent of a thought, since I am thinking about it as I write this sentence. It has the property of being thought about right now, in spite of the fact that no evidence of that fact could reach Andromeda until more than two million years from now, to say nothing of the fact that its currently being thought about will not, per se, exert a causal or effective impact on (send an "effective signal to") it, *ever*.

In fact it is metaphysically astonishing, to say nothing of deucedly lucky, that *anything* is effective—that it is in virtue of

the exemplification of any property that anything can have any causal consequence whatsoever.

The stupefying paucity of the causal or effective, I take it, establishes a ferocious challenge for humans, for logic, and for computation—for intentional systems of all sorts. It is a challenge that can never be fully met. The logician Jon Barwise was fond of saying that the most important conclusion of the 20th century study of logic was appreciation of the fact that "ultimately one cannot reduce semantics to syntax." He took this to be the underlying moral of the failure of Hilbert's program, the Gödel incompleteness results, etc., to which I would add Turing's demonstrations of the limits of computability. Given my belief that what it is to be "syntactic" (in logic) and what it is to be "computable" (in computer science) both ultimately depend on a similar notion of effectiveness, what I take these results to amount to is something of a Browning-esque recognition that even in the radically restricted domains of mathematics, logic, and formal computability, the challenge of "grasping the world" through effective mechanism is not one that can ultimately be met. We simply do the best we can. And this humility holds for formal results in arithmetic! Think about how much more daunting is the challenge when the semantic realm is opened up to the full richness of dynamic real-world phenomena.

The challenge ultimately derives from physics.[1] If anything is to happen—i.e., for anything like an event to occur—it must come about, as we often say, through causal means—must happen in virtue of the dynamic exemplification of causal, or mechanical, or effective properties. This is just a blunt, inescapable

---

[1] I am not a straightforward physicalist—even of the weakest form so far articulated: global supervenience. But just as I believe that there is something profoundly right about realism, which must be preserved in any more constructivist or embodied alternative, so too I believe that there is something both incredibly sobering and yet surpassingly powerful about physicalism, which must also be, if not preserved, then at least done justice to in any successor or alternative account.

fact about what the world is like. It is a fundamental truth—albeit one that blanket mechanism has blown out of proportion. Blanket mechanists are right to recognize that *what happens* does so in virtue of the mechanical or effective. What they fail to appreciate it that *what is the case* is transcendently larger.

Moreover, the challenge of *knowing*—of reasoning, of computing, of being able to think—derives from the discrepancy between the vanishingly small and restricted subset of the world that is mechanical or effective, and the vastly larger world of what is the case. Knowing has to happen, reasoning has to happen—we do not come blessed with divinely instilled or preordained comprehensive knowledge. What we humans have succeeded in doing, by evolutionary and then societal and cultural means, is to learn how to exploit that which is effective or mechanistic in ourselves, and in our environment, in ways that allow us to stand in referential (and, because of the deference, to an extent reverential) relation to the world as a whole. An ability to appreciate the magnitude of this achievement is what I take to be so important about studies of logic, of computing, of intentionality in general. It is also what I take formal logic to be a magnificent first stab at explaining. And this, too, is what computing is a complex practice exploring (in spite of how we have understood it theoretically). That it is metaphysically possible at all, as I keep saying, is incredibly fortuitous. That humans have evolved so as to be able to do it is undoubtedly our most staggering achievement. That we are slowly coming to understand what it involves, and, Prometheus-like, in systems of our own devising, are starting to construct synthetic instances of it—that is what makes computing important, what makes the arrival of computing on the world stage a development of profound historic consequence.

This volume is not a metaphysical defense of deferential semantics, let alone of an underlying view of global physical supervenience; nor is it a systematic account of the nature of computing or reasoning in its terms. It merely documents a few small steps taken en route to understanding what it would be to develop such a story. Perhaps most critical to understand is

that 2Lisp and 3Lisp, modest as they are, were designed within a committedly deferential overarching semantic viewpoint. Because of this, the individual moves made in their design—down to the most intricate details of 3Lisp programming—can be understood only against its strictures.

As should be obvious, these are all points on which the blanket mechanist is sentenced to blindness. Because blanket mechanists assume that the world is restricted to the effective, they cannot grasp the importance of logic, the character of computing, the essence of knowledge, or the nature of being. To recognize the importance of deferential semantics, in contrast, is to leave the world whole and vast, and to recognize that the mechanical restriction applies only to what happens, effectively, in a very particular and limited sense of "what happens."

Blanket mechanists, I know, found 3Lisp inscrutable when it was first introduced, and likely still find it inscrutable today. I only hope that the few remarks made here will help to make its architecture, and programs and computing more generally, a tad more comprehensible.

References

Abelson, Harold, Gerald Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, 1984.

Ancona, Massimo, Walter Cazzola, and Eduardo B. Fernandez. "Reflective Authorization Systems." In *Proceedings of ECOOP Workshop on Distributed Object Security (EWDOS'98).* Unité de Recherche INRIA Rhône-Alpes, 1998: 35–39.

Amin, Nada, and Tiark Rompf. "Collapsing Towers of Interpreters." In *Proceedings of the ACM on Programming Languages* 2, Issue POPL. Association for Computing Machinery, 2017: 1–33.

Asai, Kenichi. "The Reflective Language Black." PhD diss., University of Tokyo, Department of Information Science, 1997.

Austin, John L. *How to Do Things With Words: The William James Lectures Delivered at Harvard University in 1955, Second Edition.* Edited by Maria Sbisà and J. O. Urmson. Harvard University Press, 1975.

Barwise, Jon, and John Etchemendy. *The Liar: An Essay on Truth and Circularity.* Oxford University Press, 1992.

Bobrow, Daniel G., and Terry Winograd. "An Overview of KRL, a Knowledge Representation Language." *Cognitive Science* 1, no. 1 (1977): 3-46.

Bobrow, Daniel G., and Terry Winograd. "Experience with

KRL-O: One Cycle of a Knowledge Representation Language." In *IJCAI'77: Proceedings of the 5th International Joint Conference on Artificial Intelligence - Volume 1*. Morgan Kaufmann Publishers Inc., 1977: 213–222.

Burgin, Mark. "Generalized Kolmogorov Complexity and Duality in Theory of Computations." *Notices of the Russian Academy of Sciences* 25, no. 3 (1982): 19–23.

Cazzola, Walter, Andrea Sosio, and Francesco Tisato. "Shifting Up Reflection from the Implementation to the Analysis Level." In *Reflection and Software Engineering, Lecture Notes in Computer Science* 1826. Springer, June 2000: 1–20.

Chaitin, Gregory J. "On the Simplicity and Speed of Programs for Computing Infinite Sets of Natural Numbers." *Journal of the ACM* 16, no. 3 (1969): 407–422.

Chisholm, Roderick M. *Perceiving: A Philosophical Study*. Cornell University Press, 1957.

Daintith, John and Edmund Wright. "Effective Computability." In *A Dictionary of Computing, 6th Edition*. Oxford University Press, 2008.

Danvy, Olivier and Karoline Malmkjaer. "Intensions and Extensions in a Reflective Tower." In *LFP '88: Proceedings of the 1988 ACM Conference on LISP and Functional Programming*. Association for Computing Machinery, 1988: 327–341.

Demers, Francois-Nicola and Jacques Malenfant. "Reflection in Logic, Functional and Object-oriented Programming: A Short Comparative Study." In *Proceedings of the IJCAI'95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*. August 1995. 29–38.

des Rivières, Jim, and Brian Cantwell Smith. "The

Implementation of Procedurally Reflective Languages." In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*. Association for Computing Machinery, 1984: 331–347.

Dretske, Frederick I. "Machines and the Mental." Presidential Address to the 83rd Annual Meeting of the Western Division of the American Philosophical Association, Chicago: *APA Proceedings* 59 (1985): 23–33.

Dupuy, Jean-Pierre. *Aux Origines des Sciences Cognitives.* La Découverte, 1994.

Evans, Gareth. *The Varieties of Reference.* Oxford University Press, 1982.

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

Fodor, Jerry A. "Methodological Solipsism Considered as a Research Strategy in Cognitive Psychology." *Behavioral and Brain Sciences* 3, no. 1 (1980): 63–73.

Fodor, Jerry A. *Psychosemantics.* The MIT Press, 1987.

Fodor, Jerry A. "Semantics, Wisconsin Style." *Synthese* 59: 231–250.

Frege, Gottlob. "Über Sinn und Bedeutung", *Zeitschrift für Philosophie und Philosophische Kritik* 100 (1892): 25–50.

Friedman, Daniel P., and Mitchell Wand. "Reification: Reflection without Metaphysics." In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and Functional Programming.* Association for Computing Machinery, 1984: 348–355.

Glenman, Stuart. *The New Mechanical Philosophy.* Oxford University Press, 2017.

Haugeland, John. "Analog and Analog." *Philosophical Topics* 12, no. 1 (1981).

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

Haugeland, Jonn. "Representational Genera." In *Philosophy and Connectionist Theory,* edited by William Ramsay, Stephen P. Stitch, David E. Rumelhart. Lawrence Erlbaum Associates, 1991.

Haugeland, John. "Semantic Engines." In *Mind Design: Philosophy, Psychology, Artificial Intelligence,* edited by John Haugeland. The MIT Press / A Bradford Book, 1981.

Kolmogorov, Andrey. "On Tables of Random Numbers." *Sankhyā: The Indian Journal of Statistics, Series A (1961-2002)* 25, no. 4 (1963): 369–375.

Kolmogorov, Andrey. "On Tables of Random Numbers." *Theoretical Computer Science* 207, no. 2 (1998): 387–395.

Levin, Leonid A. "Some Theorems on the Algorithmic Approach to Probability Theory and Information Theory." *Annals of Pure and Applied Logic* 162, no. 3 (2010): 224–235.

McAllister, Linda. "Chisholm and Brentano on Intentionality." *Review of Metaphysics* XXVIII, no. 2 (1974): 328–338.

McCarthy, John, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart and Michael I. Levin. *Lisp 1.5 Programmer's Manual.* The MIT Press, 1962.

Minsky, Marvin L. "Computability, Effective Procedures, And Algorithms. Infinite Machines." In *Computation: Finite and Infinite Machines.* Prentice-Hall, Inc., 1967: 103-132.

Muller, Robert. "M-LISP: A Representation Independent Dialect of LISP with Reduction Semantics." *ACM Transactions on Programming Languages and Systems (TOPLAS)* 14, no. 4 (1992): 589–616.

Newell, Allen. "The Knowledge Level." *Artificial Intelligence* 18, no. 1: 81–132.

Newell, Allen. "Physical Symbol Systems," *Cognitive Science* 4 (1980): 135–183.

Newell, Alan, and Herbert A. Simon. "Computer Science as Empirical Inquiry." *Communications of the Association for Computing Machinery* 19 (1976): 113–126.

Newton, Isaac. *Scholium to the Definitions in Philosophiae Naturalis Principia Mathematica, Bk 1.* Translated by Andrew Motte (1729). Rev. Florian Cajori. University of California Press, 1934: 6–12.

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

Rey, G.. "Functionalism." *Encyclopedia Britannica*, August 9, 2024. https://www.britannica.com/topic/philosophy-of-mind/Functionalism.

Searle, John. *Intentionality: An Essay in the Philosophy of Mind.* Cambridge University Press, 1983.

Searle, John. "Minds, Brains, and Programs." *Behavioral and Brain Sciences* 3, no. 3 (1980): 417-458.

Searle, John. *Minds, Brains, and Science.* Harvard University Press, 1984.

Smith, Brian Cantwell. "The Correspondence Continuum." Report No. CSLI-87-71. Center for the Study of Language and Information, Stanford University, 1987.

Smith, Brian Cantwell. "3Lisp Legacy," forthcoming. Online version available at http://ageofsignficance.org/legacy

Smith, Brian Cantwell. *On the Origin of Objects.* The MIT Press, 1996.

Smith, Brian Cantwell. "One Hundred Billion Lines of C++." *Cog Sci News*, Lehigh University, 1997.

Smith, Brian Cantwell. "Procedural Reflection in Programming Languages." PhD diss., Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1982.

Smith, Brian Cantwell. *The Promise of Artificial Intelligence: Reckoning and Judgment.* The MIT Press, 2019.

Smith, Brian Cantwell. "Reflection and Semantics in Lisp." In *Proceedings of the 11th Annual Principles of Programming Languages Conference (POPL).* 1984.

Smith, Brian Cantwell. "Rehabilitating Representation," unpublished manuscript, March 19, 1998.

Smith, Brian Cantwell. "Solving the Halting Problem, and Other Skullduggery in the Foundations of Computing," unpublished manuscript, 2014.

Smith, Brian Cantwell. "Varieties of Self-Reference." In *Theoretical Aspects of Reasoning about Knowledge: Proceedings of the 1986 Conference,* edited by Joseph Y. Halpern, 1986.

Smith, Brian Cantwell and Jim des Rivières. "Interim 3-LISP Reference Manual." ISL-1, Intelligent Systems Laboratory, June, 1984.

Smullyan, Raymond. *What Is the Name of This Book?: The Riddle of Dracula and Other Logical Puzzles.* Prentice-Hall, 1978.

Steele, Guy Lewis Jr. "Compiler Optimization Based on Viewing LAMBDA as RENAME + GOTO." In *Artificial Intelligence: An MIT Perspective,* edited by Patrick Henry Winston and Richard H. Brown, The MIT Press, 1980.

Steele, Guy Lewis Jr. "Debunking the 'Expensive Procedure

Call' Myth or, Procedure Call Implementations Considered Harmful or, Lambda: The Ultimate GOTO." AI Memo No. 443. Massachusetts Institute of Technology Artificial Intelligence Laboratory, October, 1977.

Steele, Guy Lewis Jr. "Lambda: The Ultimate Declarative." AI Memo No. 379. Massachusetts Institute of Technology Artificial Intelligence Laboratory, November, 1976.

Steele, Guy Lewis Jr. "Rabbit: A Compiler for Scheme." AI Memo No. 474. Massachusetts Institute of Technology Artificial Intelligence Laboratory, May, 1978.

Steele, Guy Lewis Jr., and Gerald Jay Sussman. "Design of Lisp-based Processors, or Scheme: A Dielectric Lisp, or Finite Memories Considered Harmful, or Lambda: The Ultimate Opcode." AI Memo No. 514. Massachusetts Institute of Technology Artificial Intelligence Laboratory, March, 1979.

Steele, Guy Lewis Jr., and Gerald Jay Sussman. "Lambda: The Ultimate Imperative." AI Memo No. 353. Massachusetts Institute of Technology Artificial Intelligence Laboratory, March, 1976.

Stein, Gertrude. *How To Write.* Sun & Moon Press, 1931.

Sussman, Gerald Jay, and Guy Lewis Steele Jr. "The Art of the Interpreter or, the Modularity Complex." AI Memo No. 453. Massachusetts Institute of Technology Artificial Intelligence Laboratory, May, 1978.

Sussman, Gerald Jay, and Guy Lewis Steele Jr. "Scheme: An Interpreter For Extended Lambda Calculus". AI Memo No. 349. Massachusetts Institute of Technology Artificial Intelligence Laboratory, December, 1975.

Turing, Alan M. "On Computable Numbers, with an

Application to the Entscheidungsproblem." *Proceedings of the London Mathematical Society Series 2* 42, no. 1 (1937): 230–265.

Van Gelder, Tim. "What Might Cognition Be, If Not Computation?" *The Journal of Philosophy* 92, no. 7 (1995): 345–381.

Wand, Mitchell, and Daniel P. Friedman. "The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower." In *LFP '86: Proceedings of the 1986 ACM Conference on LISP and Functional programming.* Association for Computing Machinery, 1986: 298–307.

Wikimedia Foundation. "Reflective programming." Last modified September 8, 2024, 09:02 (UTC). https://en.wikipedia.org/wiki/Reflective_programming.